Oracle® Streams

Replication Administrator's Guide

10*g* Release 2 (10.2)

B14228-01

June 2005

ORACLE®

Oracle Streams Replication Administrator's Guide, 10*g* Release 2 (10.2)

B14228-01

Primary Author: Randy Urbano

Contributors: Nimar Arora, Lance Ashdown, Ram Avudaiappan, Sukanya Balaraman, Neerja Bhatt, Ragamayi Bhyravabhotla, Diego Cassinera, Debu Chatterjee, Alan Downing, Lisa Eldridge, Curt Elsbernd, Yong Feng, Jairaj Galagali, Brajesh Goyal, Sanjay Kaluskar, Lewis Kaplan, Anand Lakshminath, Jing Liu, Edwina Lu, Raghu Mani, Pat McElroy, Krishnan Meiyyappan, Shailendra Mishra, Bhagat Nainani, Anand Padmanaban, Kashan Peyetti, Maria Pratt, Arvind Rajaram, Viv Schupmann, Vipul Shah, Neeraj Shodhan, Wayne Smith, Benny Souder, Jim Stamos, Janet Stern, Mahesh Subramaniam, Bob Thome, Hung Tran, Ramkumar Venkatesan, Byron Wang, Wei Wang, James M. Wilson, Lik Wong, David Zhang

Contents

3 Streams Conflict Resolution

4 Streams Tags

5 Streams Heterogeneous Information Sharing

Part III Administering Streams Replication

9 Managing Capture, Propagation, and Apply

10 Performing Instantiations

11 Managing Logical Change Records (LCRs)

15 Single-Source Heterogeneous Replication Example

16 Multiple-Source Replication Example

Part V Appendixes

A Migrating Advanced Replication to Streams

Index

Preface

Oracle Streams Replication Administrator's Guide describes the features and functionality of Streams that can be used for data replication. This document contains conceptual information about Streams replication, along with information about configuring and managing a Streams replication environment.

This Preface contains these topics:

Audience

Oracle Streams Replication Administrator's Guide is intended for database administrators who create and maintain Streams replication environments. These administrators perform one or more of the following tasks

- Plan for a Streams replication environment
- Configure a Streams replication environment
- Configure conflict resolution in a Streams replication environment
- Administer a Streams replication environment
- Monitor a Streams replication environment
- Perform necessary troubleshooting activities for a Streams replication environment

To use this document, you need to be familiar with relational database concepts, SQL, distributed database administration, general Oracle Streams concepts, Advanced Queuing concepts, PL/SQL, and the operating systems under which you run a Streams environment.

Documentation Accessibility

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

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see these Oracle resources:

- *Oracle Streams Concepts and Administration*
- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database SQL Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL User's Guide and Reference*
- *Oracle Database Utilities*
- *Oracle Database Heterogeneous Connectivity Administrator's Guide*
- Streams online help for the Streams tool in Oracle Enterprise Manager

Many of the examples in this book use the sample schemas of the sample database, which is installed by default when you install Oracle Database. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

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

http://www.oracle.com/technology/membership/

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

http://www.oracle.com/technology/documentation/

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.

Part I

Streams Replication Concepts

This part describes conceptual information about Streams replication and contains the following chapters:

1

Understanding Streams Replication

This chapter contains conceptual information about Streams replication. This chapter contains these topics:

- Overview of Streams Replication
- Capture and Streams Replication
- Propagation and Streams Replication
- Apply and Streams Replication

> **See Also:** *Oracle Streams Concepts and Administration* for general information about Oracle Streams. This document assumes that you understand the concepts described in *Oracle Streams Concepts and Administration*.

Overview of Streams Replication

Replication is the process of sharing database objects and data at multiple databases. To maintain replicated database objects and data at multiple databases, a change to one of these database objects at a database is shared with the other databases. In this way, the database objects and data are kept synchronized at all of the databases in the replication environment. In a Streams replication environment, the database where a change originates is called the **source database**, and a database where a change is shared is called a **destination database**.

When you use Streams, replication of a DML or DDL change typically includes three steps:

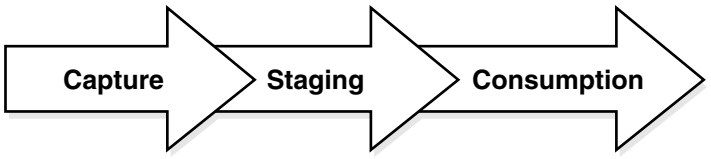
1. A capture process or an application creates one or more logical change records (LCRs) and enqueues them into a queue. An LCR is a message with a specific format that describes a database change. A capture process reformats changes captured from the redo log into LCRs, and applications can construct LCRs. If the change was a data manipulation language (DML) operation, then each LCR encapsulates a row change resulting from the DML operation to a shared table at the source database. If the change was a data definition language (DDL) operation, then an LCR encapsulates the DDL change that was made to a shared database object at a source database.
2. A propagation propagates the staged LCR to another queue, which usually resides in a database that is separate from the database where the LCR was captured. An LCR can be propagated to a number of queues before it arrives at a destination database.

3. At a destination database, an apply process consumes the change by applying the LCR to the shared database object. An apply process can dequeue the LCR and apply it directly, or an apply process can dequeue the LCR and send it to an apply handler. In a Streams replication environment, an apply handler performs customized processing of the LCR and then applies the LCR to the shared database object.

Step 1 and Step 3 are required, but Step 2 is optional because, in some cases, an application can enqueue an LCR directly into a queue at a destination database. In addition, in a heterogeneous replication environment in which an Oracle database shares information with a non-Oracle database, an apply process can apply changes directly to a non-Oracle database without propagating LCRs.

Figure 1–1 illustrates the information flow in a Streams replication environment.

Figure 1–1 Streams Information Flow



This document describes how to use Streams for replication and includes the following information:

- Conceptual information relating to Streams replication
- Information about configuring a Streams replication environment
- Instructions for administering, monitoring, and troubleshooting a Streams replication environment
- Demonstration scripts that create and maintain example Streams replication environments

Replication is one form of information sharing. Oracle Streams enables replication, and it also enables other forms of information sharing, such as messaging, event management and notification, data warehouse loading, and data protection.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about the other information sharing capabilities of Streams

Rules in a Streams Replication Environment

A **rule** is a database object that enables a client to perform an action when an event occurs and a condition is satisfied. Rules are evaluated by a **rules engine**, which is a built-in part of Oracle. You use rules to control the information flow in a Streams replication environment. Each of the following mechanisms is a client of the rules engine:

- Capture process
- Propagation
- Apply process

You control the behavior of each of these Streams clients using rules. A **rule set** contains a collection of rules, and you can associate a positive and a negative rule set with a Streams client. In a replication environment, a Streams client performs an action if an LCR satisfies its rule sets. In general, a change satisfies the rule sets for a Streams client if *no rules* in the negative rule set evaluate to `TRUE` for the LCR, and *at least one rule* in the positive rule set evaluates to `TRUE` for the LCR. If a Streams client is associated with both a positive and negative rule set, then the negative rule set is always evaluated first.

Specifically, you control the information flow in a Streams replication environment in the following ways:

- Specify the changes that a capture process captures from the redo log or discards. That is, if a change found in the redo log satisfies the rule sets for a capture process, then the capture process captures the change. If a change found in the redo log does not satisfy the rule sets for a capture process, then the capture process discards the change.
- Specify the LCRs that a propagation propagates from one queue to another or discards. That is, if an LCR in a queue satisfies the rule sets for a propagation, then the propagation propagates the LCR. If an LCR in a queue does not satisfy the rule sets for a propagation, then the propagation discards the LCR.
- Specify the LCRs that an apply process retrieves from a queue or discards. That is, if an LCR in a queue satisfies the rule sets for an apply process, then the LCR is retrieved and processed by the apply process. If an LCR in a queue does not satisfy the rule sets for an apply process, then the apply process discards the LCR.

You can use the Oracle-supplied `DBMS_STREAMS_ADM` PL/SQL package to create rules for a Streams replication environment. You can specify these system-created rules at the following levels:

- Table - Contains a rule condition that evaluates to `TRUE` for changes made to a particular table
- Schema - Contains a rule condition that evaluates to `TRUE` for changes made to a particular schema
- Global - Contains a rule condition that evaluates to `TRUE` for all changes made to a database

In addition, a single system-created rule can evaluate to `TRUE` for DML changes or for DDL changes, but not both. So, for example, if you want to replicate both DML and DDL changes to a particular table, then you need both a table-level DML rule and a table-level DDL rule for the table.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about how rules are used in Streams

Nonidentical Replicas with Streams

Streams replication supports sharing database objects that are not identical at multiple databases. Different databases in the Streams environment can contain shared database objects with different structures. You can configure rule-based transformations during capture, propagation, or apply to make any necessary changes to LCRs so that they can be applied at a destination database. In Streams replication, a **rule-based transformation** is any modification to an LCR that results when a rule in a positive rule set evaluates to `TRUE`.

For example, a table at a source database can have the same data as a table at a destination database, but some of the column names can be different. In this case, a rule-based transformation can change the names of the columns in LCRs from the source database so that they can be applied successfully at the destination database.

There are two types of rule-based transformations: declarative and custom. **Declarative rule-based transformations** cover a set of common transformation scenarios for row LCRs, including renaming a schema, renaming a table, adding a column, renaming a column, and deleting a column. You specify (or declare) such a transformation using a procedure in the `DBMS_STREAMS_ADM` package. Streams performs declarative transformations internally, without invoking PL/SQL

A **custom rule-based transformation** requires a user-defined PL/SQL function to perform the transformation. Streams invokes the PL/SQL function to perform the transformation. A custom rule-based transformation can modify either captured or user-enqueued messages, and these messages can be LCRs or user messages. For example, a custom rule-based transformation can change the datatype of a particular column in an LCR. A custom rule-based transformation must be defined as a PL/SQL function that takes an `ANYDATA` object as input and returns an `ANYDATA` object.

Rule-based transformations can be done at any point in the Streams information flow. That is, a capture process can perform a rule-based transformation on a change when a rule in its positive rule set evaluates to `TRUE` for the change. Similarly, a propagation or an apply process can perform a rule-based transformation on a message when a rule in its positive rule set evaluates to `TRUE` for the message.

Note: Throughout this document, "rule-based transformation" is used when the text applies to both declarative and custom rule-based transformations. This document distinguishes between the two types of rule-based transformations when necessary.

See Also: *Oracle Streams Concepts and Administration* for more information about rule-based transformations

Subsetting with Streams

Streams also supports subsetting of table data through the use of subset rules. If a shared table in a database in a Streams replication environment contains only a subset of data, then you can configure Streams to manage changes to a table so that only the appropriate subset of data is shared with the subset table. For example, a particular database can maintain data for employees in a particular department only. In this case, you can use subset rules to share changes to the data for employees in that department with the subset table, but not changes to employees in other departments.

Subsetting can be done at any point in the Streams information flow. That is, a capture process can use a subset rule to capture a subset of changes to a particular table, a propagation can use a subset rule to propagate a subset of changes to a particular table, and an apply process can use a subset rule to apply only a subset of changes to a particular table.

See Also: *Oracle Streams Concepts and Administration* for more information subset rules

Capture and Streams Replication

To maintain replicated database objects and data, you must capture changes made to these database objects and their data. Next, you must share these changes with the databases in the replication environment. In a Streams replication environment, you can capture changes in either of the following ways:

- Change Capture Using a Capture Process
- Change Capture Using a Custom Application

Change Capture Using a Capture Process

This section contains a brief overview of the capture process and conceptual information that is important for a capture process in a replication environment.

> **See Also:** *Oracle Streams Concepts and Administration* for general conceptual information about a capture process

Capture Process Overview

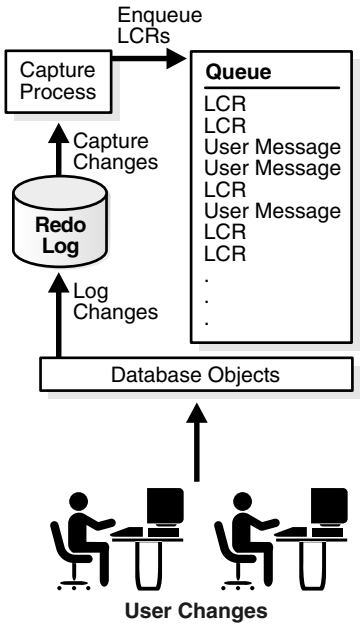
Changes made to database objects in an Oracle database are logged in the redo log to guarantee recoverability in the event of user error or media failure. A capture process is an Oracle background process that reads the database redo log to capture DML and DDL changes made to database objects. The source database for a change that was captured by a capture process is always the database where the change was generated in the redo log. A capture process formats these changes into messages called LCRs and enqueues them into a queue. Because a running capture process automatically captures changes based on its rules, change capture using a capture process is sometimes called **implicit capture**.

There are two types of LCRs: a **row LCR** contains information about a change to a row in a table resulting from a DML operation, and a **DDL LCR** contains information about a DDL change to a database object. You use rules to specify which changes are captured. A single DML operation can change more than one row in a table. Therefore, a single DML operation can result in more than one row LCR, and a single transaction can consist of multiple DML operations.

Changes are captured by a **capture user**. The capture user captures all DML changes and DDL changes that satisfy the capture process rule sets.

A capture process can capture changes locally at the source database, or it can capture changes remotely at a downstream database. Figure 1–2 illustrates a local capture process.

Figure 1–2 Local Capture Process



Downstream capture means that a capture process runs on a database other than the source database. The following types of configurations are possible for a **downstream capture process**:

- A **real-time downstream capture** configuration means that redo transport services use the log writer process (LGWR) at the source database to send redo data from the online redo log to the downstream database. At the downstream database, a remote file server process (RFS) receives the redo data and stores it in the standby redo log, and the archiver at the downstream database archives the redo data in the standby redo log. The real-time downstream capture process captures changes from the standby redo log whenever possible and from the archived redo log whenever necessary.
- An **archived-log downstream capture** configuration means that archived redo log files from the source database are copied to the downstream database, and the capture process captures changes in these archived redo log files. You can copy the archived redo log files to the downstream database using redo transport services, the `DBMS_FILE_TRANSFER` package, file transfer protocol (FTP), or some other mechanism.

Figure 1–3 illustrates a real-time downstream capture process.

Figure 1–3 Real-Time Downstream Capture

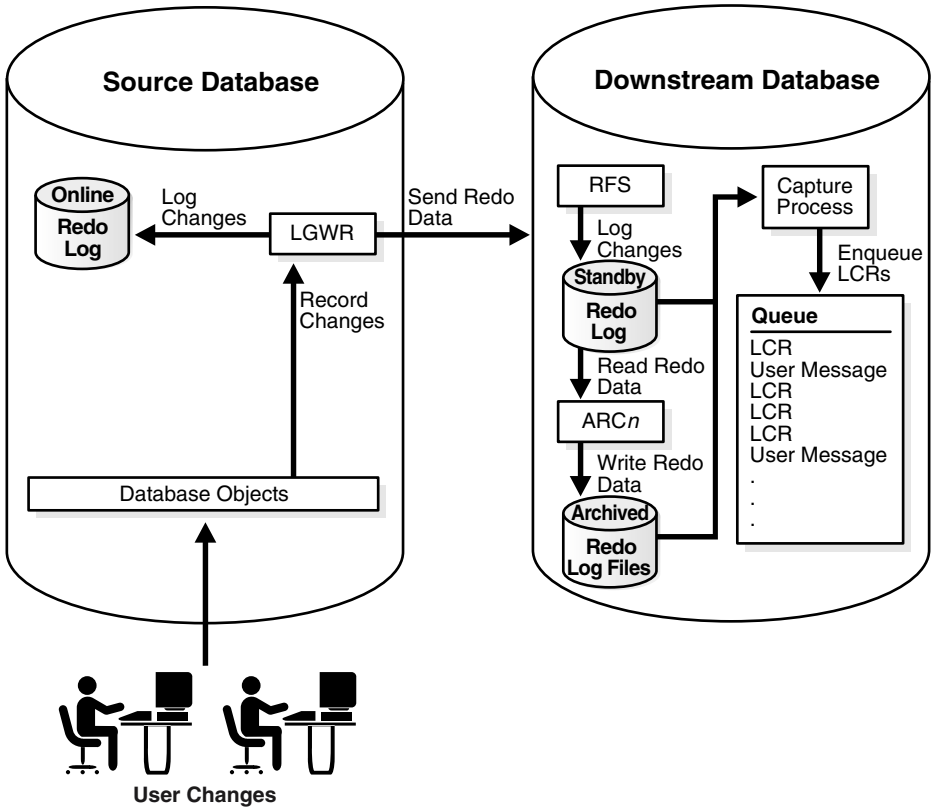
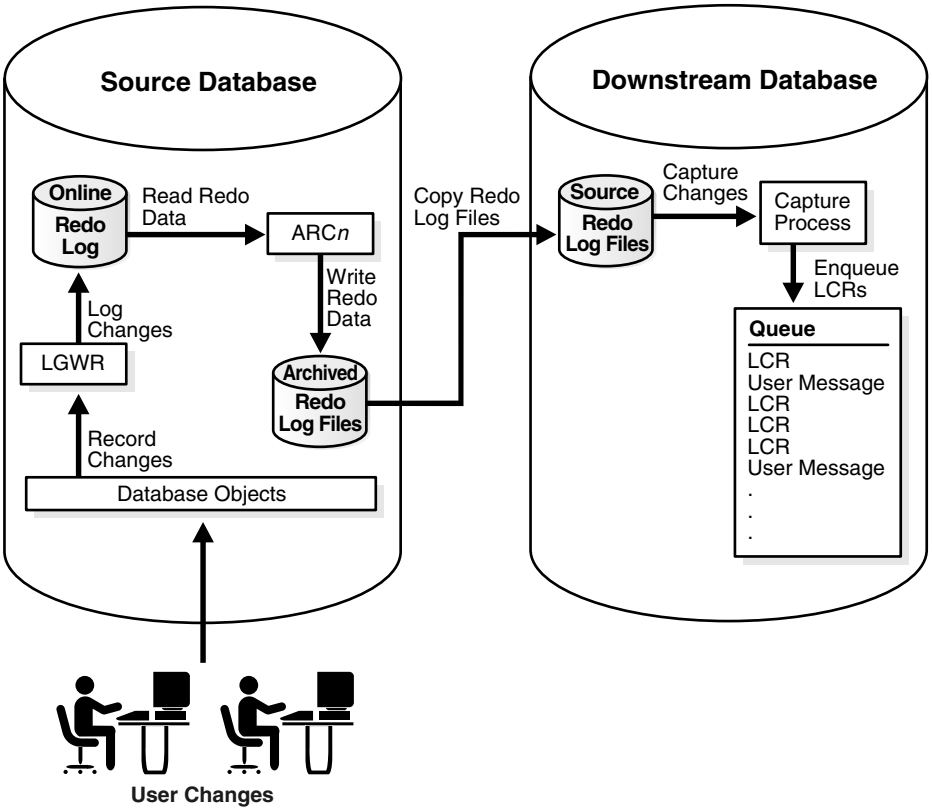


Figure 1–4 illustrates an archived-log downstream capture process.

Figure 1–4 Archived-Log Downstream Capture Process



A local capture process reads the online redo log whenever possible and archived redo log files otherwise. A real-time downstream capture process reads the standby redo log whenever possible and archived standby redo log files otherwise. An archived-log downstream capture process always reads archived redo log files from the source database.

Note:

- As illustrated in Figure 1–4, the source database for a change captured by a downstream capture process is the database where the change was recorded in the redo log, not the database running the downstream capture process.
- References to "downstream capture processes" in this document apply to both real-time downstream capture processes and archived-log downstream capture processes. This document distinguishes between the two types of downstream capture processes when necessary.

Supplemental Logging for Streams Replication

Supplemental logging places additional column data into a redo log whenever an operation is performed. The capture process captures this additional information and places it in LCRs. Supplemental logging is always configured at a source database, regardless of the location of the capture process that captures changes to the source database.

There are two types of supplemental logging: **database supplemental logging** and **table supplemental logging**. Database supplemental logging specifies supplemental logging for an entire database, while table supplemental logging enables you to specify log groups for supplemental logging of a particular table. If you use table supplemental logging, then you can choose between two types of log groups: unconditional log groups and conditional log groups.

Unconditional log groups log the before images of specified columns when the table is changed, regardless of whether the change affected any of the specified columns. Unconditional log groups are sometimes referred to as "always log groups."
Conditional log groups log the before images of all specified columns only if at least one of the columns in the log group is changed.

Supplementing logging at the database level, unconditional log groups at the table level, and conditional log groups at the table level together determine which old values are logged for a change.

If you plan to use one or more apply processes to apply LCRs captured by a capture process, then you must enable supplemental logging *at the source database* for the following types of columns in tables *at the destination database*:

- Any columns at the source database that are used in a primary key in tables for which changes are applied at a destination database must be unconditionally logged in a log group or by database supplemental logging of primary key columns.
- If the parallelism of any apply process that will apply the changes is greater than 1, then any indexed column at a destination database that comes from one or more columns at the source database must be unconditionally logged. Index columns include unique or non unique index columns, foreign key columns, and bitmap index columns.
- Any columns at the source database that are used as substitute key columns for an apply process at a destination database must be unconditionally logged. You specify substitute key columns for a table using the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package.
- The columns specified in a column list for conflict resolution during apply must be conditionally logged if more than one column at the source database is used in the column list at the destination database.
- Any columns at the source database that are used by a DML handler or error handler at a destination database must be unconditionally logged.
- Any columns at the source database that are used by a rule or a rule-based transformation must be unconditionally logged.
- Any columns at the source database that are specified in a value dependency virtual dependency definition at a destination database must be unconditionally logged.
- If you specify row subsetting for a table at a destination database, then any columns at the source database that are in the destination table or columns at the source database that are in the subset condition must be unconditionally logged. You specify a row subsetting condition for an apply process using the `dml_condition` parameter in the `ADD_SUBSET_RULES` procedure in the `DBMS_STREAMS_ADM` package.

If you do not use supplemental logging for these types of columns at a source database, then changes involving these columns might not apply properly at a destination database.

Note: LOB, LONG, LONG RAW, and user-defined type columns cannot be part of a supplemental log group.

See Also:

- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3
- "Monitoring Supplemental Logging" on page 12-2
- "Considerations for Applying DML Changes to Tables" on page 1-18 for more information about apply process behavior that might require supplemental logging at the source database
- "Column Lists" on page 3-9 for more information about supplemental logging and column lists
- "Virtual Dependency Definitions" on page 1-15 for more information about value dependencies
- *Oracle Streams Concepts and Administration* for more information about rule-based transformations

Change Capture Using a Custom Application

A custom application can capture the changes made to a Oracle database by reading from transaction logs, by using triggers, or by some other method. The application must assemble and order the transactions and must convert each change into an LCR. Next, the application must enqueue the LCRs into a queue in an Oracle database using the DBMS_STREAMS_MESSAGING package or the DBMS_AQ package. The application must commit after enqueuing all LCRs in each transaction.

Because the LCRs are constructed and enqueued manually by a user or application, change capture that manually enqueues constructed LCRs is sometimes called **explicit capture**. If you have a heterogeneous replication environment where you must capture changes at a non-Oracle database and share these changes with an Oracle database, then you can create a custom application to capture changes made to the non-Oracle database.

See Also:

- "Non-Oracle to Oracle Data Sharing with Streams" on page 5-8
- "Constructing and Enqueuing LCRs" on page 11-2

Propagation and Streams Replication

In a Streams replication environment, propagations propagate captured changes to the appropriate databases so that changes to replicated database objects can be shared. You use ANYDATA queues to stage LCRs, and propagations to propagate these LCRs to the appropriate databases. The following sections describe staging and propagation in a Streams replication environment:

- LCR Staging
- LCR Propagation

See Also: *Oracle Streams Concepts and Administration* for more information about staging and propagation in Streams

LCR Staging

Captured LCRs are staged in a staging area. In Streams, the staging area is an ANYDATA queue that can store row LCRs and DDL LCRs, as well as other types of messages. Captured LCRs are staged in a buffered queue, which is System Global Area (SGA) memory associated with an ANYDATA queue.

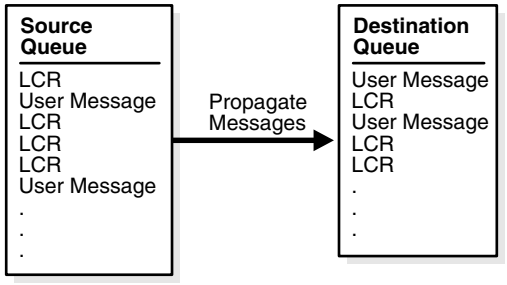
Staged LCRs can be propagated by a propagation or applied by an apply process, and a particular staged LCR can be both propagated and applied. A running propagation automatically propagates LCRs based on the rules in its rule sets, and a running apply process automatically applies LCRs based on the rules in its rule sets.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about buffered queues

LCR Propagation

In a Streams replication environment, a propagation typically propagates LCRs from a queue in the local database to a queue in a remote database. The queue from which the LCRs are propagated is called the **source queue**, and the queue that receives the LCRs is called the **destination queue**. There can be a one-to-many, many-to-one, or many-to-many relationship between source and destination queues.

Figure 1–5 Propagation from a Source Queue to a Destination Queue



Even after an LCR is propagated by a propagation or applied by an apply process, it can remain in the source queue if you have also configured Streams to propagate the LCR to one or more other queues. Also, notice that an ANYDATA queue can store non-LCR user messages as well as LCRs. Typically, non-LCR user messages are used for messaging applications, not for replication.

You can configure a Streams replication environment to propagate LCRs through one or more intermediate databases before arriving at a destination database. Such a propagation environment is called a **directed network**. An LCR might or might not be processed by an apply process at an intermediate database. Rules determine which LCRs are propagated to each destination database, and you can specify the route that LCRs will traverse on their way to a destination database.

The advantage of using a directed network is that a source database does not need to have a physical network connection with the destination database. So, if you want LCRs to propagate from one database to another, but there is no direct network connection between the computers running these databases, then you can still propagate the LCRs without reconfiguring your network, as long as one or more intermediate databases connect the source database to the destination database. If you use directed networks, and an intermediate site goes down for an extended period of time or is removed, then you might need to reconfigure the network and the Streams environment.

See Also: *Oracle Streams Concepts and Administration* for more information about directed networks

Apply and Streams Replication

In a Streams replication environment, changes made to shared database objects are captured and propagated to destination databases where they are applied. You configure one or more apply processes at each destination database to apply these changes. The following sections describe the concepts related to change apply in a Streams replication environment:

- Overview of the Apply Process
- Apply Processing Options for LCRs
- Apply Processes and Dependencies
- Considerations for Applying DML Changes to Tables
- Considerations for Applying DDL Changes
- Instantiation SCN and Ignore SCN for an Apply Process
- The Oldest SCN for an Apply Process
- Low-Watermark and High-Watermark for an Apply Process
- Trigger Firing Property

See Also: *Oracle Streams Concepts and Administration* for more information about change apply with an apply process

Overview of the Apply Process

An apply process is an optional Oracle background process that dequeues logical change records (LCRs) and user messages from a specific queue and either applies each one directly or passes it as a parameter to a user-defined procedure. The LCRs dequeued by an apply process contain the results of DML changes or DDL changes that an apply process can apply to database objects in a destination database. A user-defined message dequeued by an apply process is of type `ANYDATA` and can contain any user message, including a user-constructed LCR.

LCRs are applied by an **apply user**. The apply user applies all row changes resulting from DML operations and all DDL changes. The apply user also runs custom rule-based transformations configured for apply process rules, and runs apply handlers configured for an apply process.

Apply Processing Options for LCRs

An apply process is a flexible mechanism for processing the LCRs in a queue. You have options to consider when you configure one or more apply processes for your environment. Typically, to accomplish replication in a Streams environment, an apply process applies LCRs, not non-LCR user messages. This section discusses the LCR processing options available to you with an apply process.

Captured and User-Enqueued LCRs

A single apply process can apply either captured LCRs or user-enqueued LCRs, but not both. If a queue at a destination database contains both captured and user-enqueued LCRs, then the destination database must have at least two apply processes to process the LCRs. You can use the `DBMS_STREAMS_ADM` package or the

DBMS_APPLY_ADM package to create an apply process that applies captured LCRs, but only the CREATE_APPLY procedure in the DBMS_APPLY_ADM package can create an apply process that applies user-enqueued LCRs.

See Also: "Creating an Apply Process that Applies LCRs" on page 9-10

Direct and Custom Apply of LCRs

Direct apply means that an apply process applies an LCR without running a user procedure. The apply process either successfully applies the change in the LCR to a database object or, if a conflict or an apply error is encountered, tries to resolve the error with a conflict handler or a user-specified procedure called an error handler.

If a conflict handler can resolve the conflict, then it either applies the LCR or it discards the change in the LCR. If an error handler can resolve the error, then it should apply the LCR, if appropriate. An error handler can resolve an error by modifying the LCR before applying it. If the error handler cannot resolve the error, then the apply process places the transaction, and all LCRs associated with the transaction, into the error queue.

Custom apply means that an apply process passes the LCR as a parameter to a user procedure for processing. The user procedure can process the LCR in a customized way.

A user procedure that processes row LCRs resulting from DML statements is called a **DML handler**, while a user procedure that processes DDL LCRs resulting from DDL statements is called a **DDL handler**. An apply process can have many DML handlers but only one DDL handler, which processes all DDL LCRs dequeued by the apply process.

For each table associated with an apply process, you can set a separate DML handler to process each of the following types of operations in row LCRs:

- INSERT
- UPDATE
- DELETE
- LOB_UPDATE

For example, the hr.employees table can have one DML handler to process INSERT operations and a different DML handler to process UPDATE operations.

A user procedure can be used for any customized processing of LCRs. For example, if you want to skip DELETE operations for the hr.employees table at a certain destination database, then you can specify a DML handler for DELETE operations on this table to accomplish this goal. Such a handler is not invoked for INSERT, UPDATE, or LOB_UPDATE operations on the table. Or, if you want to log DDL changes before applying them, then you can create a user procedure that processes DDL operations to accomplish this.

A DML handler should never commit and never roll back, except to a named savepoint that the user procedure has established. To execute DDL inside a DDL handler, invoke the EXECUTE member procedure for the LCR.

In addition to DML handlers and DDL handlers, you can specify a **precommit handler** for an apply process. A precommit handler is a PL/SQL procedure that takes the commit SCN from an internal commit directive in the queue used by the apply process. The precommit handler can process the commit information in any

customized way. For example, it can record the commit information for an apply process in an audit table.

Attention: Do not modify `LONG`, `LONG RAW`, or nonassembled LOB column data in an LCR with DML handlers, error handlers, or custom rule-based transformation functions. DML handlers and error handlers can modify LOB columns in row LCRs that have been constructed by LOB assembly.

See Also:

- "Managing a DML Handler" on page 9-12
- "Managing a DDL Handler" on page 9-16
- Chapter 11, "Managing Logical Change Records (LCRs)" for more information about managing row LCRs with `LONG`, `LONG RAW`, or LOB columns
- *Oracle Streams Concepts and Administration* for more information about message processing options with an apply process

Apply Processes and Dependencies

When apply process parallelism is set to 1, a single apply server applies transactions in the same order as the order in which they were executed on the source database, and dependencies are not an issue. For example, if transaction A committed before transaction B on the source database, then, on the destination database, all of the LCRs in transaction A are applied before any LCRs in transaction B.

However, when apply process parallelism is set to a value greater than 1, multiple apply servers apply LCRs simultaneously. The coordinator process of an apply process monitors all of the apply servers to ensure that LCRs are applied and transactions are committed in the correct order. If a transaction being handled by an apply server has a dependency on another transaction that has not been applied, then the apply server contacts the coordinator process and waits for instructions.

For example, consider these two transactions:

1. A row is inserted into a table, and the transaction commits.
2. The same row is updated to change certain column values, and the transaction commits.

In this case, transaction 2 is dependent on transaction 1, because the row cannot be updated until after it is inserted. Suppose these transactions are captured from the redo log at a source database, propagated to a destination database, and applied at the destination database. Apply server A handles the insert transaction, and apply server B handles the update transaction.

If apply server B is ready to apply the update transaction before apply server A has applied the insert transaction, then apply server B waits for instructions from the coordinator process. After apply server A has applied the insert transaction, the coordinator process instructs apply server B to apply the update transaction.

When the `commit_serialization` apply process parameter is set to `none`, the commit order of nondependent transactions can be different than the commit order on the source database. When apply process parallelism is set to a value greater than one, and the `commit_serialization` apply process parameter is set to `full`,

nondependent, individual LCRs can be executed in a different order than they were on the source database, but the commit order for transactions always matches the commit order on the source database.

If the names of shared database objects are the same at the source and destination databases, and if the objects are in the same schemas at these databases, then an apply process automatically detects dependencies between row LCRs, regardless of apply process parallelism, assuming constraints are defined for the database objects at the destination database. Information about these constraints is stored in the data dictionary at the destination database. When an apply process is applying a transaction that contains row LCRs that depend on row LCRs in another transaction, the apply process ensures that the row LCRs are applied in the correct order and that the transactions are committed in the correct order to maintain the dependencies. Apply processes detect dependencies for both captured and user-enqueued row LCRs.

When rule-based transformations are specified for rules used by an apply process, and apply handlers are configured for the apply process, LCRs are processed in the following order:

1. The apply process dequeues LCRs from its queue.
2. The apply process runs rule-based transformations on LCRs, when appropriate.
3. The apply process detects dependencies between LCRs.
4. The apply process passes LCRs to apply handlers, when appropriate.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about apply servers

Virtual Dependency Definitions

In some cases, an apply process requires additional information to detect dependencies in row LCRs that are being applied in parallel. The following are examples of cases in which an apply process requires additional information to detect dependencies:

- The data dictionary at the destination database does not contain the required information. The following are examples of this case:
  - The apply process cannot find information about a database object in the destination databases's data dictionary. This can happen when there are data dictionary differences for shared database objects between the source and destination databases. For example, a shared database object can have a different name or can be in a different schema at the source database and destination database.
  - A relationship exists between two or more tables, and the relationship is not recorded in the destination databases's data dictionary. This can happen when database constraints are not defined to improve performance or when an application enforces dependencies during database operations instead of database constraints.
- Data is denormalized by an apply handler after dependency computation. For example, the information in a single row LCR can be used to create multiple row LCRs that are applied to multiple tables.

Apply errors or incorrect processing can result when an apply process cannot determine dependencies properly. In some of the cases described in the previous list, rule-based transformations can be used to avoid apply problems. For example, if a shared database object is in different schemas at the source and destination databases,

then a rule-based transformation can change the schema in the appropriate LCRs. However, the disadvantage with using rule-based transformations is that they cannot be executed in parallel.

A **virtual dependency definition** is a description of a dependency that is used by an apply process to detect dependencies between transactions at a destination database. A virtual dependency definition is not described as a constraint in the destination database's data dictionary. Instead, it is specified using procedures in the `DBMS_APPLY_ADM` package. Virtual dependency definitions enable an apply process to detect dependencies that it would not be able to detect by using only the constraint information in the data dictionary. After dependencies are detected, an apply process schedules LCRs and transactions in the correct order for apply.

Virtual dependency definitions provide required information so that apply processes can detect dependencies correctly before applying LCRs directly or passing LCRs to apply handlers. Virtual dependency definitions enable apply handlers to process these LCRs correctly, and the apply handlers can process them in parallel to improve performance.

A virtual dependency definition can define one of the following types of dependencies:

- Value Dependency
- Object Dependency

Note: A destination database must be running Oracle Database 10*g* Release 2 or later to specify virtual dependency definitions.

See Also:

- "Using Virtual Dependency Definitions" on page 9-18
- "Monitoring Virtual Dependency Definitions" on page 12-9

Value Dependency A **value dependency** defines a table constraint, such as a unique key, or a relationship between the columns of two or more tables. A value dependency is set for one or more columns, and an apply process uses a value dependency to detect dependencies between row LCRs that contain values for these columns. Value dependencies can define virtual foreign key relationships between tables, but, unlike foreign key relationships, value dependencies can involve more than two tables.

Value dependencies are useful when relationships between columns in tables are not described by constraints in the destination database's data dictionary. Value dependencies describe these relationships, and an apply process uses the value dependencies to determine when two or more row LCRs in different transactions involve the same row in a table at the destination database. For transactions that are being applied in parallel, when two or more row LCRs involve the same row, the transactions that include these row LCRs are dependent transactions.

Use the `SET_VALUE_DEPENDENCY` procedure in the `DBMS_APPLY_ADM` package to define or remove a value dependency at a destination database. In this procedure, table columns are specified as attributes.

The following restrictions pertain to value dependencies:

- The row LCRs that involve the database objects specified in a value dependency must originate from a single source database.
- Each value dependency must contain only one set of attributes for a particular database object.

Also, any columns specified in a value dependency at a destination database must be supplementally logged at the source database. These columns must be unconditionally logged.

> **See Also:** "Supplemental Logging for Streams Replication" on page 1-8

Object Dependency An **object dependency** defines a parent-child relationship between two objects at a destination database. An apply process schedules execution of transactions that involve the child object after all transactions with lower commit system change number (CSCN) values that involve the parent object have been committed. An apply process uses the object identifier in each row LCR to detect dependencies. The apply process does not use column values in the row LCRs to detect object dependencies.

Object dependencies are useful when relationships between tables are not described by constraints in the destination database's data dictionary. Object dependencies describe these relationships, and an apply process uses the object dependencies to determine when two or more row LCRs in different transactions involve these tables. For transactions that are being applied in parallel, when a row LCR in one transaction involves the child table, and a row LCR in a different transaction involves the parent table, the transactions that include these row LCRs are dependent transactions.

Use the `CREATE_OBJECT_DEPENDENCY` procedure to create an object dependency at a destination database. Use the `DROP_OBJECT_DEPENDENCY` procedure to drop an object dependency at a destination database. Both of these procedures are in the in the `DBMS_APPLY_ADM` package.

Note: Tables with circular dependencies can result in apply process deadlocks when apply process parallelism is greater than 1. The following is an example of a circular dependency: Table A has a foreign key constraint on table B, and table B has a foreign key constraint on table A. Apply process deadlocks are possible when two or more transactions that involve the tables with circular dependencies commit at the same SCN.

How Dependent Transactions Are Applied

When an apply process is applying transactions in parallel, and it detects dependent transactions, it applies the row LCRs in the transaction with the lower CSCN and commits this transaction before it applies the row LCRs in the transaction with the higher CSCN.

For example, consider two transactions: transaction A and transaction B. The transactions are dependent transactions, and each transaction contains 100 row LCRs. Transaction A committed on the source database before transaction B. Therefore, transaction A has the lower CSCN of the two transactions. An apply process can apply these transactions in parallel in the following way:

1. The apply process begins to apply row LCRs from both transactions in parallel.
2. Using a constraint in the destination database's data dictionary or a virtual dependency definition at the destination database, the apply process detects a dependency between a row LCR in transaction A and a row LCR in transaction B.
3. Because transaction B has the higher CSCN of the two transactions, the apply process waits to apply transaction B and does not apply the dependent row LCR in transaction B. The row LCRs before the dependent row LCR in transaction B have been applied. For example, if the dependent row LCR in transaction B is the 81st row LCR, then the apply process could have applied 80 of the 100 row LCRs in transaction B.
4. Because transaction A has the lower CSCN of the two transactions, the apply process applies the dependent row LCR in transaction A and the remaining row LCRs in transaction A. When all of the row LCRs in transactions A are applied, the apply process commits transaction A.
5. The apply process applies the dependent row LCR in transaction B and the remaining row LCRs in transaction B. When all of the row LCRs in transaction B are applied, the apply process commits transaction B.

Barrier Transactions

When an apply process cannot identify the table row or the database object specified in a row LCR by using the destination database's data dictionary and virtual dependency definitions, the transaction that contains the row LCR is applied after all of the other transactions with lower CSCN values. Such a transaction is called a **barrier transaction**. Transactions with higher CSCN values than the barrier transaction are not applied until after the barrier transaction has committed. In addition, all DDL transactions are barrier transactions.

Considerations for Applying DML Changes to Tables

The following sections discuss considerations for applying DML changes to tables:

- Constraints and Applying DML Changes to Tables
- Substitute Key Columns
- Apply Process Behavior for Column Discrepancies
- Index-Organized Tables and an Apply Process
- Conflict Resolution and an Apply Process
- Handlers and Row LCR Processing

Constraints and Applying DML Changes to Tables

You must ensure that the primary key columns at the destination database are logged in the redo log at the source database for every update. A unique key or foreign key constraint at a destination database that contains data from more that one column at the source database requires additional logging at the source database.

There are various ways to ensure that a column is logged at the source database. For example, whenever the value of a column is updated, the column is logged. Also, Oracle has a feature called supplemental logging that automates the logging of specified columns.

For a unique key and foreign key constraint at a destination database that contains data from only one column at a source database, no supplemental logging is required.

However, for a constraint that contains data from multiple columns at the source database, you must create a conditional supplemental log group containing all the columns at the source database that are used by the constraint at the destination database.

Typically, unique key and foreign key constraints include the same columns at the source database and destination database. However, in some cases, an apply handler or custom rule-based transformation can combine a multi-column constraint from the source database into a single key column at the destination database. Also, an apply handler or custom rule-based transformation can separate a single key column from the source database into a multi-column constraint at the destination database. In such cases, the number of columns in the constraint at the source database determines whether a conditional supplemental log group is required. If there is more than one column in the constraint at the source database, then a conditional supplemental log group containing all the constraint columns is required at the source database. If there is only one column in the constraint at the source database, then no supplemental logging is required for the key column.

> **See Also:** "Supplemental Logging for Streams Replication" on page 1-8

Substitute Key Columns

If possible, each table for which changes are applied by an apply process should have a primary key. When a primary key is not possible, Oracle recommends that each table have a set of columns that can be used as a unique identifier for each row of the table. If the tables that you plan to use in your Streams environment do not have a primary key or a set of unique columns, then consider altering these tables accordingly.

To detect conflicts and handle errors accurately, Oracle must be able to identify uniquely and match corresponding rows at different databases. By default, Streams uses the primary key of a table to identify rows in the table, and if a primary key does not exist, Streams uses the smallest unique key that has at least one `NOT NULL` column to identify rows in the table. When a table at a destination database does not have a primary key or a unique key with at least one `NOT NULL` column, or when you want to use columns other than the primary key or unique key for the key, you can designate a substitute key at the destination database. A substitute key is a column or set of columns that Oracle can use to identify rows in the table during apply.

You can specify the substitute primary key for a table using the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package. Unlike true primary keys, the substitute key columns can contain nulls. Also, the substitute key columns take precedence over any existing primary key or unique keys for the specified table for all apply processes at the destination database.

If you specify a substitute key for a table in a destination database, and these columns are not a primary key for the same table at the source database, then you must create an unconditional supplemental log group containing the substitute key columns at the source database.

In the absence of substitute key columns, primary key constraints, and unique key constraints, an apply process uses all of the columns in the table as the key columns, excluding LOB, `LONG`, and `LONG RAW` columns. In this case, you must create an unconditional supplemental log group containing these columns at the source database. Using substitute key columns is preferable when there is no primary key constraint for a table because fewer columns are needed in the row LCR.

Note:

- Oracle recommends that each column you specify as a substitute key column be a `NOT NULL` column. You should also create a single index that includes all of the columns in a substitute key. Following these guidelines improves performance for changes because the database can locate the relevant row more efficiently.
- LOB, `LONG`, `LONG RAW`, and user-defined type columns cannot be specified as substitute key columns.

See Also:

- The `DBMS_APPLY_ADM.SET_KEY_COLUMNS` procedure in the *Oracle Database PL/SQL Packages and Types Reference*
- "Supplemental Logging for Streams Replication" on page 1-8
- "Managing the Substitute Key Columns for a Table" on page 9-11

Apply Process Behavior for Column Discrepancies

A column discrepancy is any difference in the columns in a table at a source database and the columns in the same table at a destination database. If there are column discrepancies in your Streams environment, then use rule-based transformations or DML handlers to make the columns in row LCRs being applied by an apply process match the columns in the relevant tables at a destination database. The following sections describe apply process behavior for common column discrepancies.

See Also:

- *Oracle Streams Concepts and Administration* for more information about apply process handlers and rule-based transformations
- *Oracle Database PL/SQL Packages and Types Reference* for more information about LCRs

Missing Columns at the Destination Database If the table at the destination database is missing one or more columns that are in the table at the source database, then an apply process raises an error and moves the transaction that caused the error into the error queue. You can avoid such an error by creating a rule-based transformation or DML handler that deletes the missing columns from the LCRs before they are applied. Specifically, the transformation or handler can remove the extra columns using the `DELETE_COLUMN` member procedure on the row LCR.

Extra Columns at the Destination Database If the table at the destination database has more columns than the table at the source database, then apply process behavior depends on whether the extra columns are required for dependency computations. If the extra columns are not used for dependency computations, then an apply process applies changes to the destination table. In this case, if column defaults exist for the extra columns at the destination database, then these defaults are used for these columns for all inserts. Otherwise, these inserted columns are `NULL`.

If, however, the extra columns are used for dependency computations, then an apply process places the transactions that include these changes in the error queue. The following types of columns are required for dependency computations:

- For all changes, all key columns
- For `INSERT` and `DELETE` statements, all columns involved with constraints
- For `UPDATE` statements, if a constraint column is changed, such as a unique key constraint column or a foreign key constraint column, then all columns involved in the constraint

Column Datatype Mismatch If the datatype for a column in a table at the destination database does not match the datatype for the same column at the source database, then an apply process places transactions containing the changes to the mismatched column into the error queue. To avoid such an error, you can create a custom rule-based transformation or DML handler that converts the datatype.

Index-Organized Tables and an Apply Process

An apply process can apply changes made to an index-organized table only if the index-organized table does not contain any columns of the following datatypes:

- `ROWID`
- `UROWID`
- User-defined types (including object types, `REF`s, varrays, and nested tables)

If an index-organized table contains a column of one of these datatypes, then an apply process raises an error if it tries to apply LCRs that contain changes to it.

> **See Also:** *Oracle Streams Concepts and Administration* for information about the datatypes supported by an apply process

Conflict Resolution and an Apply Process

Conflicts are possible in a Streams configuration where data is shared between multiple databases. A **conflict** is a mismatch between the old values in an LCR and the expected data in a table. A conflict can occur if DML changes are allowed to a table for which changes are captured and to a table where these changes are applied.

For example, a transaction at the source database can update a row at nearly the same time as a different transaction that updates the same row at a destination database. In this case, if data consistency between the two databases is important, then when the change is propagated to the destination database, an apply process must be instructed either to keep the change at the destination database or replace it with the change from the source database. When data conflicts occur, you need a mechanism to ensure that the conflict is resolved in accordance with your business rules.

Streams automatically detects conflicts and, for update conflicts, tries to use an update conflict handler to resolve them if one is configured. Streams offers a variety of prebuilt handlers that enable you to define a conflict resolution system for your database that resolves conflicts in accordance with your business rules. If you have a unique situation that a prebuilt conflict resolution handler cannot resolve, then you can build and use your own custom conflict resolution handlers in an error handler or DML handler. Conflict detection can be disabled for nonkey columns.

> **See Also:** Chapter 3, "Streams Conflict Resolution"

Handlers and Row LCR Processing

Any of the following handlers can process a row LCR:

- DML handler
- Error handler
- Update conflict handler

The following sections describe the possible scenarios involving these handlers:

- No Relevant Handlers
- Relevant Update Conflict Handler
- DML Handler But No Relevant Update Conflict Handler
- DML Handler And a Relevant Update Conflict Handler
- Error Handler But No Relevant Update Conflict Handler
- Error Handler And a Relevant Update Conflict Handler

You cannot have a DML handler and an error handler simultaneously for the same operation on the same table. Therefore, there is no scenario in which they could both be invoked.

No Relevant Handlers If there are no relevant handlers for a row LCR, then an apply process tries to apply the change specified in the row LCR directly. If the apply process can apply the row LCR, then the change is made to the row in the table. If there is a conflict or an error during apply, then the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

Relevant Update Conflict Handler Consider a case where there is a relevant update conflict handler configured, but no other relevant handlers are configured. An apply process tries to apply the change specified in a row LCR directly. If the apply process can apply the row LCR, then the change is made to the row in the table.

If there is an error during apply that is caused by a condition other than an update conflict, including a uniqueness conflict or a delete conflict, then the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

If there is an update conflict during apply, then the relevant update conflict handler is invoked. If the update conflict handler resolves the conflict successfully, then the apply process either applies the LCR or discards the LCR, depending on the resolution of the update conflict, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets. If the update conflict handler cannot resolve the conflict, then the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

DML Handler But No Relevant Update Conflict Handler Consider a case where an apply process passes a row LCR to a DML handler, and there is no relevant update conflict handler configured.

The DML handler processes the row LCR. The designer of the DML handler has complete control over this processing. Some DML handlers can perform SQL operations or run the `EXECUTE` member procedure of the row LCR. If the DML handler runs the `EXECUTE` member procedure of the row LCR, then the apply process

tries to apply the row LCR. This row LCR might have been modified by the DML handler.

If any SQL operation performed by the DML handler fails, or if an attempt to run the `EXECUTE` member procedure fails, then the DML handler can try to handle the exception. If the DML handler does not raise an exception, then the apply process assumes the DML handler has performed the appropriate action with the row LCR, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets.

If the DML handler cannot handle the exception, then the DML handler should raise an exception. In this case, the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

DML Handler And a Relevant Update Conflict Handler Consider a case where an apply process passes a row LCR to a DML handler and there is a relevant update conflict handler configured.

The DML handler processes the row LCR. The designer of the DML handler has complete control over this processing. Some DML handlers might perform SQL operations or run the `EXECUTE` member procedure of the row LCR. If the DML handler runs the `EXECUTE` member procedure of the row LCR, then the apply process tries to apply the row LCR. This row LCR could have been modified by the DML handler.

If any SQL operation performed by the DML handler fails, or if an attempt to run the `EXECUTE` member procedure fails for any reason other than an update conflict, then the behavior is the same as that described in "DML Handler But No Relevant Update Conflict Handler" on page 1-22. Note that uniqueness conflicts and delete conflicts are not update conflicts.

If an attempt to run the `EXECUTE` member procedure fails because of an update conflict, then the behavior depends on the setting of the `conflict_resolution` parameter in the `EXECUTE` member procedure:

The conflict_resolution Parameter Is Set to true

If the `conflict_resolution` parameter is set to `true`, then the relevant update conflict handler is invoked. If the update conflict handler resolves the conflict successfully, and all other operations performed by the DML handler succeed, then the DML handler finishes without raising an exception, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets.

If the update conflict handler cannot resolve the conflict, then the DML handler can try to handle the exception. If the DML handler does not raise an exception, then the apply process assumes the DML handler has performed the appropriate action with the row LCR, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets. If the DML handler cannot handle the exception, then the DML handler should raise an exception. In this case, the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

The conflict_resolution Parameter Is Set to false

If the `conflict_resolution` parameter is set to `false`, then the relevant update conflict handler is not invoked. In this case, the behavior is the same as that described in "DML Handler But No Relevant Update Conflict Handler" on page 1-22.

Error Handler But No Relevant Update Conflict Handler Consider a case where an apply process encounters an error when it tries to apply a row LCR. This error can be caused by a conflict or by some other condition. There is an error handler for the table operation but no relevant update conflict handler configured.

The row LCR is passed to the error handler. The error handler processes the row LCR. The designer of the error handler has complete control over this processing. Some error handlers might perform SQL operations or run the `EXECUTE` member procedure of the row LCR. If the error handler runs the `EXECUTE` member procedure of the row LCR, then the apply process tries to apply the row LCR. This row LCR could have been modified by the error handler.

If any SQL operation performed by the error handler fails, or if an attempt to run the `EXECUTE` member procedure fails, then the error handler can try to handle the exception. If the error handler does not raise an exception, then the apply process assumes the error handler has performed the appropriate action with the row LCR, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets.

If the error handler cannot handle the exception, then the error handler should raise an exception. In this case, the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

Error Handler And a Relevant Update Conflict Handler Consider a case where an apply process encounters an error when it tries to apply a row LCR. There is an error handler for the table operation, and there is a relevant update conflict handler configured.

The handler that is invoked to handle the error depends on the type of error it is:

- If the error is caused by a condition other than an update conflict, including a uniqueness conflict or a delete conflict, then the error handler is invoked, and the behavior is the same as that described in "Error Handler But No Relevant Update Conflict Handler" on page 1-24.
- If the error is caused by an update conflict, then the update conflict handler is invoked. If the update conflict handler resolves the conflict successfully, then the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets. In this case, the error handler is not invoked.

  If the update conflict handler cannot resolve the conflict, then the error handler is invoked. If the error handler does not raise an exception, then the apply process assumes the error handler has performed the appropriate action with the row LCR, and the apply process continues applying the other LCRs in the transaction that should be applied according to the apply process rule sets. If the error handler cannot process the LCR, then the error handler should raise an exception. In this case, the transaction containing the row LCR is rolled back, and all of the LCRs in the transaction that should be applied according to the apply process rule sets are moved to the error queue.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `EXECUTE` member procedure for row LCRs
- *Oracle Streams Concepts and Administration* for more information about managing apply handlers and for more information about how rules are used in Streams
- "Managing Streams Conflict Detection and Resolution" on page 9-22

Considerations for Applying DDL Changes

The following sections discuss considerations for applying DDL changes to tables:

- Types of DDL Changes Ignored by an Apply Process
- Database Structures in a Streams Environment
- Current Schema User Must Exist at Destination Database
- System-Generated Names
- CREATE TABLE AS SELECT Statements

Types of DDL Changes Ignored by an Apply Process

The following types of DDL changes are not supported by an apply process. These types of DDL changes are not applied:

- `ALTER MATERIALIZED VIEW`
- `ALTER MATERIALIZED VIEW LOG`
- `CREATE DATABASE LINK`
- `CREATE SCHEMA AUTHORIZATION`
- `CREATE MATERIALIZED VIEW`
- `CREATE MATERIALIZED VIEW LOG`
- `DROP DATABASE LINK`
- `DROP MATERIALIZED VIEW`
- `DROP MATERIALIZED VIEW LOG`
- `RENAME`

If an apply process receives a DDL LCR that specifies an operation that cannot be applied, then the apply process ignores the DDL LCR and records the following message in the apply process trace file, followed by the DDL text that was ignored:

```
Apply process ignored the following DDL:
```

An apply process applies all other types of DDL changes if the DDL LCRs containing the changes should be applied according to the apply process rule sets. Also, an apply process can apply valid, user-enqueued DDL LCRs.

Note:

- An apply process applies `ALTER` *`object_type object_name`* `RENAME` changes, such as `ALTER TABLE jobs RENAME`. Therefore, if you want DDL changes that rename objects to be applied, then use `ALTER` *`object_type object_name`* `RENAME` statements instead of `RENAME` statements. After changing the name of a database object, new rules that specify the new database object name might be needed to replicate changes to the database object.
- The name "materialized view" is synonymous with the name "snapshot". Snapshot equivalents of the statements on materialized views are ignored by an apply process.

See Also: *Oracle Streams Concepts and Administration* for more information about how rules are used in Streams

Database Structures in a Streams Environment

For captured DDL changes to be applied properly at a destination database, either the destination database must have the same database structures as the source database, or the nonidentical database structural information must not be specified in the DDL statement. Database structures include data files, tablespaces, rollback segments, and other physical and logical structures that support database objects.

For example, for captured DDL changes to tables to be applied properly at a destination database, the following conditions must be met:

- The same storage parameters must be specified in the `CREATE TABLE` statement at the source database and destination database.
- If a DDL statement refers to specific tablespaces or rollback segments, then the tablespaces or rollback segments must have the same names and compatible specifications at the source database and destination database.

  However, if the tablespaces and rollback segments are not specified in the DDL statement, then the default tablespaces and rollback segments are used. In this case, the tablespaces and rollback segments can differ at the source database and destination database.
- The same partitioning specifications must be used at the source database and destination database.

Current Schema User Must Exist at Destination Database

For a DDL LCR to be applied at a destination database successfully, the user specified as the `current_schema` in the DDL LCR must exist at the destination database. The current schema is the schema that is used if no schema is specified for an object in the DDL text.

See Also:

- *Oracle Database Concepts* for more information about database structures
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `current_schema` attribute in DDL LCRs

System-Generated Names

If you plan to capture DDL changes at a source database and apply these DDL changes at a destination database, then avoid using system-generated names. If a DDL statement results in a system-generated name for an object, then the name of the object typically will be different at the source database and each destination database applying the DDL change from this source database. Different names for objects can result in apply errors for future DDL changes.

For example, suppose the following DDL statement is run at a source database:

```
CREATE TABLE sys_gen_name (n1 NUMBER  NOT NULL);
```

This statement results in a `NOT NULL` constraint with a system-generated name. For example, the `NOT NULL` constraint might be named `sys_001500`. When this change is applied at a destination database, the system-generated name for this constraint might be `sys_c1000`.

Suppose the following DDL statement is run at the source database:

```
ALTER TABLE sys_gen_name DROP CONSTRAINT sys_001500;
```

This DDL statement succeeds at the source database, but it fails at the destination database and results in an apply error.

To avoid such an error, explicitly name all objects resulting from DDL statements. For example, to name a `NOT NULL` constraint explicitly, run the following DDL statement:

```
CREATE TABLE sys_gen_name (n1 NUMBER CONSTRAINT sys_gen_name_nn NOT NULL);
```

CREATE TABLE AS SELECT Statements

When applying a change resulting from a `CREATE TABLE AS SELECT` statement, an apply process performs two steps:

1. The `CREATE TABLE AS SELECT` statement is executed at the destination database, but it creates only the structure of the table. It does not insert any rows into the table. If the `CREATE TABLE AS SELECT` statement fails, then an apply process error results. Otherwise, the statement auto commits, and the apply process performs Step 2.
2. The apply process inserts the rows that were inserted at the source database as a result of the `CREATE TABLE AS SELECT` statement into the corresponding table at the destination database. It is possible that a capture process, a propagation, or an apply process will discard all of the row LCRs with these inserts based on their rule sets. In this case, the table remains empty at the destination database.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about how rules are used in Streams

Instantiation SCN and Ignore SCN for an Apply Process

In a Streams environment that shares information within a single database or between multiple databases, a source database is the database where changes are generated in the redo log. Suppose an environment has the following characteristics:

- A capture process captures changes to tables at the source database and stages the changes as LCRs in a queue.
- An apply process applies these LCRs, either at the same database or at a destination database to which the LCRs have been propagated.

In such an environment, for the each table, only changes that committed after a specific system change number (SCN) at the source database are applied. An **instantiation SCN** specifies this value for each table.

An instantiation SCN can be set during instantiation, or an instantiation SCN can be set using a procedure in the DBMS_APPLY_ADM package. If the tables do not exist at the destination database before the Streams replication environment is configured, then these table are physically created (instantiated) using copies from the source database, and the instantiation SCN is set for each table during instantiation. If the tables already exist at the destination database before the Streams replication environment is configured, then these table are not instantiated using copies from the source database. Instead, the instantiation SCN must be set manually for each table using one of the following procedures in the DBMS_APPLY_ADM package: SET_TABLE_INSTANTIATION_SCN, SET_SCHEMA_INSTANATIATION_SCN, or SET_GLOBAL_INSTANTIATION_SCN.

The instantiation SCN for a database object controls which LCRs that contain changes to the database object are ignored by an apply process and which LCRs are applied by an apply process. If the commit SCN of an LCR for a database object from a source database is less than or equal to the instantiation SCN for that database object at a destination database, then the apply process at the destination database discards the LCR. Otherwise, the apply process applies the LCR.

Also, if there are multiple source databases for a shared database object at a destination database, then an instantiation SCN must be set for each source database, and the instantiation SCN can be different for each source database. You can set instantiation SCNs by using export/import or transportable tablespaces. You can also set an instantiation SCN by using a procedure in the DBMS_APPLY_ADM package.

Streams also records the **ignore SCN** for each database object. The ignore SCN is the SCN below which changes to the database object cannot be applied. The instantiation SCN for an object cannot be set lower than the ignore SCN for the object. This value corresponds to the SCN value at the source database at the time when the object was prepared for instantiation. An ignore SCN is set for a database object only when the database object is instantiated using Export/Import.

You can view the instantiation SCN and ignore SCN for database objects by querying the DBA_APPLY_INSTANTIATED_OBJECTS data dictionary view.

See Also:

- "Setting Instantiation SCNs at a Destination Database" on page 10-27
- "Instantiating Objects in a Streams Replication Environment" on page 10-3
- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

The Oldest SCN for an Apply Process

If an apply process is running, then the **oldest SCN** is the earliest SCN of the transactions currently being dequeued and applied. For a stopped apply process, the oldest SCN is the earliest SCN of the transactions that were being applied when the apply process was stopped.

The following are two common scenarios in which the oldest SCN is important:

- You must recover the database in which the apply process is running to a certain point in time.
- You stop using an existing capture process that captures changes for the apply process and use a different capture process to capture changes for the apply process.

In both cases, you should determine the oldest SCN for the apply process by querying the DBA_APPLY_PROGRESS data dictionary view. The OLDEST_MESSAGE_NUMBER column in this view contains the oldest SCN. Next, set the start SCN for the capture process that is capturing changes for the apply process to the same value as the oldest SCN value. If the capture process is capturing changes for other apply processes, then these other apply processes might receive duplicate LCRs when you reset the start SCN for the capture process. In this case, the other apply processes automatically discard the duplicate LCRs.

See Also:

- *Oracle Streams Concepts and Administration* for more information about SCN values relating to a capture process
- "Performing Database Point-in-Time Recovery in a Streams Environment" on page 9-30

Low-Watermark and High-Watermark for an Apply Process

The **low-watermark** for an apply process is the system change number (SCN) up to which all LCRs have been applied. That is, LCRs that were committed at an SCN less than or equal to the low-watermark number have definitely been applied, but some LCRs that were committed with a higher SCN also might have been applied. The low-watermark SCN for an apply process is equivalent to the **applied SCN** for a capture process.

The **high-watermark** for an apply process is the SCN beyond which no LCRs have been applied. That is, no LCRs that were committed with an SCN greater than the high-watermark have been applied.

You can view the low-watermark and high-watermark for one or more apply processes by querying the V$STREAMS_APPLY_COORDINATOR and ALL_APPLY_PROGRESS data dictionary views.

Trigger Firing Property

You can control a DML or DDL trigger's firing property using the SET_TRIGGER_FIRING_PROPERTY procedure in the DBMS_DDL package. This procedure lets you specify whether a trigger's firing property is set to fire once.

If a trigger's firing property is set to fire once, then it does not fire in the following cases:

- When a relevant change is made by an apply process
- When a relevant change results from the execution of one or more apply errors using the EXECUTE_ERROR or EXECUTE_ALL_ERRORS procedure in the DBMS_APPLY_ADM package

If a trigger is not set to fire once, then it fires in both of these cases.

By default, DML and DDL triggers are set to fire once. You can check a trigger's firing property by using the IS_TRIGGER_FIRE_ONCE function in the DBMS_DDL package.

For example, in the hr schema, the update_job_history trigger adds a row to the job_history table when data is updated in the job_id or department_id column in the employees table. Suppose, in a Streams environment, the following configuration exists:

- A capture process captures changes to both of these tables at the dbs1.net database.
- A propagation propagates these changes to the dbs2.net database.
- An apply process applies these changes at the dbs2.net database.
- The update_job_history trigger exists in the hr schema in both databases.

If the update_job_history trigger is not set to fire once at dbs2.net in this scenario, then these actions result:

1. The job_id column is updated for an employee in the employees table at dbs1.net.
2. The update_job_history trigger fires at dbs1.net and adds a row to the job_history table that records the change.
3. The capture process at dbs1.net captures the changes to both the employees table and the job_history table.
4. A propagation propagates these changes to the dbs2.net database.
5. An apply process at the dbs2.net database applies both changes.
6. The update_job_history trigger fires at dbs2.net when the apply process updates the employees table.

In this case, the change to the employees table is recorded twice at the dbs2.net database: when the apply process applies the change to the job_history table and when the update_job_history trigger fires to record the change made to the employees table by the apply process.

A database administrator might not want the update_job_history trigger to fire at the dbs2.net database when a change is made by the apply process. Similarly, a database administrator might not want a trigger to fire because of the execution of an apply error transaction. If the update_job_history trigger's firing property is set to fire once, then it does not fire at dbs2.net when the apply process applies a change to the employees table, and it does not fire when an executed error transaction updates the employees table.

Also, if you use the ON SCHEMA clause to create a schema trigger, then the schema trigger fires only if the schema performs a relevant change. Therefore, when an apply process is applying changes, a schema trigger that is set to fire always fires only if the apply user is the same as the schema specified in the schema trigger. If the schema trigger is set to fire once, then it never fires when an apply process applies changes, regardless of whether the apply user is the same as the schema specified in the schema trigger.

For example, if you specify a schema trigger that always fires on the hr schema at a source database and destination database, but the apply user at a destination database is strmadmin, then the trigger fires when the hr user performs a relevant change on the source database, but the trigger does not fire when this change is applied at the destination database. However, if you specify a schema trigger that always fires on the strmadmin schema at the destination database, then this trigger fires whenever a relevant change is made by the apply process, regardless of any trigger specifications at the source database.

Note: Only DML and DDL triggers can be set to fire once. All other types of triggers always fire.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about setting a trigger's firing property with the `SET_TRIGGER_FIRING_PROPERTY` procedure

2

Instantiation and Streams Replication

This chapter contains conceptual information about instantiation and Streams replication.

This chapter contains these topics:

- Overview of Instantiation and Streams Replication
- Capture Process Rules and Preparation for Instantiation
- Oracle Data Pump and Streams Instantiation
- Recovery Manager (RMAN) and Streams Instantiation
- Original Export/Import and Streams Instantiation

See Also: Chapter 10, "Performing Instantiations"

Overview of Instantiation and Streams Replication

In a Streams environment that shares a database object within a single database or between multiple databases, a source database is the database where changes to the object are generated in the redo log, and a destination database is the database where these changes are dequeued by an apply process. If a capture process captures, or will capture, such changes, and the changes will be applied locally or propagated to other databases and applied at destination databases, then you must **instantiate** these source database objects before you can replicate changes to the objects. If a database where changes to the source database objects will be applied is a different database than the source database, then the destination database must have a copy of these database objects.

In Streams, the following general steps instantiate a database object:

1. Prepare the object for instantiation at the source database.
2. If a copy of the object does not exist at the destination database, then create an object physically at the destination database based on an object at the source database. You can use export/import, transportable tablespaces, or RMAN to copy database objects for instantiation. If the database object already exists at the destination database, then this step is not necessary.
3. Set the instantiation SCN for the database object at the destination database. An instantiation SCN instructs an apply process at the destination database to apply only changes that committed at the source database after the specified SCN.

In some cases, Step 1 and Step 3 are completed automatically. For example, when you add rules for a database object to the positive rule set for a capture process by running a procedure in the `DBMS_STREAMS_ADM` package, the database object is prepared for

instantiation automatically. Also, when you use export/import, transportable tablespaces, or the RMAN `TRANSPORT TABLESPACE` command to copy database objects from a source database to a destination database, instantiation SCNs can be set for these database objects automatically.

Note:

- You can use either Data Pump export/import or original export/import for Streams instantiations. General references to export/import in this document refer to both Data Pump and original export/import. This document distinguishes between Data Pump and original export/import when necessary.
- The RMAN `DUPLICATE` command can be used to instantiate an entire database, but this command does not set instantiation SCNs for database objects.

If the database object being instantiated is a table, then the tables at the source and destination database do not need to be an exact match. However, if some or all of the table data is replicated between the two databases, then the data that is replicated should be consistent when the table is instantiated. Whenever you plan to replicate changes to a database object, you must always prepare the object for instantiation at the source database and set the instantiation SCN for the database object at the destination database. By preparing an object for instantiation, you are setting the lowest SCN for which changes to the object might need to be applied at destination databases. This SCN is called the ignore SCN. You should prepare a database object for instantiation after a capture process has been configured to capture changes to the database object.

When you instantiate tables using export/import, transportable tablespaces, or RMAN, any supplemental log group specifications are retained for the instantiated tables. That is, after instantiation, log group specifications for imported tables at the import database are the same as the log group specifications for these tables at the export database. If you do not want to retain supplemental log group specifications for tables at the import database, then you can drop specific supplemental log groups after import.

Database supplemental logging specifications are not retained during export/import, even if you perform a full database export/import. However, the RMAN `DUPLICATE` command retains database supplemental logging specifications at the instantiated database.

Attention:

- During an export for a Streams instantiation, make sure no DDL changes are made to objects being exported.
- When you export a database or schema that contains rules with non-`NULL` action contexts, the database or the default tablespace of the schema that owns the rules must be writeable. If the database or tablespace is read-only, then export errors result.

See Also:

- "Instantiating Objects in a Streams Replication Environment" on page 10-3
- "The Oldest SCN for an Apply Process" on page 1-28
- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for information about adding and dropping supplemental log groups

Capture Process Rules and Preparation for Instantiation

The following procedures in the DBMS_CAPTURE_ADM package prepare database objects for instantiation:

- PREPARE_TABLE_INSTANTIATION prepares a single table for instantiation.
- PREPARE_SCHEMA_INSTANTIATION prepares for instantiation all of the database objects in a schema and all database objects added to the schema in the future.
- PREPARE_GLOBAL_INSTANTIATION prepares for instantiation all of the database objects in a database and all database objects added to the database in the future.

These procedures record the lowest SCN of each object for instantiation. SCNs subsequent to the lowest SCN for an object can be used for instantiating the object. These procedures also populate the Streams data dictionary for the relevant capture processes, propagations, and apply processes that capture, propagate, or apply changes made to the table, schema, or database being prepared for instantiation. In addition, these procedures optionally can enable supplemental logging for key columns or all columns in the tables that are being prepared for instantiation.

See Also: "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

DBMS_STREAMS_ADM Package Procedures Automatically Prepare Objects

When you add rules to the positive rule set for a capture process by running a procedure in the DBMS_STREAMS_ADM package, a procedure in the DBMS_CAPTURE_ADM package is run automatically on the database objects whose changes will be captured by the capture process. The following table lists which procedure is run in the DBMS_CAPTURE_ADM package when you run a procedure in the DBMS_STREAMS_ADM package.

When you run this procedure in the DBMS_STREAMS_ADM package	This procedure in the DBMS_CAPTURE_ADM package is run automatically
ADD_TABLE_RULES ADD_SUBSET_RULES	PREPARE_TABLE_INSTANTIATION
ADD_SCHEMA_RULES	PREPARE_SCHEMA_INSTANTIATION
ADD_GLOBAL_RULES	PREPARE_GLOBAL_INSTANTIATION

More than one call to prepare for instantiation is allowed. If you are using downstream capture, and the downstream capture process uses a database link from the downstream database to the source database, then the objects are prepared for instantiation automatically when you run one of these procedures in the DBMS_STREAMS_ADM package. However, if the downstream capture process does not use a

database link from the downstream database to the source database, then you must prepare the objects for instantiation manually.

When capture process rules are created by the DBMS_RULE_ADM package instead of the DBMS_STREAMS_ADM package, you must run the appropriate procedure manually to prepare each table, schema, or database whose changes will be captured for instantiation, if you plan to apply changes that result from the capture process rules with an apply process.

When Preparing for Instantiation Is Required

Whenever you add, or modify the condition of, a capture process, propagation, or apply process rule for a database object that is in a positive rule set, you must run the appropriate procedure to prepare the database object for instantiation at the source database if any of the following conditions are met:

- One or more rules are added to the positive rule set for a capture process that instruct the capture process to capture changes made to the object.
- One or more conditions of rules in the positive rule set for a capture process are modified to instruct the capture process to capture changes made to the object.
- One or more rules are added to the positive rule set for a propagation that instruct the propagation to propagate changes made to the object.
- One or more conditions of rules in the positive rule set for a propagation are modified to instruct the propagation to propagate changes made to the object.
- One or more rules are added to the positive rule set for an apply process that instruct the apply process to apply changes made to the object at the source database.
- One or more conditions of rules in the positive rule set for an apply process are modified to instruct the apply process to apply changes made to the object at the source database.

Whenever you remove, or modify the condition of, a capture process, propagation, or apply process rule for a database object that is in a negative rule set, you must run the appropriate procedure to prepare the database object for instantiation at the source database if any of the following conditions are met:

- One or more rules are removed from the negative rule set for a capture process to instruct the capture process to capture changes made to the object.
- One or more conditions of rules in the negative rule set for a capture process are modified to instruct the capture process to capture changes made to the object.
- One or more rules are removed from the negative rule set for a propagation to instruct the propagation to propagate changes made to the object.
- One or more conditions of rules in the negative rule set for a propagation are modified to instruct the propagation to propagate changes made to the object.
- One or more rules are removed from the negative rule set for an apply process to instruct the apply process to apply changes made to the object at the source database.
- One or more conditions of rules in the negative rule set for an apply process are modified to instruct the apply process to apply changes made to the object at the source database.

When any of these conditions are met for changes to a positive or negative rule set, you must prepare the relevant database objects for instantiation at the source database

to populate any relevant Streams data dictionary that requires information about the source object, even if the object already exists at a remote database where the rules were added or changed.

The relevant Streams data dictionaries are populated asynchronously for both the local dictionary and all remote dictionaries. The procedure that prepares for instantiation adds information to the redo log at the source database. The local Streams data dictionary is populated with the information about the object when a capture process captures these redo entries, and any remote Streams data dictionaries are populated when the information is propagated to them.

See Also:

- "Capture Process Overview" on page 1-5 for more information about local and downstream capture
- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

Supplemental Logging Options During Preparation for Instantiation

Supplemental logging places additional column data into a redo log whenever an operation is performed. The procedures in the `DBMS_CAPTURE_ADM` package that prepare database objects for instantiation are `PREPARE_TABLE_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, and `PREPARE_GLOBAL_INSTANTIATION`. These procedures include a `supplemental_logging` parameter which controls the supplemental logging specifications for the database objects being prepared for instantiation.

Table 2–1 describes the values for the `supplemental_logging` parameter for each procedure.

Table 2–1 Supplemental Logging Options During Preparation for Instantiation

Procedure	supplemental_logging Parameter Setting	Description
`PREPARE_TABLE_INSTANTIATION`	`keys`	The procedure enables supplemental logging for primary key, unique key, bitmap index, and foreign key columns in the table being prepared for instantiation. The procedure places the logged columns for the table in three separate log groups: the primary key columns in an unconditional log group, the unique key columns and bitmap index columns in a conditional log group, and the foreign key columns in a conditional log group.
`PREPARE_TABLE_INSTANTIATION`	`all`	The procedure enables supplemental logging for all columns in the table being prepared for instantiation. The procedure places all of the columns for the table in an unconditional log group.
`PREPARE_SCHEMA_INSTANTIATION`	`keys`	The procedure enables supplemental logging for primary key, unique key, bitmap index, and foreign key columns in the tables in the schema being prepared for instantiation and for any table added to this schema in the future. Primary key columns are logged unconditionally. Unique key, bitmap index, and foreign key columns are logged conditionally.

Table 2–1 (Cont.) Supplemental Logging Options During Preparation for Instantiation

Procedure	supplemental_logging Parameter Setting	Description
PREPARE_SCHEMA_INSTANTIATION	all	The procedure enables supplemental logging for all columns in the tables in the schema being prepared for instantiation and for any table added to this schema in the future. The columns are logged unconditionally.
PREPARE_GLOBAL_INSTANTIATION	keys	The procedure enables database supplemental logging for primary key, unique key, bitmap index, and foreign key columns in the tables in the database being prepared for instantiation and for any table added to the database in the future. Primary key columns are logged unconditionally. Unique key, bitmap index, and foreign key columns are logged conditionally.
PREPARE_GLOBAL_INSTANTIATION	all	The procedure enables supplemental logging for all columns in all of the tables in the database being prepared for instantiation and for any table added to the database in the future. The columns are logged unconditionally.
Any Prepare Procedure	none	The procedure does not enable supplemental logging for any columns in the tables being prepared for instantiation.

If the supplemental_logging parameter is not specified when one of prepare procedures is run, then keys is the default. Some of the procedures in the DBMS_STREAMS_ADM package prepare tables for instantiation when they add rules to a positive capture process rule set. In this case, the default supplemental logging option, keys, is specified for the tables being prepared for instantiation.

Note:

- When all is specified for the supplemental_logging parameter, supplemental logging is not enabled for columns of the following types: LOB, LONG, LONG RAW, and user-defined type.
- Specifying keys for the supplemental_logging parameter does not enable supplemental logging of bitmap join index columns.
- Oracle Database 10*g* Release 2 introduces the supplemental_logging parameter for the prepare procedures. By default, running these procedures enables supplemental logging. Prior to this release, these procedures did not enable supplemental logging. If you remove a Streams environment, or if you remove certain database objects from a Streams environment, then you can also remove the supplemental logging enabled by these procedures to avoid unnecessary logging.

See Also:

- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1
- "Supplemental Logging for Streams Replication" on page 1-8
- "DBMS_STREAMS_ADM Package Procedures Automatically Prepare Objects" on page 2-3
- "Aborting Preparation for Instantiation at a Source Database" on page 10-3 for information about removing supplemental logging enabled by the prepare procedures

Oracle Data Pump and Streams Instantiation

The following sections contain information about Streams instantiations that use Oracle Data Pump.

See Also:

- "Instantiating Objects Using Data Pump Export/Import" on page 10-4
- *Oracle Streams Concepts and Administration* for information about performing a full database export/import on a database using Streams
- *Oracle Database Utilities* for more information about Data Pump

Data Pump Export and Object Consistency

During export, Oracle Data Pump automatically uses the Oracle Flashback feature to ensure that the exported data and the exported procedural actions for each database object are consistent to a single point in time. When you perform an instantiation in a Streams environment, some degree of consistency is required. Using the Data Pump Export utility is sufficient to ensure this consistency for Streams instantiations.

If you are using an export dump file for other purposes in addition to a Streams instantiation, and these other purposes have more stringent consistency requirements than those provided by Data Pump's default export, then you can use the Data Pump Export utility parameters `FLASHBACK_SCN` or `FLASHBACK_TIME` for Streams instantiations. For example, if an export includes objects with foreign key constraints, then more stringent consistency might be required.

Oracle Data Pump Import and Streams Instantiation

The following sections provide more information about Oracle Data Pump import and Streams instantiation.

Instantiation SCNs and Data Pump Imports

During Data Pump import, an instantiation SCN is set at the import database for each database object that was prepared for instantiation at the export database before the Data Pump export was performed. The instantiation SCN settings are based on metadata obtained during Data Pump export.

See Also: "Instantiation SCN and Ignore SCN for an Apply Process" on page 1-27

Instantiation SCNs and Streams Tags Resulting from Data Pump Imports

A Data Pump import session can set its Streams tag to the hexadecimal equivalent of `'00'` to avoid cycling the changes made by the import. Redo entries resulting from such an import have this tag value.

Whether the import session tag is set to the hexadecimal equivalent of `'00'` depends on the export that is being imported. Specifically, the import session tag is set to the hexadecimal equivalent of `'00'` in either of the following cases:

- The Data Pump export was in `FULL` or `SCHEMA` mode.
- The Data Pump export was in `TABLE` or `TABLESPACE` mode and at least one table included in the export was prepared for instantiation at the export database before the export was performed.

If neither one of these conditions is true for a Data Pump export that is being imported, then the import session tag is `NULL`.

Note:

- If you perform a network import using Data Pump, then an implicit export is performed in the same mode as the import. For example, if the network import is in schema mode, then the implicit export is in schema mode also.
- The import session tag is not set if the Data Pump import is performed in `TRANSPORTABLE TABLESPACE` mode. An import performed in this mode does not generate any redo data for the imported data. Therefore, setting the session tag is not required.

See Also: Chapter 4, "Streams Tags"

The STREAMS_CONFIGURATION Data Pump Import Utility Parameter

The `STREAMS_CONFIGURATION` Data Pump Import utility parameter specifies whether to import any general Streams metadata that is present in the export dump file. This import parameter is relevant only if you are performing a full database import. By default the `STREAMS_CONFIGURATION` Import utility parameter is set to `y`. Typically, specify `y` if an import is part of a backup or restore operation.

The following objects are imported regardless of the `STREAMS_CONFIGURATION` setting if the information is present in the export dump file:

- `ANYDATA` queues and their queue tables.
- Queue subscribers.
- Advanced Queuing agents.
- Job queue jobs related to Streams propagations.
- Rules, including their positive and negative rule sets and evaluation contexts. All rules are imported, including Streams rules and non-Streams rules. Streams rules are rules generated by the system when certain procedures in the `DBMS_STREAMS_ADM` package are run, while non-Streams rules are rules created using the `DBMS_RULE_ADM` package.

  If the `STREAMS_CONFIGURATION` parameter is set to `n`, then information about Streams rules is not imported into the following data dictionary views: `ALL_STREAMS_RULES`, `ALL_STREAMS_GLOBAL_RULES`, `ALL_STREAMS_SCHEMA_`

RULES, ALL_STREAMS_TABLE_RULES, DBA_STREAMS_RULES, DBA_STREAMS_GLOBAL_RULES, DBA_STREAMS_SCHEMA_RULES, and DBA_STREAMS_TABLE_RULES. However, regardless of the STREAMS_CONFIGURATION parameter setting, information about these rules is imported into the ALL_RULES, ALL_RULE_SETS, ALL_RULE_SET_RULES, DBA_RULES, DBA_RULE_SETS, DBA_RULE_SET_RULES, USER_RULES, USER_RULE_SETS, and USER_RULE_SET_RULES data dictionary views.

When the STREAMS_CONFIGURATION Import utility parameter is set to y, the import includes the following information, if the information is present in the export dump file; when the STREAMS_CONFIGURATION Import utility parameter is set to n, the import does not include the following information:

- Capture processes that capture local changes, including the following information for each capture process:
  - Name of the capture process.
  - State of the capture process.
  - Capture process parameter settings.
  - Queue owner and queue name of the queue used by the capture process.
  - Rule set owner and rule set name of each positive and negative rule set used by the capture process.
  - Capture user for the capture process.
  - Extra attribute settings if a capture process is configured to include extra attributes in LCRs.
  - The time that the status of the capture process last changed. This information is recorded in the DBA_CAPTURE data dictionary view.
  - If the capture process disabled or aborted, then the error number and message of the error that was the cause. This information is recorded in the DBA_CAPTURE data dictionary view.
- If any tables have been prepared for instantiation at the export database, then these tables are prepared for instantiation at the import database.
- If any schemas have been prepared for instantiation at the export database, then these schemas are prepared for instantiation at the import database.
- If the export database has been prepared for instantiation, then the import database is prepared for instantiation.
- The state of each ANYDATA queue that is used by a Streams client, either started or stopped. Streams clients include capture processes, propagations, apply process, and messaging clients. ANYDATA queues themselves are imported regardless of the STREAMS_CONFIGURATION Import utility parameter setting.
- Propagations, including the following information for each propagation:
  - Name of the propagation.
  - Queue owner and queue name of the source queue.
  - Queue owner and queue name of the destination queue.
  - Destination database link.
  - Rule set owner and rule set name of each positive and negative rule set used by the propagation.

- Apply processes, including the following information for each apply process:
  - Name of the apply process.
  - State of the apply process.
  - Apply process parameter settings.
  - Queue owner and queue name of the queue used by the apply process.
  - Rule set owner and rule set name of each positive and negative rule set used by the apply process.
  - Whether the apply process applies captured or user-enqueued messages.
  - Apply user for the apply process.
  - Message handler used by the apply process, if one exists.
  - DDL handler used by the apply process, if one exists.
  - Precommit handler used by the apply process, if one exists.
  - Tag generated in the redo log for changes made by the apply process.
  - Apply database link, if one exists.
  - Source database for the apply process.
  - The information about apply progress in the `DBA_APPLY_PROGRESS` data dictionary view, including applied message number, oldest message number (oldest SCN), apply time, and applied message create time.
  - Apply errors.
  - The time that the status of the apply process last changed. This information is recorded in the `DBA_APPLY` data dictionary view.
  - If the apply process disabled or aborted, then the error number and message of the error that was the cause. This information is recorded in the `DBA_APPLY` data dictionary view.
- DML handlers.
- Error handlers.
- Update conflict handlers.
- Substitute key columns for apply tables.
- Instantiation SCN for each apply object.
- Ignore SCN for each apply object.
- Messaging Clients, including the following information for each messaging client:
  - Name of the messaging client.
  - Queue owner and queue name of the queue used by the messaging client.
  - Rule set owner and rule set name of each positive and negative rule set used by the messaging client.
  - Message notification settings.
- Some data dictionary information about Streams rules. The rules themselves are imported regardless of the setting for the `STREAMS_CONFIGURATION` parameter.
- Data dictionary information about Streams administrators, messaging clients, message rules, and extra attributes used in message rules.

Note: Downstream capture processes are not included in an import regardless of the STREAMS_CONFIGURATION setting.

Recovery Manager (RMAN) and Streams Instantiation

The RMAN DUPLICATE and CONVERT DATABASE commands can instantiate an entire database, and the RMAN TRANSPORT TABLESPACE command can instantiate a tablespace or set of tablespaces. Using RMAN for instantiation usually is faster than other instantiation methods. The following sections contain information about using these RMAN commands for instantiation.

The RMAN DUPLICATE and CONVERT DATABASE Commands and Instantiation

The RMAN DUPLICATE command creates a copy of the target database in another location. The command uses an RMAN auxiliary instance to restore backups of the target database files and create a new database. In a Streams instantiation, the target database is the source database and the new database that is created is the destination database. The DUPLICATE command requires that the source and destination database are running on the same platform.

The RMAN CONVERT DATABASE command generates the datafiles and an initialization parameter file for a new destination database on a different platform. It also generates a script that creates the new destination database. These files can be used to instantiate an entire destination database that runs on a different platform than the source database but has the same endian format as the source database.

The RMAN DUPLICATE and CONVERT DATABASE commands do not set the instantiation SCN values for the database objects. The instantiation SCN values must be set manually during instantiation.

See Also:

- "Instantiating an Entire Database Using RMAN" on page 10-16
- *Oracle Database Backup and Recovery Advanced User's Guide* for instructions on using the RMAN DUPLICATE command and the RMAN CONVERT DATABASE command

The RMAN TRANSPORT TABLESPACE Command and Instantiation

The RMAN TRANSPORT TABLESPACE command uses Data Pump and an RMAN-managed auxiliary instance to export the database objects in a tablespace or tablespace set while the tablespace or tablespace set remains online in the source database. RMAN automatically starts up an auxiliary instance with a system-generated name. The RMAN TRANSPORT TABLESPACE command produces a Data Pump export dump file and datafiles for the tablespace or tablespaces.

Data Pump can be used to import the dump file at the destination database, or the ATTACH_TABLESPACES procedure in the DBMS_STREAMS_TABLESPACE_ADM package can be used to attach the tablespace or tablespaces to the destination database. Also, instantiation SCN values for the database objects in the tablespace or tablespaces are set automatically at the destination database when the tablespaces are imported or attached.

Note: The RMAN TRANSPORT TABLESPACE command does not support user-managed auxiliary instances.

See Also: "Instantiating Objects Using Transportable Tablespace from Backup with RMAN" on page 10-11

Original Export/Import and Streams Instantiation

This section describes parameters for the original Export and Import utilities that are relevant to Streams.

See Also:

- "Instantiating Objects Using Original Export/Import" on page 10-14
- *Oracle Streams Concepts and Administration* for information about performing a full database export/import on a database using Streams
- *Oracle Database Utilities* for information about performing exports and imports using the original Export and Import utilities

The OBJECT_CONSISTENT Export Utility Parameter and Streams

The `OBJECT_CONSISTENT` Export utility parameter specifies whether or not the Export utility repeatedly uses the `SET TRANSACTION READ ONLY` statement to ensure that the exported data and the exported procedural actions for each object are consistent to a single point in time. If `OBJECT_CONSISTENT` is set to `y`, then each object is exported in its own read-only transaction, even if it is partitioned. In contrast, if you use the `CONSISTENT` Export utility parameter, then there is only one read-only transaction.

When you perform an instantiation in a Streams environment, some degree of consistency is required for the database objects being instantiated. The `OBJECT_CONSISTENT` Export utility parameter is sufficient to ensure this consistency for Streams instantiations. If you are using an export dump file for other purposes in addition to a Streams instantiation, and these other purposes have more stringent consistency requirements than those provided by `OBJECT_CONSISTENT`, then you can use Export utility parameters `CONSISTENT`, `FLASHBACK_SCN`, or `FLASHBACK_TIME` for Streams instantiations. For example, if an export includes objects with foreign key constraints, then more stringent consistency might be required.

By default the `OBJECT_CONSISTENT` Export utility parameter is set to `n`. Specify `y` when an export is performed as part of a Streams instantiation and no more stringent Export utility parameter is needed.

Original Import Utility Parameters Relevant to Streams

The following parameters for the original Import utility are relevant to Streams.

The STREAMS_INSTANTIATION Import Utility Parameter and Streams

The `STREAMS_INSTANTIATION` Import utility parameter specifies whether to import Streams instantiation metadata that is present in the export dump file. When this parameter is set to `y`, and the export dump file contains the metadata for instantiation SCNs, an instantiation SCN is set at the import database for each database object imported.

In addition, when this parameter is set to `y`, the import session sets its Streams tag to the hexadecimal equivalent of `'00'` to avoid cycling the changes made by the import. Redo entries resulting from the import have this tag value.

By default the `STREAMS_INSTANTIATION` Import utility parameter is set to `n`. Specify `y` when an import is performed as part of a Streams instantiation.

See Also:

- "Instantiation SCN and Ignore SCN for an Apply Process" on page 1-27
- Chapter 4, "Streams Tags"

The STREAMS_CONFIGURATION Import Utility Parameter and Streams

The `STREAMS_CONFIGURATION` Import utility parameter behaves the same for the original Import utility and the Data Pump Import utility.

See Also: "The STREAMS_CONFIGURATION Data Pump Import Utility Parameter" on page 2-8 for more information about this parameter

3

Streams Conflict Resolution

Some Streams environments must use conflict handlers to resolve possible data conflicts that can result from sharing data between multiple databases.

This chapter contains these topics:

See Also: "Managing Streams Conflict Detection and Resolution" on page 9-22

About DML Conflicts in a Streams Environment

A **conflict** is a mismatch between the old values in an LCR and the expected data in a table. Conflicts can occur in a Streams environment that permits concurrent data manipulation language (DML) operations on the same data at multiple databases. In a Streams environment, DML conflicts can occur only when an apply process is applying a message that contains a row change resulting from a DML operation. This type of message is called a row logical change record, or row LCR. An apply process automatically detects conflicts caused by row LCRs.

For example, when two transactions originating at different databases update the same row at nearly the same time, a conflict can occur. When you configure a Streams environment, you must consider whether conflicts can occur. You can configure conflict resolution to resolve conflicts automatically, if your system design permits conflicts.

In general, you should try to design a Streams environment that avoids the possibility of conflicts. Using the conflict avoidance techniques discussed later in this chapter, most system designs can avoid conflicts in all or a large percentage of the shared data. However, many applications require that some percentage of the shared data be updatable at multiple databases at any time. If this is the case, then you must address the possibility of conflicts.

Note: An apply process does not detect DDL conflicts or conflicts resulting from user-enqueued messages. Make sure your environment avoids these types of conflicts.

See Also: *Oracle Streams Concepts and Administration* for more information about row LCRs

Conflict Types in a Streams Environment

You can encounter these types of conflicts when you share data at multiple databases:

- Update Conflicts in a Streams Environment
- Uniqueness Conflicts in a Streams Environment
- Delete Conflicts in a Streams Environment
- Foreign Key Conflicts in a Streams Environment

Update Conflicts in a Streams Environment

An **update conflict** occurs when the apply process applies a row LCR containing an update to a row that conflicts with another update to the same row. Update conflicts can happen when two transactions originating from different databases update the same row at nearly the same time.

Uniqueness Conflicts in a Streams Environment

A **uniqueness conflict** occurs when the apply process applies a row LCR containing a change to a row that violates a uniqueness integrity constraint, such as a `PRIMARY KEY` or `UNIQUE` constraint. For example, consider what happens when two transactions originate from two different databases, each inserting a row into a table with the same primary key value. In this case, the transactions cause a uniqueness conflict.

Delete Conflicts in a Streams Environment

A **delete conflict** occurs when two transactions originate at different databases, with one transaction deleting a row and another transaction updating or deleting the same row. In this case, the row referenced in the row LCR does not exist to be either updated or deleted.

Foreign Key Conflicts in a Streams Environment

A **foreign key conflict** occurs when the apply process applies a row LCR containing a change to a row that violates a foreign key constraint. For example, in the `hr` schema, the `department_id` column in the `employees` table is a foreign key of the `department_id` column in the `departments` table. Consider what can happen when the following changes originate at two different databases (`A` and `B`) and are propagated to a third database (`C`):

- At database `A`, a row is inserted into the `departments` table with a `department_id` of `271`. This change is propagated to database `B` and applied there.
- At database `B`, a row is inserted into the `employees` table with an `employee_id` of `206` and a `department_id` of `271`.

If the change that originated at database B is applied at database C before the change that originated at database A, then a foreign key conflict results because the row for the department with a department_id of 271 does not yet exist in the departments table at database C.

Conflicts and Transaction Ordering in a Streams Environment

Ordering conflicts can occur in a Streams environment when three or more databases share data and the data is updated at two or more of these databases. For example, consider a scenario in which three databases share information in the hr.departments table. The database names are mult1.net, mult2.net, and mult3.net. Suppose a change is made to a row in the hr.departments table at mult1.net that will be propagated to both mult2.net and mult3.net. The following series of actions might occur:

1. The change is propagated to mult2.net.
2. An apply process at mult2.net applies the change from mult1.net.
3. A different change to the same row is made at mult2.net.
4. The change at mult2.net is propagated to mult3.net.
5. An apply process at mult3.net attempts to apply the change from mult2.net before another apply process at mult3.net applies the change from mult1.net.

In this case, a conflict occurs because a column value for the row at mult3.net does not match the corresponding old value in the row LCR propagated from mult2.net.

In addition to causing a data conflict, transactions that are applied out of order might experience referential integrity problems at a remote database if supporting data has not been successfully propagated to that database. Consider the scenario where a new customer calls an order department. A customer record is created and an order is placed. If the order data is applied at a remote database before the customer data, then a referential integrity error is raised because the customer that the order references does not exist at the remote database.

If an ordering conflict is encountered, then you can resolve the conflict by reexecuting the transaction in the error queue after the required data has been propagated to the remote database and applied.

Conflict Detection in a Streams Environment

An apply process detects update, uniqueness, delete, and foreign key conflicts as follows:

- An apply process detects an update conflict if there is any difference between the old values for a row in a row LCR and the current values of the same row at the destination database.
- An apply process detects a uniqueness conflict if a uniqueness constraint violation occurs when applying an LCR that contains an insert or update operation.
- An apply process detects a delete conflict if it cannot find a row when applying an LCR that contains an update or delete operation, because the primary key of the row does not exist.
- An apply process detects a foreign key conflict if a foreign key constraint violation occurs when applying an LCR.

A conflict can be detected when an apply process attempts to apply an LCR directly or when an apply process handler, such as a DML handler, runs the EXECUTE member procedure for an LCR. A conflict can also be detected when either the EXECUTE_ERROR or EXECUTE_ALL_ERRORS procedure in the DBMS_APPLY_ADM package is run.

Note:

- If a column is updated and the column's old value equals its new value, then Oracle never detects a conflict for this column update.
- Any old LOB values in update LCRs, delete LCRs, and LCRs dealing with piecewise updates to LOB columns are not used by conflict detection.

Control Over Conflict Detection for Nonkey Columns

By default, an apply process compares old values for all columns during conflict detection, but you can stop conflict detection for nonkey columns using the COMPARE_OLD_VALUES procedure in the DBMS_APPLY_ADM package. Conflict detection might not be needed for some nonkey columns.

See Also:

- "Stopping Conflict Detection for Nonkey Columns" on page 9-25
- "Displaying Information About Conflict Detection" on page 12-11

Rows Identification During Conflict Detection in a Streams Environment

To detect conflicts accurately, Oracle must be able to identify and match corresponding rows at different databases uniquely. By default, Oracle uses the primary key of a table to identify rows in a table uniquely. When a table does not have a primary key, you should designate a substitute key. A substitute key is a column or set of columns that Oracle can use to identify uniquely rows in the table.

See Also: "Substitute Key Columns" on page 1-19

Conflict Avoidance in a Streams Environment

This section describes ways to avoid data conflicts.

Use a Primary Database Ownership Model

You can avoid the possibility of conflicts by limiting the number of databases in the system that have simultaneous update access to the tables containing shared data. Primary ownership prevents all conflicts, because only a single database permits updates to a set of shared data. Applications can even use row and column subsetting to establish more granular ownership of data than at the table level. For example, applications might have update access to specific columns or rows in a shared table on a database-by-database basis.

Avoid Specific Types of Conflicts

If a primary database ownership model is too restrictive for your application requirements, then you can use a shared ownership data model, which means that conflicts might be possible. Even so, typically you can use some simple strategies to avoid specific types of conflicts.

Avoid Uniqueness Conflicts in a Streams Environment

You can avoid uniqueness conflicts by ensuring that each database uses unique identifiers for shared data. There are three ways to ensure unique identifiers at all databases in a Streams environment.

One way is to construct a unique identifier by executing the following select statement:

```
SELECT SYS_GUID() OID FROM DUAL;
```

This SQL operator returns a 16-byte globally unique identifier. This value is based on an algorithm that uses time, date, and the computer identifier to generate a globally unique identifier. The globally unique identifier appears in a format similar to the following:

```
A741C791252B3EA0E034080020AE3E0A
```

Another way to avoid uniqueness conflicts is to create a sequence at each of the databases that shares data and concatenate the database name (or other globally unique value) with the local sequence. This approach helps to avoid any duplicate sequence values and helps to prevent uniqueness conflicts.

Finally, you can create a customized sequence at each of the databases that shares data so that no two databases can generate the same value. You can accomplish this by using a combination of starting, incrementing, and maximum values in the `CREATE SEQUENCE` statement. For example, you might configure the following sequences:

Table 3–1 Customized Sequences for Streams Replication Environments

Parameter	Database A	Database B	Database C
`START WITH`	1	3	5
`INCREMENT BY`	10	10	10
Range Example	1, 11, 21, 31, 41,...	3, 13, 23, 33, 43,...	5, 15, 25, 35, 45,...

Using a similar approach, you can define different ranges for each database by specifying a `START WITH` and `MAXVALUE` that would produce a unique range for each database.

Avoid Delete Conflicts in a Streams Environment

Always avoid delete conflicts in shared data environments. In general, applications that operate within a shared ownership data model should not delete rows using `DELETE` statements. Instead, applications should mark rows for deletion and then configure the system to purge logically deleted rows periodically.

Avoid Update Conflicts in a Streams Environment

After trying to eliminate the possibility of uniqueness and delete conflicts, you should also try to limit the number of possible update conflicts. However, in a shared ownership data model, update conflicts cannot be avoided in all cases. If you cannot avoid all update conflicts, then you must understand the types of conflicts possible and configure the system to resolve them if they occur.

Conflict Resolution in a Streams Environment

After an update conflict has been detected, a conflict handler can attempt to resolve it. Streams provides prebuilt conflict handlers to resolve update conflicts, but not uniqueness, delete, foreign key, or ordering conflicts. However, you can build your own custom conflict handler to resolve data conflicts specific to your business rules. Such a conflict handler can be part of a DML handler or an error handler.

Whether you use prebuilt or custom conflict handlers, a conflict handler is applied as soon as a conflict is detected. If neither the specified conflict handler nor the relevant apply handler can resolve the conflict, then the conflict is logged in the error queue. You might want to use the relevant apply handler to notify the database administrator when a conflict occurs.

When a conflict causes a transaction to be moved to the error queue, sometimes it is possible to correct the condition that caused the conflict. In these cases, you can reexecute a transaction using the `EXECUTE_ERROR` procedure in the `DBMS_APPLY_ADM` package.

See Also:

- *Oracle Streams Concepts and Administration* for more information about DML handlers, error handlers, and the error queue
- "Handlers and Row LCR Processing" on page 1-22 for more information about how update conflict handlers interact with DML handlers and error handlers
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `EXECUTE_ERROR` procedure in the `DBMS_APPLY_ADM` package

Prebuilt Update Conflict Handlers

This section describes the types of prebuilt update conflict handlers available to you and how column lists and resolution columns are used in prebuilt update conflict handlers. A column list is a list of columns for which the update conflict handler is called when there is an update conflict. The resolution column is the column used to identify an update conflict handler. If you use a `MAXIMUM` or `MINIMUM` prebuilt update conflict handler, then the resolution column is also the column used to resolve the conflict. The resolution column must be one of the columns in the column list for the handler.

Use the `SET_UPDATE_CONFLICT_HANDLER` procedure in the `DBMS_APPLY_ADM` package to specify one or more update conflict handlers for a particular table. There are no prebuilt conflict handlers for uniqueness, delete, or foreign key conflicts.

See Also:

- "Managing Streams Conflict Detection and Resolution" on page 9-22 for instructions on adding, modifying, and removing an update conflict handler
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `SET_UPDATE_CONFLICT_HANDLER` procedure
- "Column Lists" on page 3-9
- "Resolution Columns" on page 3-10

Types of Prebuilt Update Conflict Handlers

Oracle provides the following types of prebuilt update conflict handlers for a Streams environment: `OVERWRITE`, `DISCARD`, `MAXIMUM`, and `MINIMUM`.

The description for each type of handler later in this section refers to the following conflict scenario:

1. The following update is made at the `dbs1.net` source database:

   ```
   UPDATE hr.employees SET salary = 4900 WHERE employee_id = 200;
   COMMIT;
   ```

   This update changes the salary for employee `200` from `4400` to `4900`.

2. At nearly the same time, the following update is made at the `dbs2.net` destination database:

   ```
   UPDATE hr.employees SET salary = 5000 WHERE employee_id = 200;
   COMMIT;
   ```

3. A capture process captures the update at the `dbs1.net` source database and puts the resulting row LCR in a queue.
4. A propagation propagates the row LCR from the queue at `dbs1.net` to a queue at `dbs2.net`.
5. An apply process at `dbs2.net` attempts to apply the row LCR to the `hr.employees` table but encounters a conflict because the salary value at `dbs2.net` is `5000`, which does not match the old value for the salary in the row LCR (`4400`).

The following sections describe each prebuilt conflict handler and explain how the handler resolves this conflict.

OVERWRITE When a conflict occurs, the `OVERWRITE` handler replaces the current value at the destination database with the new value in the LCR from the source database.

If the `OVERWRITE` handler is used for the `hr.employees` table at the `dbs2.net` destination database in the conflict example, then the new value in the row LCR overwrites the value at `dbs2.net`. Therefore, after the conflict is resolved, the salary for employee `200` is `4900`.

DISCARD When a conflict occurs, the `DISCARD` handler ignores the values in the LCR from the source database and retains the value at the destination database.

If the `DISCARD` handler is used for the `hr.employees` table at the `dbs2.net` destination database in the conflict example, then the new value in the row LCR is

discarded. Therefore, after the conflict is resolved, the salary for employee 200 is 5000 at dbs2.net.

MAXIMUM When a conflict occurs, the MAXIMUM conflict handler compares the new value in the LCR from the source database with the current value in the destination database for a designated resolution column. If the new value of the resolution column in the LCR is greater than the current value of the column at the destination database, then the apply process resolves the conflict in favor of the LCR. If the new value of the resolution column in the LCR is less than the current value of the column at the destination database, then the apply process resolves the conflict in favor of the destination database.

If the MAXIMUM handler is used for the salary column in the hr.employees table at the dbs2.net destination database in the conflict example, then the apply process does not apply the row LCR, because the salary in the row LCR is less than the current salary in the table. Therefore, after the conflict is resolved, the salary for employee 200 is 5000 at dbs2.net.

If you want to resolve conflicts based on the time of the transactions involved, then one way to do this is to add a column to a shared table that automatically records the transaction time with a trigger. You can designate this column as a resolution column for a MAXIMUM conflict handler, and the transaction with the latest (or greater) time would be used automatically.

The following is an example of a trigger that records the time of a transaction for the hr.employees table. Assume that the job_id, salary, and commission_pct columns are part of the column list for the conflict resolution handler. The trigger should fire only when an UPDATE is performed on the columns in the column list or when an INSERT is performed.

```
CONNECT hr/hr

ALTER TABLE hr.employees ADD (time TIMESTAMP WITH TIME ZONE);

CREATE OR REPLACE TRIGGER hr.insert_time_employees
BEFORE
  INSERT OR UPDATE OF job_id, salary, commission_pct ON hr.employees
FOR EACH ROW
BEGIN
   -- Consider time synchronization problems. The previous update to this
   -- row might have originated from a site with a clock time ahead of the
   -- local clock time.
   IF :OLD.TIME IS NULL OR :OLD.TIME < SYSTIMESTAMP THEN
     :NEW.TIME := SYSTIMESTAMP;
   ELSE
     :NEW.TIME := :OLD.TIME + 1 / 86400;
   END IF;
END;
/
```

If you use such a trigger for conflict resolution, then make sure the trigger's firing property is fire once, which is the default. Otherwise, a new time might be marked when transactions are applied by an apply process, resulting in the loss of the actual time of the transaction.

See Also: "Trigger Firing Property" on page 1-29

MINIMUM When a conflict occurs, the `MINIMUM` conflict handler compares the new value in the LCR from the source database with the current value in the destination database for a designated resolution column. If the new value of the resolution column in the LCR is less than the current value of the column at the destination database, then the apply process resolves the conflict in favor of the LCR. If the new value of the resolution column in the LCR is greater than the current value of the column at the destination database, then the apply process resolves the conflict in favor of the destination database.

If the `MINIMUM` handler is used for the `salary` column in the `hr.employees` table at the `dbs2.net` destination database in the conflict example, then the apply process resolves the conflict in favor of the row LCR, because the salary in the row LCR is less than the current salary in the table. Therefore, after the conflict is resolved, the salary for employee `200` is `4900`.

Column Lists

Each time you specify a prebuilt update conflict handler for a table, you must specify a **column list**. A column list is a list of columns for which the update conflict handler is called. If an update conflict occurs for one or more of the columns in the list when an apply process tries to apply a row LCR, then the update conflict handler is called to resolve the conflict. The update conflict handler is not called if a conflict occurs only in columns that are not in the list. The scope of conflict resolution is a single column list on a single row LCR.

You can specify more than one update conflict handler for a particular table, but the same column cannot be in more than one column list. For example, suppose you specify two prebuilt update conflict handlers on `hr.employees` table:

- The first update conflict handler has the following columns in its column list: `salary` and `commission_pct`.
- The second update conflict handler has the following columns in its column list: `job_id` and `department_id`.

Also, assume that no other conflict handlers exist for this table. In this case, if a conflict occurs for the `salary` column when an apply process tries to apply a row LCR, then the first update conflict handler is called to resolve the conflict. If, however, a conflict occurs for the `department_id` column, then the second update conflict handler is called to resolve the conflict. If a conflict occurs for a column that is not in a column list for any conflict handler, then no conflict handler is called, and an error results. In this example, if a conflict occurs for the `manager_id` column in the `hr.employees` table, then an error results. If conflicts occur in more than one column list when a row LCR is being applied, and there are no conflicts in any columns that are not in a column list, then the appropriate update conflict handler is invoked for each column list with a conflict.

Column lists enable you to use different handlers to resolve conflicts for different types of data. For example, numeric data is often suited for a maximum or minimum conflict handler, while an overwrite or discard conflict handler might be preferred for character data.

If a conflict occurs in a column that is not in a column list, then the error handler for the specific operation on the table attempts to resolve the conflict. If the error handler cannot resolve the conflict, or if there is no such error handler, then the transaction that caused the conflict is moved to the error queue.

Also, if a conflict occurs for a column in a column list that uses either the `OVERWRITE`, `MAXIMUM`, or `MINIMUM` prebuilt handler, and the row LCR does not contain all of the columns in this column list, then the conflict cannot be resolved because all of the

values are not available. In this case, the transaction that caused the conflict is moved to the error queue. If the column list uses the DISCARD prebuilt method, then the row LCR is discarded and no error results, even if the row LCR does not contain all of the columns in this column list.

A conditional supplemental log group must be specified for the columns specified in a column list if more than one column at the source database affects the column list at the destination database. Supplemental logging is specified at the source database and adds additional information to the LCR, which is needed to resolve conflicts properly. Typically, a conditional supplemental log group must be specified for the columns in a column list if there is more than one column in the column list, but not if there is only one column in the column list.

However, in some cases, a conditional supplemental log group is required even if there is only one column in a column list. That is, an apply handler or custom rule-based transformation can combine multiple columns from the source database into a single column in the column list at the destination database. For example, a custom rule-based transformation can take three columns that store street, state, and postal code data from a source database and combine the data into a single address column at a destination database.

Also, in some cases, no conditional supplemental log group is required even if there is more than one column in a column list. For example, an apply handler or custom rule-based transformation can separate one address column from the source database into multiple columns that are in a column list at the destination database. A custom rule-based transformation can take an address that includes street, state, and postal code data in one address column at a source database and separate the data into three columns at a destination database.

Note: Prebuilt update conflict handlers do not support LOB, LONG, LONG RAW, and user-defined type columns. Therefore, you should not include these types of columns in the column_list parameter when running the SET_UPDATE_CONFLICT_HANDLER procedure.

See Also: "Supplemental Logging for Streams Replication" on page 1-8

Resolution Columns

The **resolution column** is the column used to identify a prebuilt update conflict handler. If you use a MAXIMUM or MINIMUM prebuilt update conflict handler, then the resolution column is also the column used to resolve the conflict. The resolution column must be one of the columns in the column list for the handler.

For example, if the salary column in the hr.employees table is specified as the resolution column for a maximum or minimum conflict handler, then the salary column is evaluated to determine whether column list values in the row LCR are applied or the destination database values for the column list are retained.

In either of the following situations involving a resolution column for a conflict, the apply process moves the transaction containing the row LCR that caused the conflict to the error queue, if the error handler cannot resolve the problem. In these cases, the conflict cannot be resolved and the values of the columns at the destination database remain unchanged:

- The new LCR value and the destination row value for the resolution column are the same (for example, if the resolution column was not the column causing the conflict).
- Either the new LCR value of the resolution column or the current value of the resolution column at the destination database is NULL.

Note: Although the resolution column is not used for OVERWRITE and DISCARD conflict handlers, a resolution column must be specified for these conflict handlers.

Data Convergence

When you share data between multiple databases, and you want the data to be the same at all of these databases, then make sure you use conflict resolution handlers that cause the data to converge at all databases. If you allow changes to shared data at all of your databases, then data convergence for a table is possible only if all databases that are sharing data capture changes to the shared data and propagate these changes to all of the other databases that are sharing the data.

In such an environment, the MAXIMUM conflict resolution method can guarantee convergence only if the values in the resolution column are always increasing. A time-based resolution column meets this requirement, as long as successive timestamps on a row are distinct. The MINIMUM conflict resolution method can guarantee convergence in such an environment only if the values in the resolution column are always decreasing.

Custom Conflict Handlers

You can create a PL/SQL procedure to use as a custom conflict handler. You use the SET_DML_HANDLER procedure in the DBMS_APPLY_ADM package to designate one or more custom conflict handlers for a particular table. Specifically, set the following parameters when you run this procedure to specify a custom conflict handler:

- Set the object_name parameter to the fully qualified name of the table for which you want to perform conflict resolution.
- Set the object_type parameter to TABLE.
- Set the operation_name parameter to the type of operation for which the custom conflict handler is called. The possible operations are the following: INSERT, UPDATE, DELETE, and LOB_UPDATE.
- If you want an error handler to perform conflict resolution when an error is raised, then set the error_handler parameter to true. Or, if you want to include conflict resolution in your DML handler, then set the error_handler parameter to false.

  If you specify false for this parameter, then, when you execute a row LCR using the EXECUTE member procedure for the LCR, the conflict resolution within the DML handler is performed for the specified object and operation(s).
- Specify the procedure to resolve a conflict by setting the user_procedure parameter. This user procedure is called to resolve any conflicts on the specified table resulting from the specified type of operation.

If the custom conflict handler cannot resolve the conflict, then the apply process moves the transaction containing the conflict to the error queue and does not apply the transaction.

If both a prebuilt update conflict handler and a custom conflict handler exist for a particular object, then the prebuilt update conflict handler is invoked only if both of the following conditions are met:

- The custom conflict handler executes the row LCR using the `EXECUTE` member procedure for the LCR.
- The `conflict_resolution` parameter in the `EXECUTE` member procedure for the row LCR is set to `true`.

See Also:

- "Handlers and Row LCR Processing" on page 1-22 for more information about how update conflict handlers interact with DML handlers and error handlers
- "Managing a DML Handler" on page 9-12
- *Oracle Streams Concepts and Administration* for more information about managing error handlers
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `SET_DML_HANDLER` procedure

4

Streams Tags

This chapter explains the concepts related to Streams tags.

This chapter contains these topics:

- Introduction to Tags
- Tags and Rules Created by the DBMS_STREAMS_ADM Package
- Tags and Online Backup Statements
- Tags and an Apply Process
- Streams Tags in a Replication Environment

See Also: "Managing Streams Tags" on page 9-26

Introduction to Tags

Every redo entry in the redo log has a **tag** associated with it. The datatype of the tag is `RAW`. By default, when a user or application generates redo entries, the value of the tag is `NULL` for each redo entry, and a `NULL` tag consumes no space. The size limit for a tag value is 2000 bytes.

You can configure how tag values are interpreted. For example, a tag can be used to determine whether an LCR contains a change that originated in the local database or at a different database, so that you can avoid change cycling (sending an LCR back to the database where it originated). Tags can be used for other LCR tracking purposes as well. You can also use tags to specify the set of destination databases for each LCR.

You can control the value of the tags generated in the redo log in the following ways:

- Use the `DBMS_STREAMS.SET_TAG` procedure to specify the value of the redo tags generated in the current session. When a database change is made in the session, the tag becomes part of the redo entry that records the change. Different sessions can have the same tag setting or different tag settings.
- Use the `CREATE_APPLY` or `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package to control the value of the redo tags generated when an apply process runs. All sessions coordinated by the apply process coordinator use this tag setting. By default, redo entries generated by an apply process have a tag value that is the hexadecimal equivalent of `'00'` (double zero).

Based on the rules in the rule sets for the capture process, the tag value in the redo entry for a change can determine whether or not the change is captured. For captured changes, the tags become part of the LCRs captured by a capture process retrieving changes from the redo log.

Similarly, once a tag is part of an LCR, the value of the tag can determine whether a propagation propagates the LCR and whether an apply process applies the LCR. The behavior of a custom rule-based transformation, DML handler, or error handler can also depend on the value of the tag. In addition, you can set the tag value for an existing LCR using the SET_TAG member procedure for the LCR. For example, you can set a tag in an LCR during a custom rule-based transformation.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the SET_TAG member procedure for LCRs
- *Oracle Streams Concepts and Administration* for more information about how rules are used in Streams

Tags and Rules Created by the DBMS_STREAMS_ADM Package

When you use a procedure in the DBMS_STREAMS_ADM package to create rules and set the include_tagged_lcr parameter to false, each rule contains a condition that evaluates to TRUE only if the tag is NULL. In DML rules, the condition is the following:

```
:dml.is_null_tag()='Y'
```

In DDL rules, the condition is the following:

```
:ddl.is_null_tag()='Y'
```

Consider a positive rule set with a single rule and assume the rule contains such a condition. In this case, Streams capture processes, propagations, and apply processes behave in the following way:

- A capture process captures a change only if the tag in the redo log entry for the change is NULL and the rest of the rule conditions evaluate to TRUE for the change.
- A propagation propagates an LCR only if the tag in the LCR is NULL and the rest of the rule conditions evaluate to TRUE for the LCR.
- An apply process applies an LCR only if the tag in the LCR is NULL and the rest of the rule conditions evaluate to TRUE for the LCR.

Alternatively, consider a negative rule set with a single rule and assume the rule contains such a condition. In this case, Streams capture processes, propagations, and apply processes behave in the following way:

- A capture process discards a change only if the tag in the redo log entry for the change is NULL and the rest of the rule conditions evaluate to TRUE for the change.
- A propagation or apply process discards LCR only if the tag in the LCR is NULL and the rest of the rule conditions evaluate to TRUE for the LCR.

In most cases, specify true for the include_tagged_lcr parameter if rules are being added to a negative rule set so that changes are discarded regardless of their tag values.

The following procedures in the DBMS_STREAMS_ADM package create rules that contain one of these conditions by default:

- ADD_GLOBAL_PROPAGATION_RULES
- ADD_GLOBAL_RULES
- ADD_SCHEMA_PROPAGATION_RULES
- ADD_SCHEMA_RULES

- ADD_SUBSET_PROPAGATION_RULES
- ADD_SUBSET_RULES
- ADD_TABLE_PROPAGATION_RULES
- ADD_TABLE_RULES

If you do not want the rules to contain such a condition, then set the include_tagged_lcr parameter to true when you run these procedures. This setting results in no conditions relating to tags in the rules. Therefore, rule evaluation of the LCR does not depend on the value of the tag.

For example, consider a table rule that evaluates to TRUE for all DML changes to the hr.locations table that originated at the dbs1.net source database.

Assume the ADD_TABLE_RULES procedure is run to generate this rule:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name              =>  'hr.locations',
    streams_type            =>  'capture',
    streams_name            =>  'capture',
    queue_name              =>  'streams_queue',
    include_tagged_lcr      =>  false,  -- Note parameter setting
    source_database         =>  'dbs1.net',
    include_dml             =>  true,
    include_ddl             =>  false);
END;
/
```

Notice that the include_tagged_lcr parameter is set to false, which is the default. The ADD_TABLE_RULES procedure generates a rule with a rule condition similar to the following:

```
(((:dml.get_object_owner() = 'HR' and :dml.get_object_name() = 'LOCATIONS'))
and :dml.is_null_tag() = 'Y' and :dml.get_source_database_name() = 'DBS1.NET' )
```

If a capture process uses a positive rule set that contains this rule, then the rule evaluates to FALSE if the tag for a change in a redo entry is a non-NULL value, such as '0' or '1'. So, if a redo entry contains a row change to the hr.locations table, then the change is captured only if the tag for the redo entry is NULL.

However, suppose the include_tagged_lcr parameter is set to true when ADD_TABLE_RULES is run:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name              =>  'hr.locations',
    streams_type            =>  'capture',
    streams_name            =>  'capture',
    queue_name              =>  'streams_queue',
    include_tagged_lcr      =>  true,   -- Note parameter setting
    source_database         =>  'dbs1.net',
    include_dml             =>  true,
    include_ddl             =>  false);
END;
/
```

In this case, the ADD_TABLE_RULES procedure generates a rule with a rule condition similar to the following:

```
(((:dml.get_object_owner() = 'HR' and :dml.get_object_name() = 'LOCATIONS'))
and :dml.get_source_database_name() = 'DBS1.NET' )
```

Notice that there is no condition relating to the tag. If a capture process uses a positive rule set that contains this rule, then the rule evaluates to TRUE if the tag in a redo entry for a DML change to the hr.locations table is a non-NULL value, such as '0' or '1'. The rule also evaluates to TRUE if the tag is NULL. So, if a redo entry contains a DML change to the hr.locations table, then the change is captured regardless of the value for the tag.

If you want to modify the is_null_tag condition in an existing system-created rule, then you should use an appropriate procedure in the DBMS_STREAMS_ADM package to create a new rule that is the same as the rule you want to modify, except for the is_null_tag condition. Next, use the REMOVE_RULE procedure in the DBMS_STREAMS_ADM package to remove the old rule from the appropriate rule set. In addition, you can use the and_condition parameter for the procedures that create rules in the DBMS_STREAMS_ADM package to add conditions relating to tags to system-created rules.

If you created a rule with the DBMS_RULE_ADM package, then you can add, remove, or modify the is_null_tag condition in the rule by using the ALTER_RULE procedure in this package.

See Also:

- *Oracle Streams Concepts and Administration* for examples of rules generated by the procedures in the DBMS_STREAMS_ADM package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_STREAMS_ADM package and the DBMS_RULE_ADM.ALTER_RULE procedure
- "Setting the Tag Values Generated by an Apply Process" on page 9-27 for more information about the SET_TAG procedure

Tags and Online Backup Statements

If you are using global rules to capture and apply DDL changes for an entire database, then online backup statements will be captured, propagated, and applied by default. Typically, database administrators do not want to replicate online backup statements. Instead, they only want them to run at the database where they are executed originally. An online backup statement uses the BEGIN BACKUP and END BACKUP clauses in an ALTER TABLESPACE or ALTER DATABASE statement.

To avoid replicating online backup statements, you can use one of the following strategies:

- Include one or more calls to the DBMS_STREAMS.SET_TAG procedure in your online backup procedures, and set the session tag to a value that will cause the online backup statements to be ignored by a capture process.
- Use a DDL handler for an apply process to avoid applying the online backup statements.

Note: If you use Recovery Manager (RMAN) to perform an online backup, then the online backup statements are not used, and there is no need to set Streams tags for backups.

See Also: *Oracle Database Backup and Recovery Advanced User's Guide* for information about making backups

Tags and an Apply Process

An apply process generates entries in the redo log of a destination database when it applies DML or DDL changes. For example, if the apply process applies a change that updates a row in a table, then that change is recorded in the redo log at the destination database. You can control the tags in these redo entries by setting the `apply_tag` parameter in the `CREATE_APPLY` or `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package. For example, an apply process can generate redo tags that are equivalent to the hexadecimal value of `'0'` (zero) or `'1'`.

The default tag value generated in the redo log by an apply process is `'00'` (double zero). This value is the default tag value for an apply process if you use a procedure in the `DBMS_STREAMS_ADM` package or the `CREATE_APPLY` procedure in the `DBMS_APPLY_ADM` package to create the apply process. There is nothing special about this value beyond the fact that it is a non-`NULL` value. The fact that it is a non-`NULL` value is important because rules created by the `DBMS_STREAMS_ADM` package by default contain a condition that evaluates to `TRUE` only if the tag is `NULL` in a redo entry or an LCR. You can alter the tag value for an existing apply process using the `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package.

Redo entries generated by an apply handler for an apply process have the tag value of the apply process, unless the handler sets the tag to a different value using the `SET_TAG` procedure. If a DML handler, DDL handler, or message handler calls the `SET_TAG` procedure in the `DBMS_STREAMS` package, then any subsequent redo entries generated by the handler will include the tag specified in the `SET_TAG` call, even if the tag for the apply process is different. When the handler exits, any subsequent redo entries generated by the apply process have the tag specified for the apply process.

See Also:

- "Apply and Streams Replication" on page 1-12 for more information about the apply process
- "Tags and Rules Created by the DBMS_STREAMS_ADM Package" on page 4-2 for more information about the default tag condition in Streams rules
- "Managing Streams Tags" on page 9-26
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_STREAMS_ADM` package and the `DBMS_APPLY_ADM` package

Streams Tags in a Replication Environment

In a Streams environment that includes more than one database sharing data bidirectionally, you can use tags to avoid **change cycling**. Change cycling means sending a change back to the database where it originated. Typically, change cycling should be avoided because it can result in each change going through endless loops back to the database where it originated. Such loops can result in unintended data in the database and tax the networking and computer resources of an environment. By default, Streams is designed to avoid change cycling.

Using tags and appropriate rules for Streams capture processes, propagations, and apply processes, you can avoid such change cycles. This section describes various Streams environments and how tags and rules can be used to avoid change cycling in these environments.

This section contains these topics:

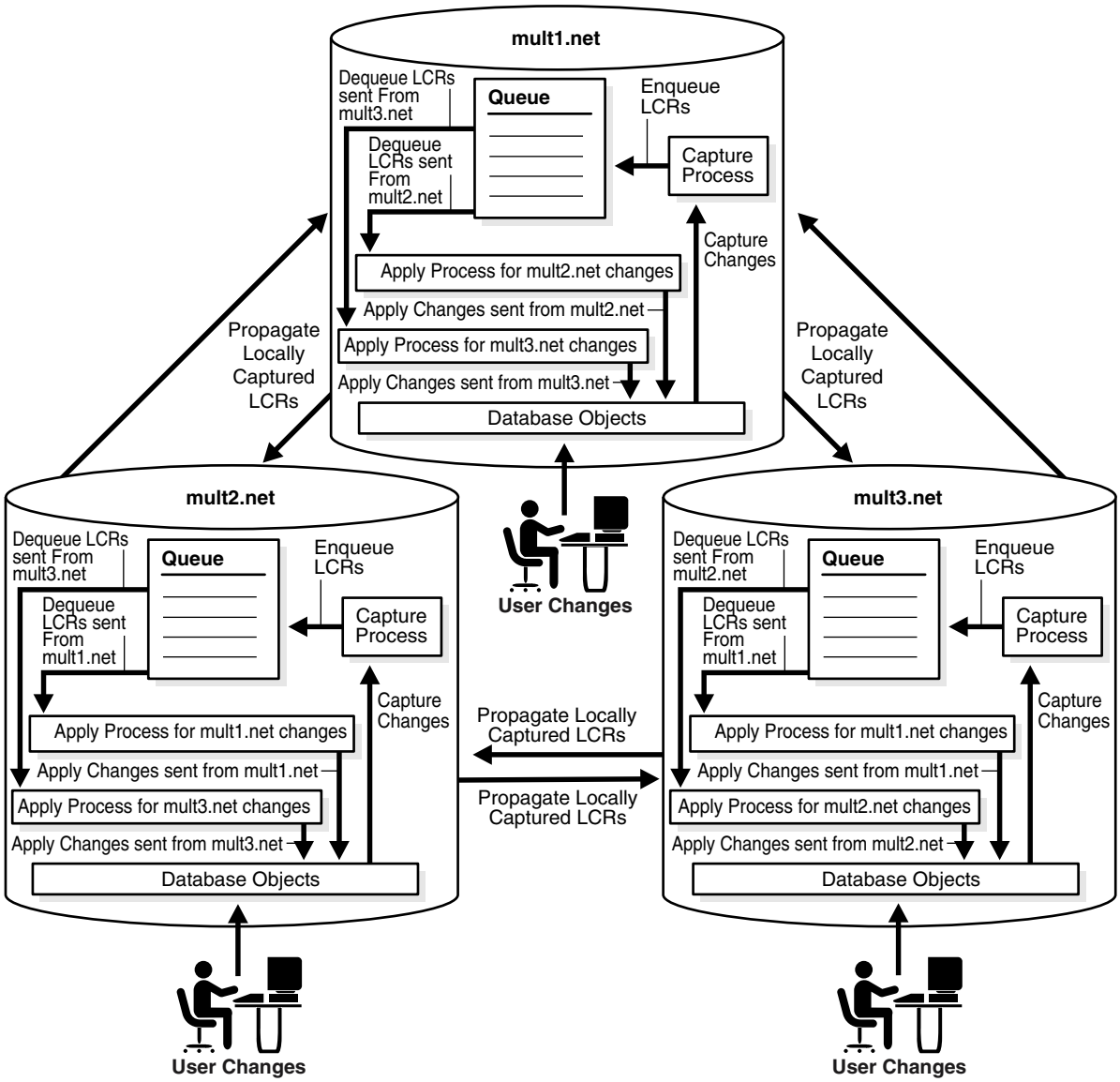
- Each Databases Is a Source and Destination Database for Shared Data
- Primary Database Sharing Data with Several Secondary Databases
- Primary Database Sharing Data with Several Extended Secondary Databases

Each Databases Is a Source and Destination Database for Shared Data

This scenario involves a Streams environment in which each database is a source database for every other database, and each database is a destination database of every other database. Each database communicates directly with every other database.

For example, consider an environment that replicates the database objects and data in the `hr` schema between three Oracle databases: `mult1.net`, `mult2.net`, and `mult3.net`. DML and DDL changes made to tables in the `hr` schema are captured at all three databases in the environment and propagated to each of the other databases in the environment, where changes are applied. Figure 4–1 illustrates a sample environment in which each database is a source database.

Figure 4–1 Each Database Is a Source and Destination Database



You can avoid change cycles by configuring such an environment in the following way:

- Configure one apply process at each database to generate non-NULL redo tags for changes from each source database. If you use a procedure in the DBMS_STREAMS_ADM package to create an apply process, then the apply process generates non-NULL tags with a value of '00' in the redo log by default. In this case, no further action is required for the apply process to generate non-NULL tags.

  If you use the CREATE_APPLY procedure in the DBMS_APPLY_ADM package to create an apply process, then do not set the apply_tag parameter. Again, the apply process generates non-NULL tags with a value of '00' in the redo log by default, and no further action is required.

- Configure the capture process at each database to capture changes only if the tag in the redo entry for the change is `NULL`. You do this by ensuring that each DML rule in the positive rule set used by the capture process has the following condition:

  ```
  :dml.is_null_tag()='Y'
  ```

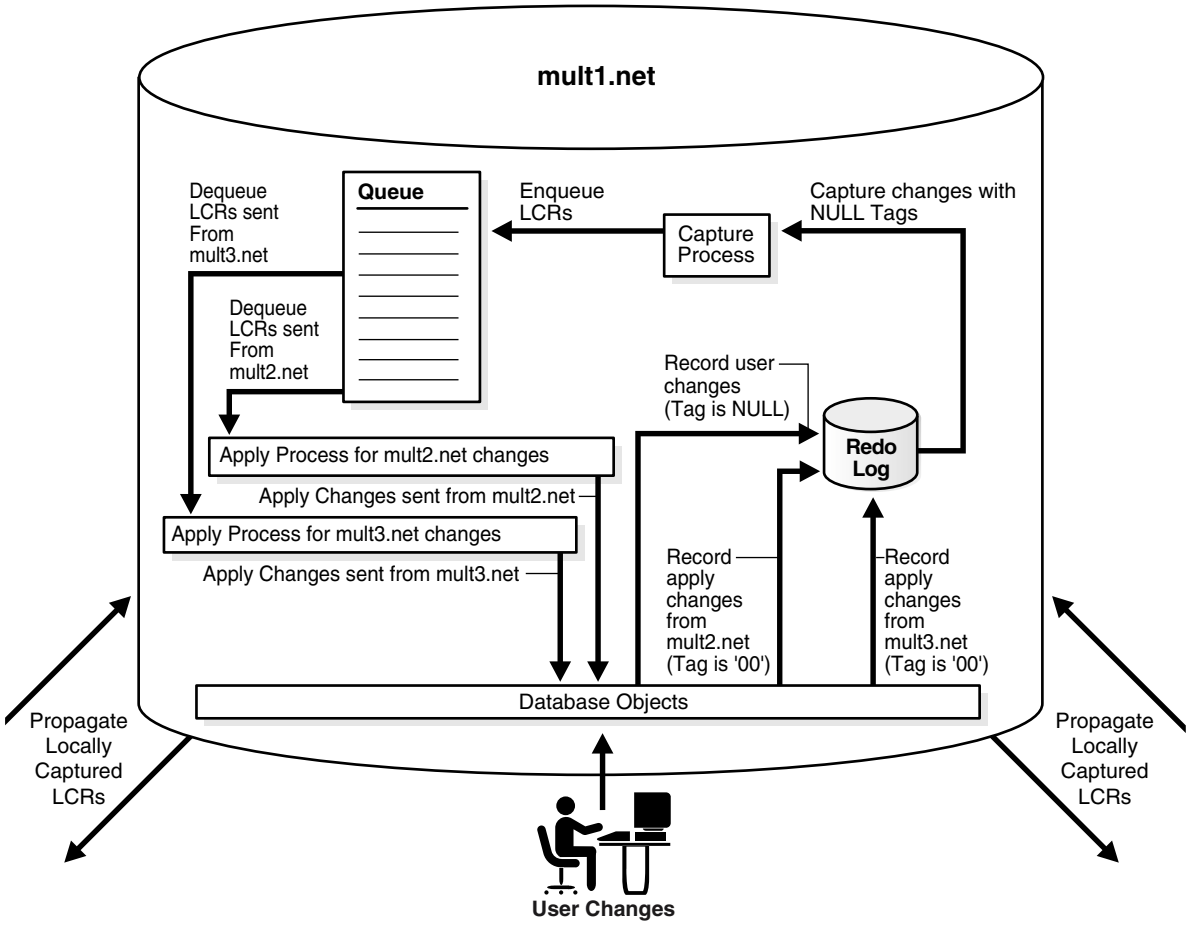
  Each DDL rule should have the following condition:

  ```
  :ddl.is_null_tag()='Y'
  ```

  These rule conditions indicate that the capture process captures a change only if the tag for the change is `NULL`. If you use the `DBMS_STREAMS_ADM` package to generate rules, then each rule has such a condition by default.

This configuration prevents change cycling because all of the changes applied by the apply processes are never recaptured (they were captured originally at the source databases). Each database sends all of its changes to the `hr` schema to every other database. So, in this environment, no changes are lost, and all databases are synchronized. Figure 4–2 illustrates how tags can be used in a database in a multiple-source environment.

Figure 4–2 Tag Use When Each Database Is a Source and Destination Database



See Also: Chapter 16, "Multiple-Source Replication Example" for a detailed illustration of this example

Primary Database Sharing Data with Several Secondary Databases

This scenario involves a Streams environment in which one database is the primary database, and this primary database shares data with several secondary databases. The secondary databases share data only with the primary database. The secondary databases do not share data directly with each other, but, instead, share data indirectly with each other through the primary database. This type of environment is sometimes called a "hub and spoke" environment, with the primary database being the hub and the secondary databases being the spokes.

In such an environment, changes are captured, propagated, and applied in the following way:

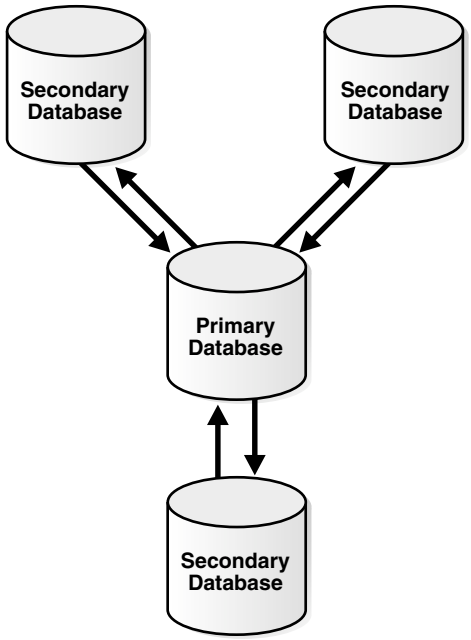
- The primary database captures local changes to the shared data and propagates these changes to all secondary databases, where these changes are applied at each secondary database locally.
- Each secondary database captures local changes to the shared data and propagates these changes to the primary database only, where these changes are applied at the primary database locally.
- The primary database applies changes from each secondary database locally. Next, these changes are captured at the primary database and propagated to all secondary databases, except for the one at which the change originated. Each secondary database applies the changes from the other secondary databases locally, after they have gone through the primary database. This configuration is an example of apply forwarding.

  An alternate scenario might use queue forwarding. If this environment used queue forwarding, then changes from secondary databases that are applied at the primary database are not captured at the primary database. Instead, these changes are forwarded from the queue at the primary database to all secondary databases, except for the one at which the change originated.

  See Also: *Oracle Streams Concepts and Administration* for more information about apply forwarding and queue forwarding

For example, consider an environment that replicates the database objects and data in the `hr` schema between one primary database named `ps1.net` and three secondary databases named `ps2.net`, `ps3.net`, and `ps4.net`. DML and DDL changes made to tables in the `hr` schema are captured at the primary database and at the three secondary databases in the environment. Next, these changes are propagated and applied as described previously. The environment uses apply forwarding, not queue forwarding, to share data between the secondary databases through the primary database. Figure 4–3 illustrates a sample environment which has one primary database and multiple secondary databases.

Figure 4–3 Primary Database Sharing Data with Several Secondary Databases



You can avoid change cycles by configuring the environment in the following way:

- Configure each apply process at the primary database `ps1.net` to generate non-`NULL` redo tags that indicate the site from which it is receiving changes. In this environment, the primary database has at least one apply process for each secondary database from which it receives changes. For example, if an apply process at the primary database receives changes from the `ps2.net` secondary database, then this apply process can generate a raw value that is equivalent to the hexadecimal value `'2'` for all changes it applies. You do this by setting the `apply_tag` parameter in the `CREATE_APPLY` or `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package to the non-`NULL` value.

  For example, run the following procedure to create an apply process that generates redo entries with tags that are equivalent to the hexadecimal value `'2'`:

```
BEGIN
  DBMS_APPLY_ADM.CREATE_APPLY(
    queue_name      => 'strmadmin.streams_queue',
    apply_name      => 'apply_ps2',
    rule_set_name   => 'strmadmin.apply_rules_ps2',
    apply_tag       => HEXTORAW('2'),
    apply_captured  => true);
END;
/
```

- Configure the apply process at each secondary database to generate non-`NULL` redo tags. The exact value of the tags is irrelevant as long as it is non-`NULL`. In this environment, each secondary database has one apply process that applies changes from the primary database.

  If you use a procedure in the `DBMS_STREAMS_ADM` package to create an apply process, then the apply process generates non-`NULL` tags with a value of `'00'` in the redo log by default. In this case, no further action is required for the apply process to generate non-`NULL` tags.

For example, assuming no apply processes exist at the secondary databases, run the ADD_SCHEMA_RULES procedure in the DBMS_STREAMS_ADM package at each secondary database to create an apply process that generates non-NULL redo entries with tags that are equivalent to the hexadecimal value '00':

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name     => 'hr',
    streams_type    => 'apply',
    streams_name    => 'apply',
    queue_name      => 'strmadmin.streams_queue',
    include_dml     => true,
    include_ddl     => true,
    source_database => 'ps1.net',
    inclusion_rule  => true);
END;
/
```

- Configure the capture process at the primary database to capture changes to the shared data regardless of the tags. You do this by setting the include_tagged_lcr parameter to true when you run one of the procedures that generate capture process rules in the DBMS_STREAMS_ADM package. If you use the DBMS_RULE_ADM package to create rules for the capture process at the primary database, then make sure the rules do not contain is_null_tag conditions, because these conditions involve tags in the redo log.

  For example, run the following procedure at the primary database to produce one DML capture process rule and one DDL capture process rule that each have a condition that evaluates to TRUE for changes in the hr schema, regardless of the tag for the change:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name         => 'hr',
    streams_type        => 'capture',
    streams_name        => 'capture',
    queue_name          => 'strmadmin.streams_queue',
    include_tagged_lcr  => true, -- Note parameter setting
    include_dml         => true,
    include_ddl         => true,
    inclusion_rule      => true);
END;
/
```

- Configure the capture process at each secondary database to capture changes only if the tag in the redo entry for the change is NULL. You do this by ensuring that each DML rule in the positive rule set used by the capture process at the secondary database has the following condition:

  ```
  :dml.is_null_tag()='Y'
  ```

  DDL rules should have the following condition:

  ```
  :ddl.is_null_tag()='Y'
  ```

  These rules indicate that the capture process captures a change only if the tag for the change is NULL. If you use the DBMS_STREAMS_ADM package to generate rules, then each rule has one of these conditions by default. If you use the DBMS_RULE_ADM package to create rules for the capture process at a secondary database, then make sure each rule contains one of these conditions.

- Configure one propagation from the queue at the primary database to the queue at each secondary database. Each propagation should use a positive rule set with rules that instruct the propagation to propagate all LCRs in the queue at the primary database to the queue at the secondary database, except for changes that originated at the secondary database.

  For example, if a propagation propagates changes to the secondary database `ps2.net`, whose tags are equivalent to the hexadecimal value `'2'`, then the rules for the propagation should propagate all LCRs relating to the `hr` schema to the secondary database, except for LCRs with a tag of `'2'`. For row LCRs, such rules should include the following condition:

  ```
  :dml.get_tag()!=HEXTORAW('2')
  ```

  For DDL LCRs, such rules should include the following condition:

  ```
  :ddl.get_tag()!=HEXTORAW('2')
  ```

  You can use the `and_condition` parameter in a procedure in the `DBMS_STREAMS_ADM` package to add these conditions to system-created rules, or you can use the `CREATE_RULE` procedure in the `DBMS_RULE_ADM` package to create rules with these conditions. See *Oracle Streams Concepts and Administration* for more information about the `and_condition` parameter.

- Configure one propagation from the queue at each secondary database to the queue at the primary database. A queue at one of the secondary databases contains only local changes made by user sessions and applications at the secondary database, not changes made by an apply process. Therefore, no further configuration is necessary for these propagations.

This configuration prevents change cycling in the following way:

- Changes that originated at a secondary database are never propagated back to that secondary database.
- Changes that originated at the primary database are never propagated back to the primary database.
- All changes made to the shared data at any database in the environment are propagated to every other database in the environment.

So, in this environment, no changes are lost, and all databases are synchronized.

Figure 4–4 illustrates how tags are used at the primary database `ps1.net`.

Figure 4–4 Tags Used at the Primary Database

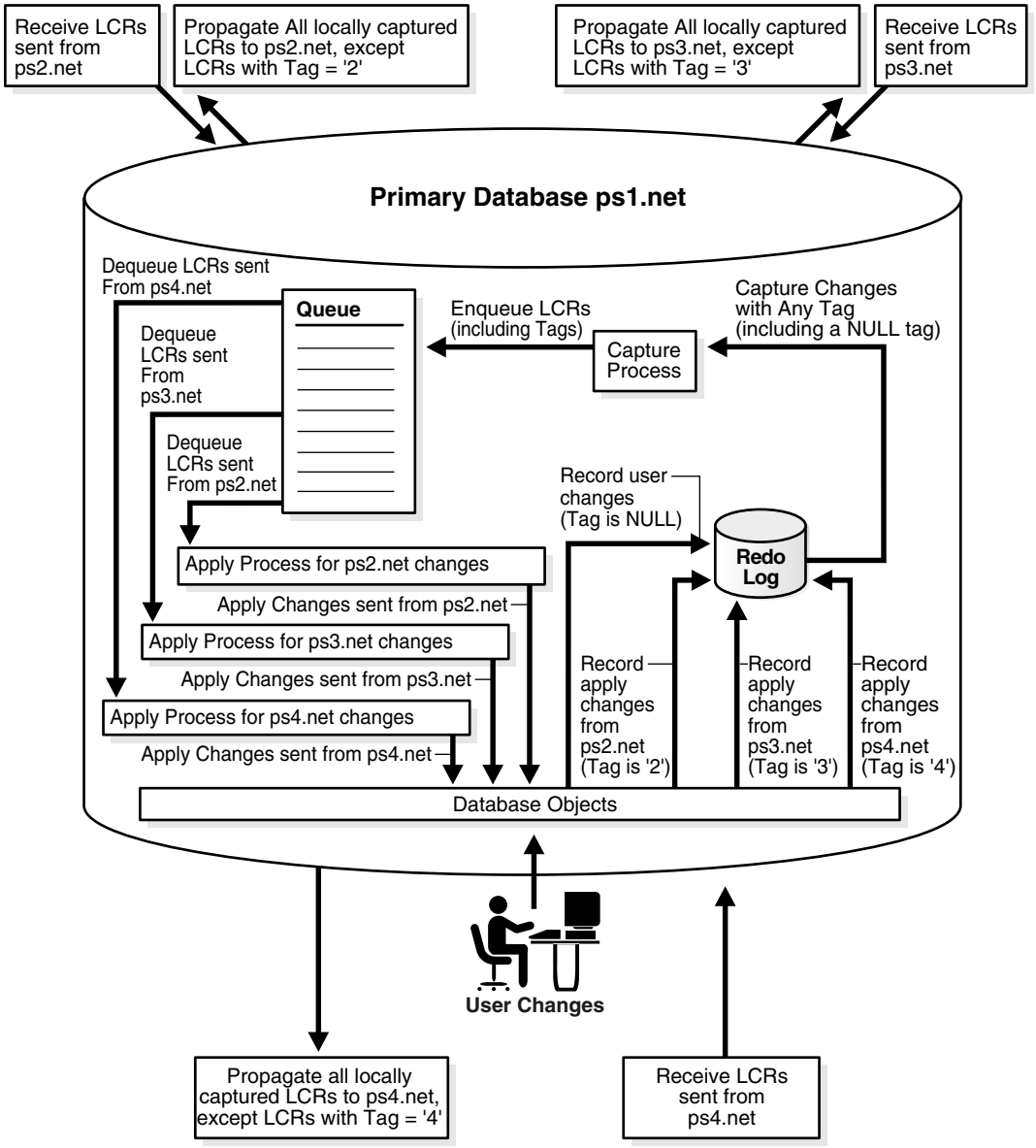
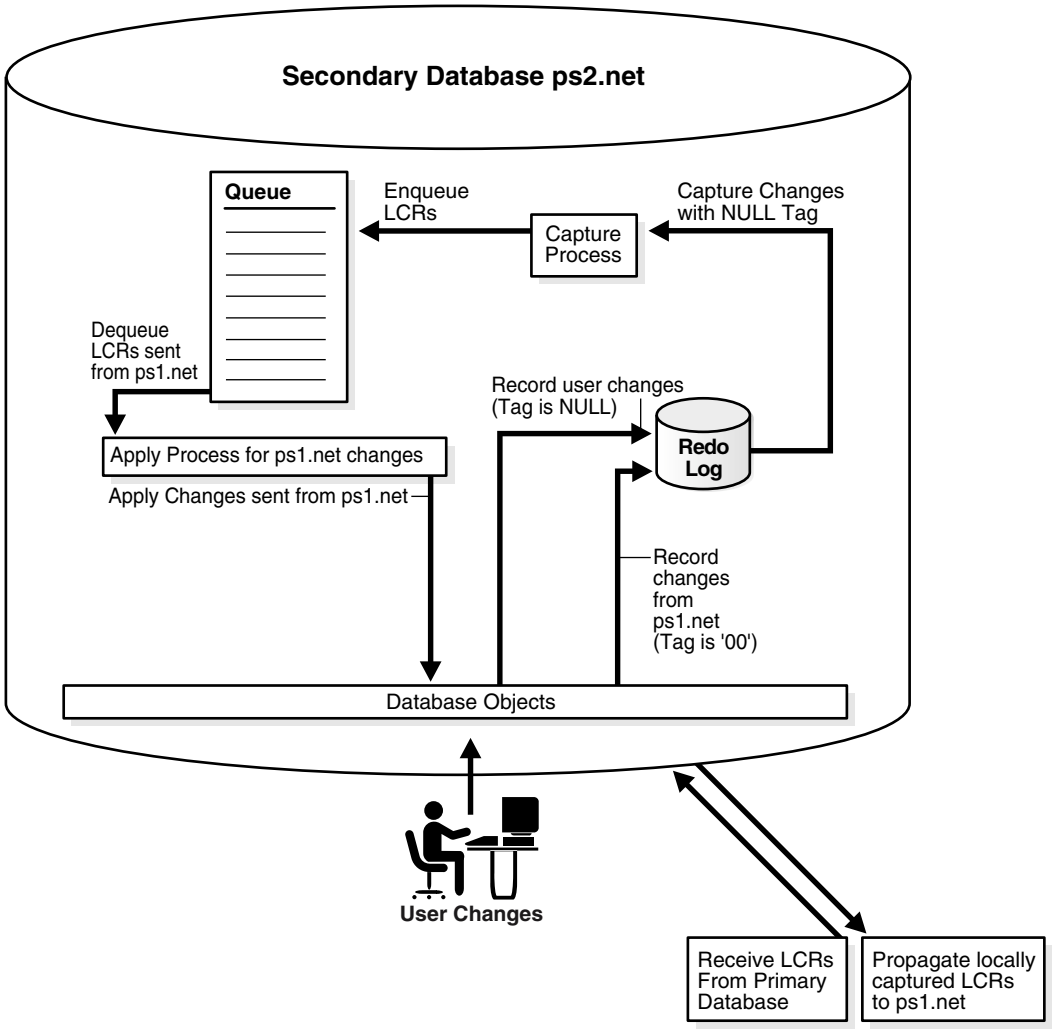


Figure 4–5 illustrates how tags are used at one of the secondary databases (`ps2.net`).

Figure 4–5 Tags Used at a Secondary Database

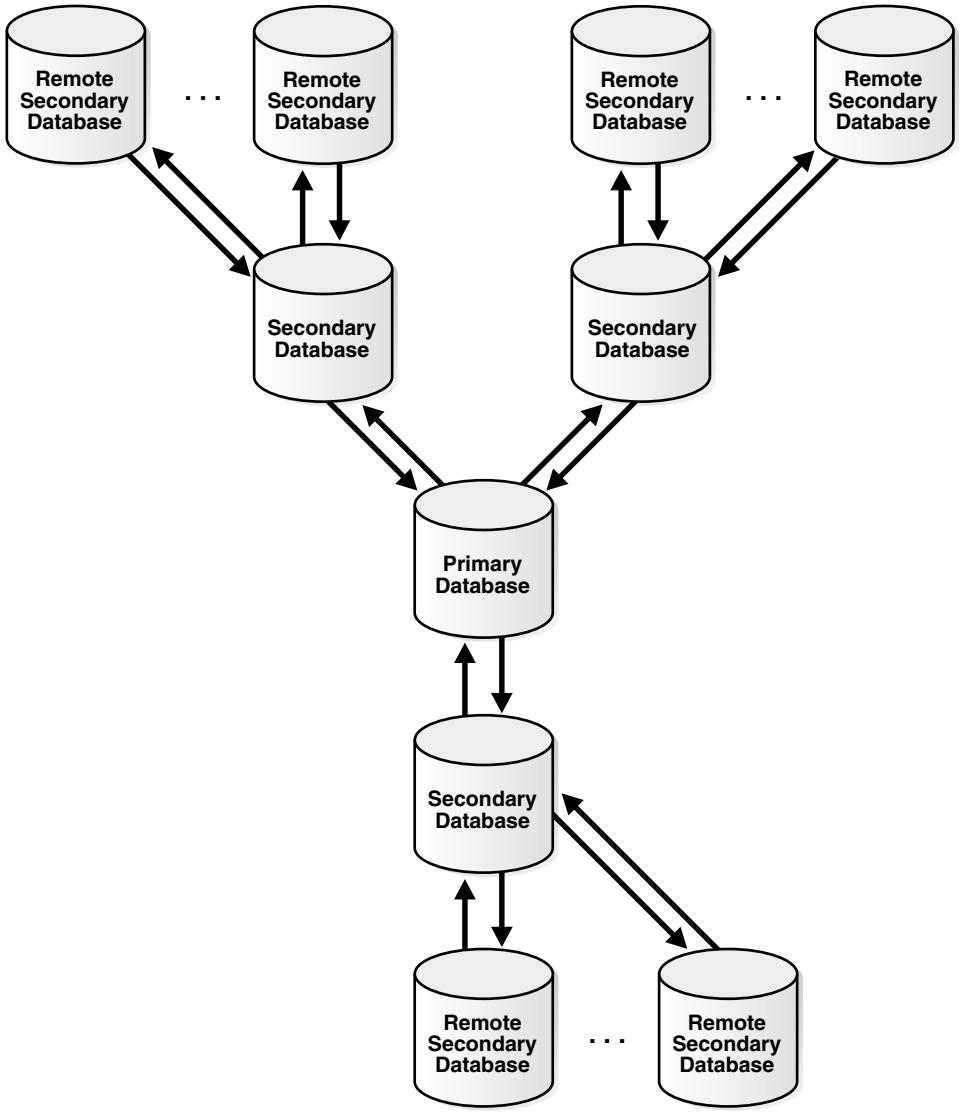


Primary Database Sharing Data with Several Extended Secondary Databases

In this environment, one primary database shares data with several secondary databases, but the secondary databases have other secondary databases connected to them, which will be called *remote secondary* databases. This environment is an extension of the environment described in "Primary Database Sharing Data with Several Secondary Databases" on page 4-9.

A remote secondary database does not share data directly with the primary database, but instead shares data indirectly with the primary database through a secondary database. So, the shared data exists at the primary database, at each secondary database, and at each remote secondary database. Changes made at any of these databases are captured and propagated to all of the other databases. Figure 4–6 illustrates an environment with one primary database and multiple extended secondary databases.

Figure 4–6 Primary Database and Several Extended Secondary Databases



In such an environment, you can avoid change cycling in the following way:

- Configure the primary database in the same way that it is configured in the example described in "Primary Database Sharing Data with Several Secondary Databases" on page 4-9.
- Configure each remote secondary database similar to the way that each secondary database is configured in the example described in "Primary Database Sharing Data with Several Secondary Databases" on page 4-9. The only difference is that the remote secondary databases share data directly with secondary databases, not the primary database.
- At each secondary database, configure one apply process to apply changes from the primary database with a redo tag value that is equivalent to the hexadecimal value `'00'`. This value is the default tag value for an apply process.

- At each secondary database, configure one apply process to apply changes from each of its remote secondary databases with a redo tag value that is unique for the remote secondary database.
- Configure the capture process at each secondary database to capture all changes to the shared data in the redo log, regardless of the tag value for the changes.
- Configure one propagation from the queue at each secondary database to the queue at the primary database. The propagation should use a positive rule set with rules that instruct the propagation to propagate all LCRs in the queue at the secondary database to the queue at the primary database, except for changes that originated at the primary database. You do this by adding a condition to the rules that evaluates to `TRUE` only if the tag in the LCR does not equal `'00'`. For example, enter a condition similar to the following for row LCRs:

  ```
  :dml.get_tag()!=HEXTORAW('00')
  ```

  You can use the `and_condition` parameter in a procedure in the `DBMS_STREAMS_ADM` package to add this condition to system-created rules, or you can use the `CREATE_RULE` procedure in the `DBMS_RULE_ADM` package to create rules with this condition. See *Oracle Streams Concepts and Administration* for more information about the `and_condition` parameter.
- Configure one propagation from the queue at each secondary database to the queue at each remote secondary database. Each propagation should use a positive rule set with rules that instruct the propagation to propagate all LCRs in the queue at the secondary database to the queue at the remote secondary database, except for changes that originated at the remote secondary database. You do this by adding a condition to the rules that evaluates to `TRUE` only if the tag in the LCR does not equal the tag value for the remote secondary database.

  For example, if the tag value of a remote secondary database is equivalent to the hexadecimal value `'19'`, then enter a condition similar to the following for row LCRs:

  ```
  :dml.get_tag()!=HEXTORAW('19')
  ```

  You can use the `and_condition` parameter in a procedure in the `DBMS_STREAMS_ADM` package to add this condition to system-created rules, or you can use the `CREATE_RULE` procedure in the `DBMS_RULE_ADM` package to create rules with this condition. See *Oracle Streams Concepts and Administration* for more information about the `and_condition` parameter.

By configuring the environment in this way, you prevent change cycling, and no changes originating at any database are lost.

5

Streams Heterogeneous Information Sharing

This chapter explains concepts relating to Streams support for information sharing between Oracle databases and non-Oracle databases.

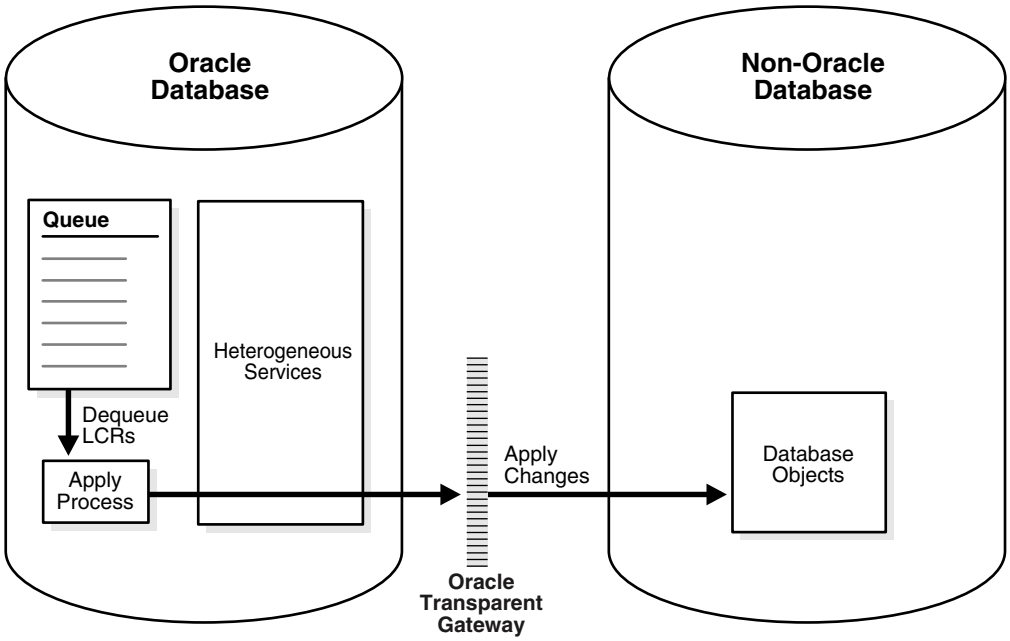
This chapter contains these topics:

- Oracle to Non-Oracle Data Sharing with Streams
- Non-Oracle to Oracle Data Sharing with Streams
- Non-Oracle to Non-Oracle Data Sharing with Streams

Oracle to Non-Oracle Data Sharing with Streams

To share DML changes from an Oracle source database to a non-Oracle destination database, the Oracle database functions as a proxy and carries out some of the steps that would normally be done at the destination database. That is, the LCRs intended for the non-Oracle destination database are dequeued in the Oracle database itself and an apply process at the Oracle database applies the changes to the non-Oracle database across a network connection through an Oracle Transparent Gateway. Figure 5–1 shows an Oracle database sharing data with a non-Oracle database.

Figure 5–1 Oracle to Non-Oracle Heterogeneous Data Sharing



You should configure the Oracle Transparent Gateway to use the transaction model `COMMIT_CONFIRM`.

> **See Also:** Your Oracle-supplied gateway-specific documentation for information about using the transaction model `COMMIT_CONFIRM` for your Oracle Transparent Gateway

Change Capture and Staging in an Oracle to Non-Oracle Environment

In an Oracle to non-Oracle environment, the capture process functions the same way as it would in an Oracle-only environment. That is, it finds changes in the redo log, captures them based on capture process rules, and enqueues the captured changes as logical change records (LCRs) into an `ANYDATA` queue. In addition, a single capture process can capture changes that will be applied at both Oracle and non-Oracle databases.

Similarly, the `ANYDATA` queue that stages the captured LCRs functions the same way as it would in an Oracle-only environment, and you can propagate LCRs to any number of intermediate queues in Oracle databases before they are applied at a non-Oracle database.

> **See Also:**
>
> - *Oracle Streams Concepts and Administration* for general information about capture processes, staging, and propagations
> - Chapter 1, "Understanding Streams Replication" for information about capture processes, staging, and propagations in a Streams replication environment

Change Apply in an Oracle to Non-Oracle Environment

An apply process running in an Oracle database uses Heterogeneous Services and an Oracle Transparent Gateway to apply changes encapsulated in LCRs directly to database objects in a non-Oracle database. The LCRs are not propagated to a queue in the non-Oracle database, as they would be in an Oracle-only Streams environment. Instead, the apply process applies the changes directly through a database link to the non-Oracle database.

See Also:

- *Oracle Streams Concepts and Administration* for general information about apply processes
- "Apply and Streams Replication" on page 1-12 for information about apply processes in a Streams replication environment

Apply Process Configuration in an Oracle to Non-Oracle Environment

This section describes the configuration of an apply process that will apply changes to a non-Oracle database.

Database Link to the Non-Oracle Database When you create an apply process that will apply changes to a non-Oracle database, you previously must have configured Heterogeneous Services, the Oracle Transparent Gateway, and a database link, which will be used by the apply process to apply the changes to the non-Oracle database. The database link must be created with an explicit `CONNECT TO` clause.

When the database link is created and working properly, create the apply process using the `CREATE_APPLY` procedure in the `DBMS_APPLY_ADM` package and specify the database link for the `apply_database_link` parameter. After you create an apply process, you can use apply process rules to specify which changes are applied at the non-Oracle database.

See Also:

- *Oracle Database Heterogeneous Connectivity Administrator's Guide* for more information about Heterogeneous Services and Oracle Transparent Gateways
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the procedures in the `DBMS_APPLY_ADM` package
- *Oracle Streams Concepts and Administration* for information about specifying apply process rules

Substitute Key Columns in an Oracle to Non-Oracle Heterogeneous Environment If you use substitute key columns for any of the tables at the non-Oracle database, then specify the database link to the non-Oracle database when you run the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package.

See Also:

- "Substitute Key Columns" on page 1-19
- "Managing the Substitute Key Columns for a Table" on page 9-11

Parallelism in an Oracle to Non-Oracle Heterogeneous Environment You must set the `parallelism` apply process parameter to `1`, the default setting, when an apply process is applying changes to a non-Oracle database. Currently, parallel apply to non-Oracle databases is not supported. However, you can use multiple apply processes to apply changes a non-Oracle database.

DML Handlers in an Oracle to Non-Oracle Heterogeneous Environment If you use a DML handler to process row LCRs for any of the tables at the non-Oracle database, then specify the database link to the non-Oracle database when you run the `SET_DML_HANDLER` procedure in the `DBMS_APPLY_ADM` package.

> **See Also:** "Managing a DML Handler" on page 9-12 and *Oracle Streams Concepts and Administration* for information about message processing options for an apply process

Message Handlers in an Oracle to Non-Oracle Heterogeneous Environment If you want to use a message handler to process user-enqueued messages for a non-Oracle database, then, when you run the `CREATE_APPLY` procedure in the `DBMS_APPLY_ADM` package, specify the database link to the non-Oracle database using the `apply_database_link` parameter, and specify the message handler procedure using the `message_handler` parameter.

> **See Also:** *Oracle Streams Concepts and Administration* for information about message processing options and managing message handlers

Error and Conflict Handlers in an Oracle to Non-Oracle Heterogeneous Environment Currently, error handlers and conflict handlers are not supported when sharing data from an Oracle database to a non-Oracle database. If an apply error occurs, then the transaction containing the LCR that caused the error is moved into the error queue in the Oracle database.

Datatypes Applied at Non-Oracle Databases

When applying changes to a non-Oracle database, an apply process applies changes made to columns of only the following datatypes:

- `CHAR`
- `VARCHAR2`
- `NCHAR`
- `NVARCHAR2`
- `NUMBER`
- `DATE`
- `RAW`
- `TIMESTAMP`
- `TIMESTAMP WITH TIME ZONE`
- `TIMESTAMP WITH LOCAL TIME ZONE`
- `INTERVAL YEAR TO MONTH`
- `INTERVAL DAY TO SECOND`

The apply process does not apply changes in columns of the following datatypes to non-Oracle databases: CLOB, NCLOB, BLOB, BFILE, LONG, LONG RAW, ROWID, UROWID, and user-defined types (including object types, REFs, varrays, and nested tables). The apply process raises an error when an LCR contains a datatype that is not listed, and the transaction containing the LCR that caused the error is moved to the error queue in the Oracle database.

Each Oracle Transparent Gateway might have further limitations regarding datatypes. For a datatype to be supported in an Oracle to non-Oracle environment, the datatype must be supported by both Streams and the Oracle Transparent Gateway being used.

See Also:

- *Oracle Database SQL Reference* for more information about these datatypes
- Your Oracle-supplied gateway-specific documentation for information about transparent gateways

Types of DML Changes Applied at Non-Oracle Databases

When you specify that DML changes made to certain tables should be applied at a non-Oracle database, an apply process can apply only the following types of DML changes:

- INSERT
- UPDATE
- DELETE

Note: The apply process cannot apply DDL changes at non-Oracle databases.

Instantiation in an Oracle to Non-Oracle Environment

Before you start an apply process that applies changes to a non-Oracle database, complete the following steps to instantiate each table at the non-Oracle database:

1. Use the DBMS_HS_PASSTHROUGH package or the tools supplied with the non-Oracle database to create the table at the non-Oracle database.

   The following is an example that uses the DBMS_HS_PASSTHROUGH package to create the hr.regions table in the het.net non-Oracle database:

```
CONNECT hr/hr

DECLARE
 ret INTEGER;
BEGIN
ret := DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE@het.net (
  'CREATE TABLE regions (region_id INTEGER, region_name VARCHAR(50))');
END;
/
COMMIT;
```

See Also: *Oracle Database Heterogeneous Connectivity Administrator's Guide* and your Oracle supplied gateway-specific documentation for more information about Heterogeneous Services and Oracle Transparent Gateway

2. If the changes that will be shared between the Oracle and non-Oracle database are captured by a capture process at the Oracle database, then prepare all tables that will share data for instantiation.

   See Also: "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

3. Create a PL/SQL procedure (or a C program) that performs the following actions:

   - Gets the current SCN using the `GET_SYSTEM_CHANGE_NUMBER` function in the `DBMS_FLASHBACK` package.
   - Invokes the `ENABLE_AT_SYSTEM_CHANGE_NUMBER` procedure in the `DBMS_FLASHBACK` package to set the current session to the obtained SCN. This action ensures that all fetches are done using the same SCN.
   - Populates the table at the non-Oracle site by fetching row by row from the table at the Oracle database and then inserting row by row into the table at the non-Oracle database. All fetches should be done at the SCN obtained using the `GET_SYSTEM_CHANGE_NUMBER` function.

   For example, the following PL/SQL procedure gets the flashback SCN, fetches each row in the `hr.regions` table in the current Oracle database, and inserts them into the `hr.regions` table in the `het.net` non-Oracle database. Notice that flashback is disabled before the rows are inserted into the non-Oracle database.

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE insert_reg IS
  CURSOR c1 IS
    SELECT region_id, region_name FROM hr.regions;
  c1_rec c1 % ROWTYPE;
  scn NUMBER;
BEGIN
  scn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();
  DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(
     query_scn  =>  scn);
  /* Open c1 in flashback mode */
  OPEN c1;
  /* Disable Flashback */
  DBMS_FLASHBACK.DISABLE;
  LOOP
    FETCH c1 INTO c1_rec;
    EXIT WHEN c1%NOTFOUND;
   /*
     Note that all the DML operations inside the loop are performed
     with Flashback disabled
   */
   INSERT INTO hr.regions@het.net VALUES (
     c1_rec.region_id,
     c1_rec.region_name);
  END LOOP;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('SCN = ' || scn);
  EXCEPTION WHEN OTHERS THEN
    DBMS_FLASHBACK.DISABLE;
    RAISE;
END;
/
```

   Make a note of the SCN returned.

If the Oracle Transparent Gateway you are using supports the Heterogeneous Services callback functionality, then you can replace the loop in the previous example with the following SQL statement:

```
INSERT INTO hr.region@het.net SELECT * FROM hr.region@!;
```

Note: The user who creates and runs the procedure in the previous example must have `EXECUTE` privilege on the `DBMS_FLASHBACK` package and all privileges on the tables involved.

See Also: *Oracle Database Heterogeneous Connectivity Administrator's Guide* and your Oracle-supplied gateway-specific documentation for information about callback functionality and your Oracle Transparent Gateway

4. Set the instantiation SCN for the table at the non-Oracle database. Specify the SCN you obtained in Step 3 in the `SET_TABLE_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package to instruct the apply process to skip all LCRs with changes that occurred before the SCN you obtained in Step 3. Make sure you set the `apply_database_link` parameter to the database link for the remote non-Oracle database.

   See Also: "Setting Instantiation SCNs at a Destination Database" on page 10-27 and *Oracle Database PL/SQL Packages and Types Reference* for more information about the `SET_TABLE_INSTANTIATION_SCN` procedure

Transformations in an Oracle to Non-Oracle Environment

In an Oracle to non-Oracle environment, you can specify rule-based transformations during capture or apply the same way as you would in an Oracle-only environment. In addition, if your environment propagates LCRs to one or more intermediate Oracle databases before they are applied at a non-Oracle database, then you can specify a rule-based transformation during propagation from a queue at an Oracle database to another queue at an Oracle database.

See Also: *Oracle Streams Concepts and Administration* for more information about rule-based transformations

Messaging Gateway and Streams

Messaging Gateway is a feature of the Oracle database that provides propagation between Oracle queues and non-Oracle message queuing systems. Messages enqueued into an Oracle queue are automatically propagated to a non-Oracle queue, and the messages enqueued into a non-Oracle queue are automatically propagated to an Oracle queue. It provides guaranteed message delivery to the non-Oracle messaging system and supports the native message format for the non-Oracle messaging system. It also supports specification of user-defined transformations that are invoked while propagating from an Oracle queue to the non-Oracle messaging system or from the non-Oracle messaging system to an Oracle queue.

See Also: *Oracle Streams Advanced Queuing User's Guide and Reference* for more information about the Messaging Gateway

Error Handling in an Oracle to Non-Oracle Environment

If the apply process encounters an unhandled error when it tries to apply an LCR at a non-Oracle database, then the transaction containing the LCR is placed in the error queue in the Oracle database that is running the apply process. The apply process detects data conflicts in the same way as it does in an Oracle-only environment, but automatic conflict resolution is not supported currently in an Oracle to non-Oracle environment. Therefore, any data conflicts encountered are treated as apply errors.

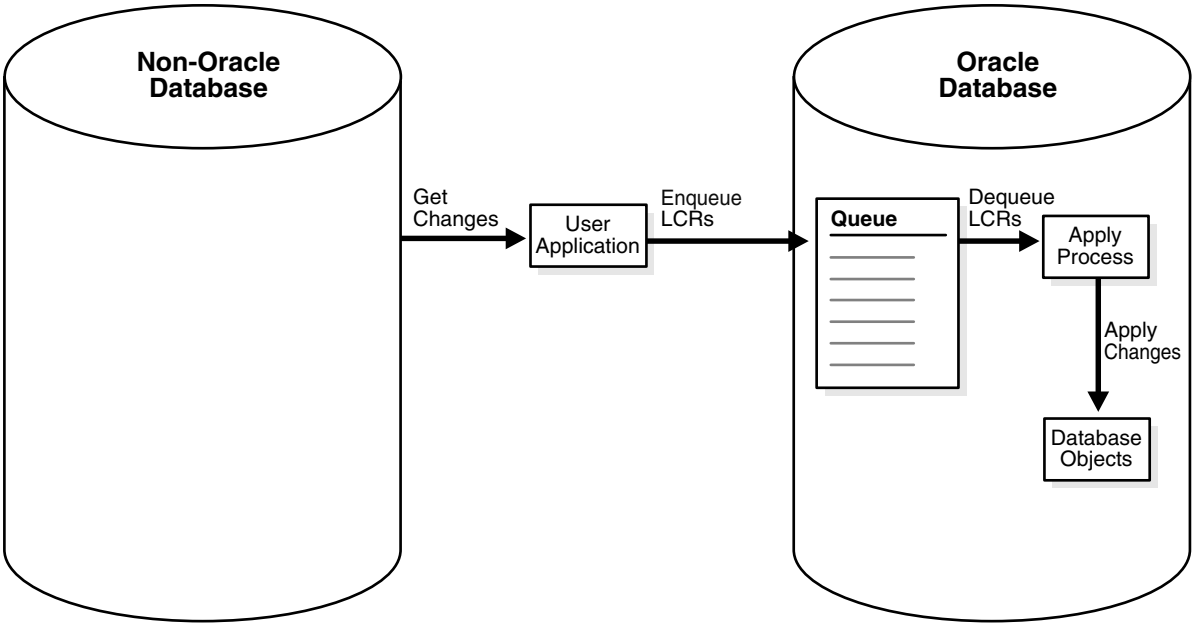
Example Oracle to Non-Oracle Streams Environment

Chapter 15, "Single-Source Heterogeneous Replication Example" contains a detailed example that includes sharing data in an Oracle to non-Oracle Streams environment.

Non-Oracle to Oracle Data Sharing with Streams

To capture and propagate changes from a non-Oracle database to an Oracle database, a custom application is required. This application gets the changes made to the non-Oracle database by reading from transaction logs, by using triggers, or by some other method. The application must assemble and order the transactions and must convert each change into a logical change record (LCR). Next, the application must enqueue the LCRs into a queue in an Oracle database using the `DBMS_STREAMS_MESSAGING` package or the `DBMS_AQ` package. The application must commit after enqueuing all LCRs in each transaction. Figure 5–2 shows a non-Oracle databases sharing data with an Oracle database.

Figure 5–2 Non-Oracle to Oracle Heterogeneous Data Sharing



Change Capture in a Non-Oracle to Oracle Environment

Because the custom user application is responsible for assembling changes at the non-Oracle database into LCRs and enqueuing the LCRs into a queue at the Oracle database, the application is completely responsible for change capture. This means that the application must construct LCRs that represent changes at the non-Oracle database and then enqueue these LCRs into the queue at the Oracle database. The application can enqueue multiple transactions concurrently, but the transactions must be committed in the same order as the transactions on the non-Oracle source database.

> **See Also:** "Constructing and Enqueuing LCRs" on page 11-2 for more information about constructing and enqueuing LCRs

Staging in a Non-Oracle to Oracle Environment

If you want to ensure the same transactional consistency at both the Oracle database where changes are applied and the non-Oracle database where changes originate, then you must use a transactional queue to stage the LCRs at the Oracle database. For example, suppose a single transaction contains three row changes, and the custom application enqueues three row LCRs, one for each change, and then commits. With a transactional queue, a commit is performed by the apply process after the third row LCR, retaining the consistency of the transaction. If you use a nontransactional queue, then a commit is performed for each row LCR by the apply process. The `SET_UP_QUEUE` procedure in the `DBMS_STREAMS_ADM` package creates a transactional queue automatically.

Also, the queue at the Oracle database should be a commit-time queue. A commit-time queue orders LCRs by approximate commit system change number (approximate CSCN) of the transaction that includes the LCRs. Commit-time queues preserve transactional dependency ordering between LCRs in the queue, assuming that the application that enqueued the LCRs commit transactions in the correct order. Also, commit-time queues ensure consistent browses of LCRs in a queue.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about transactional queues and commit-time queues

Change Apply in a Non-Oracle to Oracle Environment

In a non-Oracle to Oracle environment, the apply process functions the same way as it would in an Oracle-only environment. That is, it dequeues each LCR from its associated queue based on apply process rules, performs any rule-based transformation, and either sends the LCR to a handler or applies it directly. Error handling and conflict resolution also function the same as they would in an Oracle-only environment. So, you can specify a prebuilt update conflict handler or create a custom conflict handler to resolve conflicts.

The apply process should be configured to apply user-enqueued LCRs, not captured LCRs. So, the apply process should be created using the `CREATE_APPLY` procedure in the `DBMS_APPLY_ADM` package, and the `apply_captured` parameter should be set to `false` when you run this procedure. After the apply process is created, you can use procedures in the `DBMS_STREAMS_ADM` package to add rules for LCRs to the apply process rule sets.

See Also:

- *Oracle Streams Concepts and Administration* for more information about apply processes, rules, and rule-based transformations
- Chapter 3, "Streams Conflict Resolution"

Instantiation from a Non-Oracle Database to an Oracle Database

There is no automatic way to instantiate tables that exist at a non-Oracle database at an Oracle database. However, you can perform the following general procedure to instantiate a table manually:

1. At the non-Oracle database, use a non-Oracle utility to export the table to a flat file.
2. At the Oracle database, create an empty table that matches the table at the non-Oracle database.
3. At the Oracle database, use SQL*Loader to load the contents of the flat file into the table.

See Also: *Oracle Database Utilities* for information about using SQL*Loader

Non-Oracle to Non-Oracle Data Sharing with Streams

Streams supports data sharing between two non-Oracle databases through a combination of non-Oracle to Oracle data sharing and Oracle to non-Oracle data sharing. Such an environment would use Streams in an Oracle database as an intermediate database between two non-Oracle databases.

For example, a non-Oracle to non-Oracle environment can consist of the following databases:

- A non-Oracle database named `het1.net`
- An Oracle database named `dbs1.net`
- A non-Oracle database named `het2.net`

A user application assembles changes at `het1.net` and enqueues them into a queue in `dbs1.net`. Next, the apply process at `dbs1.net` applies the changes to `het2.net` using Heterogeneous Services and an Oracle Transparent Gateway. Another apply process at `dbs1.net` could apply some or all of the changes in the queue locally at `dbs1.net`. One or more propagations at `dbs1.net` could propagate some or all of the changes in the queue to other Oracle databases.

Part II

Configuring Streams Replication

This part describes configuring a Streams replication and contains the following chapters:

6

Simple Streams Replication Configuration

This chapter describes simple methods for configuring Streams replication between two databases.

This chapter contains these topics:

- Configuring Replication Using a Streams Wizard in Enterprise Manager
- Configuring Replication Using the DBMS_STREAMS_ADM Package

Configuring Replication Using a Streams Wizard in Enterprise Manager

The Streams tool in Enterprise Manager includes two wizards that configure a Streams replication environment. The following sections describe the wizards and how to open them:

- Streams Global, Schema, Table, and Subset Replication Wizard
- Streams Tablespace Replication Wizard
- Opening a Streams Replication Configuration Wizard

Streams Global, Schema, Table, and Subset Replication Wizard

The **Steams Global, Schema, Table, and Subset Replication** wizard can configure a Streams environment that replicates changes to the entire source database, certain schemas in the source database, certain tables in the source database, or subsets of tables in the source database.

This wizard can configure a Streams environment that maintains DML changes, DDL changes, or both. The database objects configured for replication by this wizard can be in multiple tablespaces in your source database. This wizard only configures a single-source replication environment. It cannot configure a bi-directional replication environment.

You can run this wizard in Database Control or Grid Control. To run this wizard in Database Control, meet the following requirements:

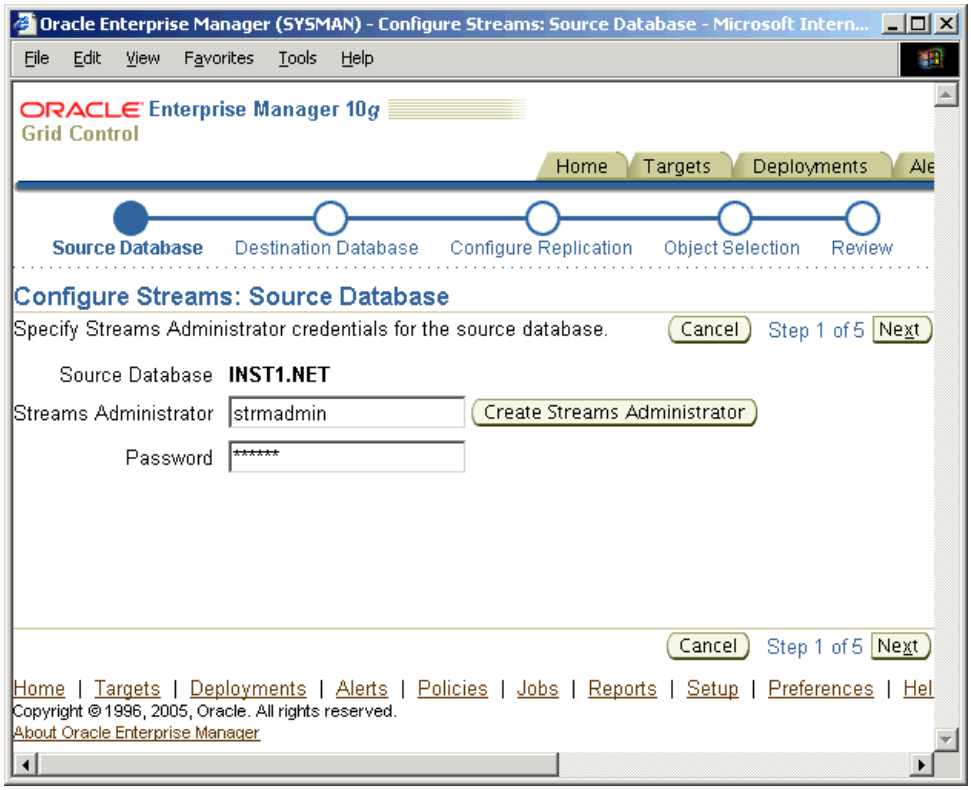
- This wizard, or the scripts generated by this wizard, must be run at an Oracle Database 10*g* Release 2 database.
- The destination database configured by this wizard must be an Oracle Database 10*g* Release 1 or later database.

To run this wizard in Grid Control, meet the following requirements:

- This wizard, or the scripts generated by this wizard, must be run at an Oracle9*i* Release 2 (9.2) or later database.
- The destination database configured by this wizard must be an Oracle9*i* Release 2 (9.2) or later database.

Figure 6–1 shows the opening page of the **Steams Global, Schema, Table, and Subset Replication** wizard.

Figure 6–1 Streams Global, Schema, Table, and Subset Replication Wizard



Streams Tablespace Replication Wizard

The **Streams Tablespace Replication** wizard can configure a Streams environment that replicates changes to all of the database objects in a particular self-contained tablespace or in a set of self-contained tablespaces. A self-contained tablespace has no references from the tablespace pointing outside of the tablespace. For example, if an index in the tablespace is for a table in a different tablespace, then the tablespace is not self-contained. When there is more than one tablespace in a tablespace set, a self-contained tablespace set has no references from inside the set of tablespaces pointing outside of the set of tablespaces.

This wizard can configure a single-source replication environment or a bi-directional replication environment. This wizard does not configure the Streams environment to maintain DDL changes to the tablespace set nor to the database objects in the tablespace set. For example, the Streams environment is not configured to replicate `ALTER TABLESPACE` statements on the tablespace, nor is it configured to replicate `ALTER TABLE` statements on tables in the tablespace.

You can run this wizard in Database Control or Grid Control. To run this wizard in Database Control, meet the following requirements:

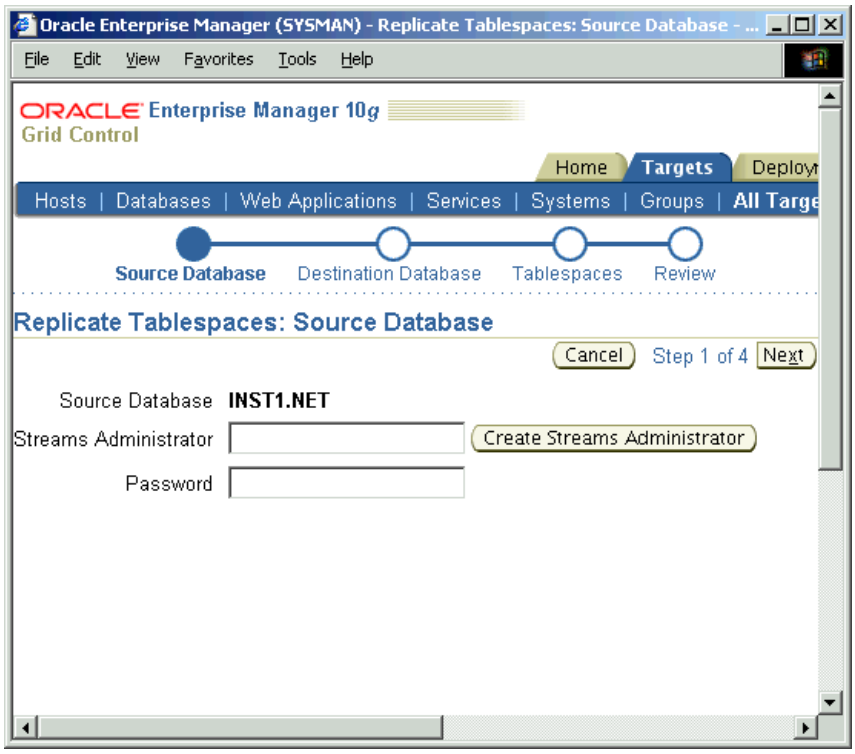
- This wizard, or the scripts generated by this wizard, must be run at an Oracle Database 10*g* Release 2 database.
- If this wizard configures the replication environment directly (not with scripts), then both databases must be Oracle Database 10*g* Release 2 databases.
- If the replication environment is configured with scripts generated by this wizard, then the destination database must be an Oracle Database 10*g* Release 1 or later database. If the script configures an Oracle Database 10*g* Release 1 database, then the script must be modified so that it does not configure features that are available only in Oracle Database 10*g* Release 2, such as queue-to-queue propagation.

To run this wizard in Grid Control, meet the following requirements:

- Each database configured by this wizard, or the scripts generated by this wizard, must be Oracle Database 10*g* Release 1 or later database.
- This wizard, or the scripts generated by this wizard, must be run at an Oracle Database 10*g* Release 1 or later database.
- If this wizard is run at an Oracle Database 10*g* Release 2 database, and the wizard configures the replication environment directly (not with scripts), then both databases must be Oracle Database 10*g* Release 2 databases.
- If this wizard is run at an Oracle Database 10*g* Release 2 database, and the replication environment is configured with generated scripts, then the destination database must be an Oracle Database 10*g* Release 1 or later database. If the script configures an Oracle Database 10*g* Release 1 database, then the script must be modified so that it does not configure features that are available only in Oracle Database 10*g* Release 2, such as queue-to-queue propagation.

Figure 6–2 shows the opening page of the **Streams Tablespace Replication** wizard.

Figure 6–2 Streams Tablespace Replication Wizard



Opening a Streams Replication Configuration Wizard

Both wizards configure, or produce scripts to configure, a Streams replication environment. A capture process is configured to capture changes to the database objects in the specified tablespaces at the source database. A propagation is configured at the source database to propagate each change in the form of a logical change record (LCR) from the source database to the destination database. An apply process at the destination database applies the LCRs to make the changes at the destination database. If you use the Streams Tablespace Replication Wizard to configure a bi-directional replication environment, then each database captures changes and propagates them to the other database, and each database applies changes from the other database. Both wizards also perform an instantiation of the specified database objects.

To open one of these wizards, complete the following steps in Enterprise Manager:

1. Navigate to the Database Home page of a database that will be a source database in the replication environment.
2. Select the **Maintenance** tab.
3. Click **Setup** in the **Streams** section.
4. Click the wizard you want to use in the **Setup Options** list. Click **Help** for more information.

Note:

- Any source database that generates redo data that will be captured by a capture process must run in `ARCHIVELOG` mode.
- You might need to configure conflict resolution if bi-directional replication is configured.

See Also:

- Chapter 1, "Understanding Streams Replication"
- Chapter 2, "Instantiation and Streams Replication"
- Chapter 3, "Streams Conflict Resolution"
- *Oracle Database Administrator's Guide* for information about running a database in `ARCHIVELOG` mode

Configuring Replication Using the DBMS_STREAMS_ADM Package

The following procedures in the `DBMS_STREAMS_ADM` package configure a replication environment that is maintained by Streams:

- `MAINTAIN_GLOBAL` configures a Streams environment that replicates changes at the database level between two databases.
- `MAINTAIN_SCHEMAS` configures a Streams environment that replicates changes to specified schemas between two databases.
- `MAINTAIN_SIMPLE_TTS` clones a simple tablespace from a source database at a destination database and uses Streams to maintain this tablespace at both databases.
- `MAINTAIN_TABLES` configures a Streams environment that replicates changes to specified tables between two databases.
- `MAINTAIN_TTS` clones a set of tablespaces from a source database at a destination database and uses Streams to maintain these tablespaces at both databases.
- `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` configure a Streams environment that replicates changes either at the database level or to specified tablespaces between two databases. These procedures must be used together, and instantiation actions must be performed manually, to complete the Streams replication configuration.

The following sections contain instructions for preparing to run one of these procedures and examples that illustrate common scenarios:

- Preparing to Configure Streams Replication Using the DBMS_STREAMS_ADM Package
- Configuring Database Replication Using the DBMS_STREAMS_ADM Package
- Configuring Tablespace Replication Using the DBMS_STREAMS_ADM Package
- Configuring Schema Replication Using the DBMS_STREAMS_ADM Package
- Configuring Table Replication Using the DBMS_STREAMS_ADM Package

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about these procedures

Preparing to Configure Streams Replication Using the DBMS_STREAMS_ADM Package

The following sections describe decisions to make and actions to complete before configuring replication with a procedure in the DBMS_STREAMS_ADM package:

- Decisions to Make Before Configuring Streams Replication
- Tasks to Complete Before Configuring Streams Replication

Decisions to Make Before Configuring Streams Replication

Make the following decisions before configuring Streams replication:

- Decide Whether to Maintain DDL Changes
- Decide Whether to Configure Local or Downstream Capture for the Source Database
- Decide Whether Replication Is Bi-Directional
- Decide Whether to Configure Replication Directly or Generate a Script
- Decide How to Perform Instantiation

Decide Whether to Maintain DDL Changes These procedures configure the replication environment to maintain data manipulation language (DML) changes to the specified database object by default. DML changes include INSERT, UPDATE, DELETE, and LOB update operations. You must decide whether you want the replication environment to maintain data definition language (DDL) changes as well. Examples of statements that result in DDL changes are CREATE TABLE, ALTER TABLE, ALTER TABLESPACE, and ALTER DATABASE.

Some Streams replication environments assume that the database objects are the same at each database. In this case, maintaining DDL changes with Streams makes it easy to keep the shared database objects synchronized. However, some Streams replication environments require that shared database objects are different at different databases. For example, a table can have a different name or shape at two different databases. In these environments, rule-based transformations and apply handlers can modify changes so that they can be shared between databases, and you might not want to maintain DDL changes with Streams.

The include_ddl parameter controls whether the procedure configures Streams replication to maintain DDL changes:

- To configure a Streams replication environment that does not maintain DDL changes, set the include_ddl parameter to false when you run one of these procedures. The default value for this parameter is false.
- To configure a Streams replication environment that maintains DDL changes, set the include_ddl parameter to true when you run one of these procedures.

Note: The MAINTAIN_SIMPLE_TTS procedure does not include the include_ddl parameter. A Streams replication environment configured by the MAINTAIN_SIMPLE_TTS procedure only maintains DML changes.

See Also:

- "Nonidentical Replicas with Streams" on page 1-3
- *Oracle Streams Concepts and Administration* for more information about rule-based transformations
- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers

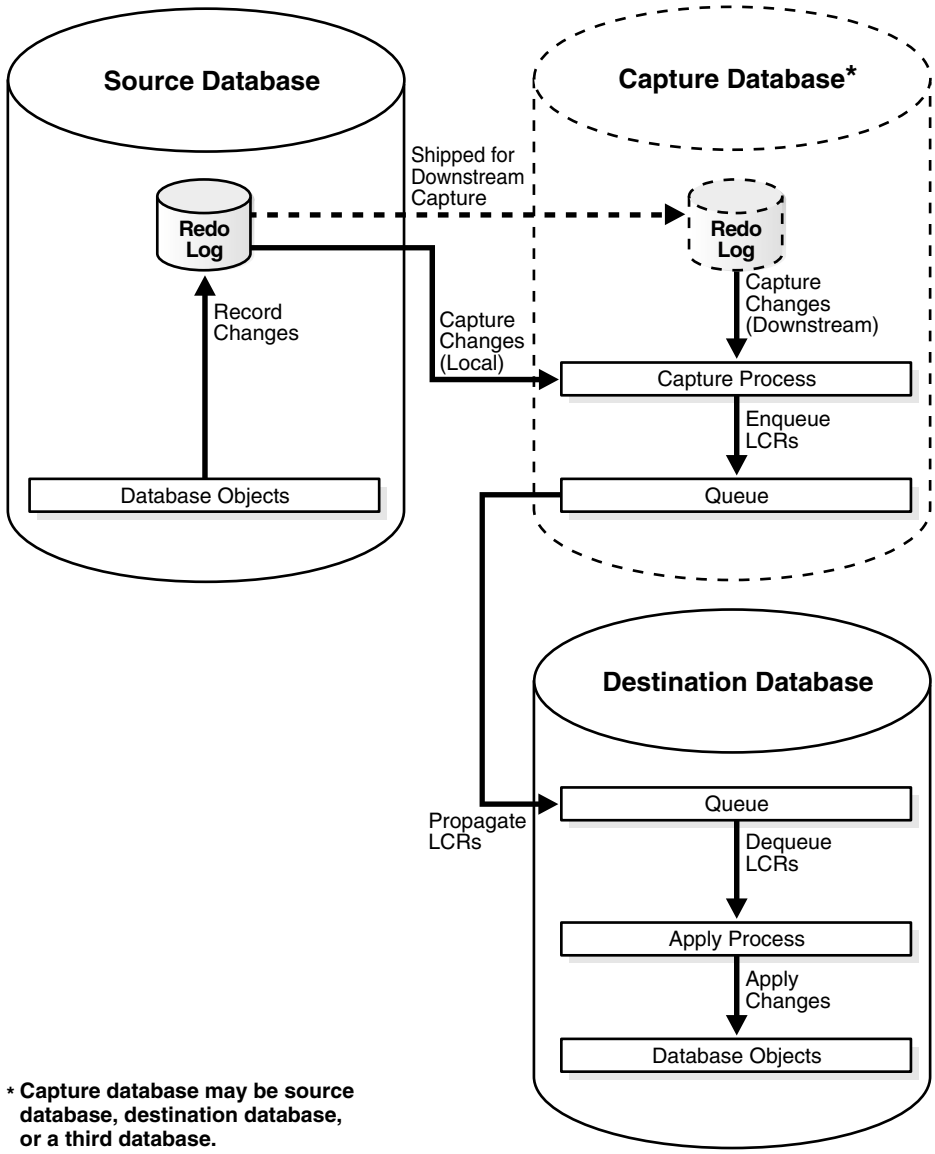
Decide Whether to Configure Local or Downstream Capture for the Source Database Local capture means that a capture process runs on the source database. Downstream capture means that a capture process runs on a database other than the source database. These procedures can either configure local capture or downstream capture for the database specified in the `source_database` parameter.

The database that captures changes made to the source database is called the **capture database**. These procedures can configure one of the following databases as the capture database:

- Source database (local capture)
- Destination database (downstream capture)
- A third database (downstream capture)

Figure 6–3 shows the role of the capture database.

Figure 6–3 The Capture Database



The database on which the procedure is run is configured as the capture database for changes made to the source database. Therefore, to configure local capture at the source database, run the procedure at the source database. To configure downstream capture at the destination database or a third database, run the procedure at the destination database or third database.

If the source database or a third database is the capture database, then these procedures configure a propagation to propagate changes from the capture database to the destination database. If the destination database is the capture database, then this propagation between databases is not needed.

Also, the `capture_name` and `capture_queue_name` parameters must be set to `NULL` when both of the following conditions are met:

- The destination database is the capture database.
- The `bi_directional` parameter is set to `true`.

When both of these conditions are met, these procedure configure two capture processes at the destination database, and these capture processes must have different names. When the `capture_name` and `capture_queue_name` parameters are set to `NULL`, the system generates a different name for the capture processes. These procedures raise an error if both conditions are met and either the `capture_name` parameter or the `capture_queue_name` parameter is set to a non-`NULL` value.

Note:

- When these procedures configure downstream capture, they always configure archived-log downstream capture. These procedures do not configure real-time downstream capture. However, the scripts generated by these procedures can be modified to configure real-time downstream capture.
- If these procedures configure bi-directional replication, then the capture process for the destination database always is a local capture process. That is, these procedures always configure the capture process for changes made to the destination database to run on the destination database.

See Also:

- *Oracle Streams Concepts and Administration* for information about local capture and downstream capture
- "Decide Whether Replication Is Bi-Directional" on page 6-9

Decide Whether Replication Is Bi-Directional These procedures set up either a single-source Streams configuration with the database specified in the `source_database` parameter as the source database, or a bi-directional Streams configuration with both databases acting as source and destination databases. The `bi_directional` parameter in each procedure controls whether the Streams configuration is single source or bi-directional.

- If the `bi_directional` parameter is `false`, then a capture process captures changes made to the source database and an apply process at the destination database applies these changes. If the destination database is not the capture database, then a propagation propagates the captured changes to the destination database. The default value for this parameter is `false`.
- If the `bi_directional` parameter is `true`, then a separate capture process captures changes made to each database, propagations propagate these changes to the other database, and each database applies changes from the other database.

When a replication environment is not bi-directional, and no changes are allowed at the destination database, Streams keeps the shared database objects synchronized at the databases. However, when a replication environment is not bi-directional, and independent changes are allowed at the destination database, the shared database objects might diverge between the databases. Independent changes can be made by users, by applications, or by replication with a third database.

These procedures cannot be used to configure multi-directional replication where changes can be cycled back to a source database by a third database in the environment. For example, these procedures cannot be used to configure a Streams replication environment with three databases where each database shares changes with the other two databases in the environment. If these procedures are used to configure a three way replication environment such as this, then changes made at a

source database would be cycled back to the same source database. In a valid three way replication environment, a particular change is made only once at each database. However, you can add additional databases to the Streams environment after using one of these procedures to configure the environment, and these procedures can be used to configure a "hub and spoke" replication environment.

Note:

- If you set the `bi_directional` parameter to `true` when you run one of these procedures, then do not allow data manipulation language (DML) or data definition language (DDL) changes to the shared database objects at the destination database while the procedure, or the script generated by the procedure, is running. This restriction does not apply if a procedure is configuring a single-source replication environment.
- You might need to configure conflict resolution if bi-directional replication is configured.

See Also:

- "Decide Whether to Configure Local or Downstream Capture for the Source Database" on page 6-7
- Chapter 8, "Adding to a Streams Replication Environment"
- "Primary Database Sharing Data with Several Secondary Databases" on page 4-9 for more information about "hub and spoke" replication environments
- Chapter 3, "Streams Conflict Resolution"

Decide Whether to Configure Replication Directly or Generate a Script These procedures can configure the Streams replication environment directly, or they can generate a script that configures the environment. Using a procedure to configure replication directly is simpler than running a script, and the environment is configured immediately. However, you might choose to generate a script for the following reasons:

- You want to review the actions performed by the procedure before configuring the environment.
- You want to modify the script to customize the configuration.

For example, you might want an apply process to use apply handlers for customized processing of the changes to certain tables before applying these changes. In this case, you can use the procedure to generate a script and modify the script to add the apply handlers.

You also might want to maintain DML changes for a number of tables, but you might want to maintain DDL changes for a subset of these tables. In this case, you can generate a script by running the `MAINTAIN_TABLES` procedure with the `include_ddl` parameter set to `false`. You can modify the script to maintain DDL changes for the appropriate tables.

The `perform_actions` parameter controls whether the procedure configures the replication environment directly:

- To configure a Streams replication environment directly when you run one of these procedures, set the `perform_actions` parameter to `true`. The default value for this parameter is `true`.

- To generate a configuration script when you run one of these procedures, set the `perform_actions` parameter to `false`, and use the `script_name` and `script_directory_object` parameters to specify the name and location of the configuration script.

Decide How to Perform Instantiation The `MAINTAIN_GLOBAL`, `MAINTAIN_SCHEMAS`, and `MAINTAIN_TABLES` procedures provide options for instantiation. Instantiation is the process of preparing database objects for instantiation at a source database, optionally copying the database objects from a source database to a destination database, and setting the instantiation SCN for each instantiated database object.

When you run one of these three procedures, you can choose to perform the instantiation in one of the following ways:

- **Data Pump Export Dump File Instantiation:** This option performs a Data Pump export of the shared database objects at the source database and a Data Pump import of the export dump file at the destination database. The instantiation SCN is set for each shared database object during import.

  To specify this instantiation option, set the `instantiation` parameter to one of the following values:

  - `DBMS_STREAMS_ADM.INSTANTIATION_FULL` if you run the `MAINTAIN_GLOBAL` procedure
  - `DBMS_STREAMS_ADM.INSTANTIATION_SCHEMA` if you run the `MAINTAIN_SCHEMAS` procedure
  - `DBMS_STREAMS_ADM.INSTANTIATION_TABLE` if you run the `MAINTAIN_TABLES` procedure

  If the `bi_directional` parameter is set to `true`, then the procedure also sets the instantiation SCN for each shared database object at the source database.

- **Data Pump Network Import Instantiation:** This option performs a network Data Pump import of the shared database objects. A network import means that Data Pump performs the import without using an export dump file. Therefore, directory objects do not need to be created for instantiation purposes when you use this option. The instantiation SCN is set for each shared database object during import.

  To specify this instantiation option, set the `instantiation` parameter to one of the following values:

  - `DBMS_STREAMS_ADM.INSTANTIATION_FULL_NETWORK` if you run the `MAINTAIN_GLOBAL` procedure
  - `DBMS_STREAMS_ADM.INSTANTIATION_SCHEMA_NETWORK` if you run the `MAINTAIN_SCHEMAS` procedure
  - `DBMS_STREAMS_ADM.INSTANTIATION_TABLE_NETWORK` if you run the `MAINTAIN_TABLES` procedure

  If the `bi_directional` parameter is set to `true`, then the procedure also sets the instantiation SCN for each shared database object at the source database.

- **Generate a Configuration Script with No Instantiation Specified:** This option does not perform an instantiation. This setting is valid only if the `perform_actions` parameter is set to `false`, and the procedure generates a configuration script. In this case, the configuration script does not perform an instantiation and does not set the instantiation SCN for each shared database object. Instead, you

must perform the instantiation and ensure that instantiation SCN values are set properly.

To specify this instantiation option, set the `instantiation` parameter to `DBMS_STREAMS_ADM.INSTANTIATION_NONE` in each procedure.

The `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` procedures do not perform an instantiation. You must perform any required instantiation actions manually after running `PRE_INSTANTIATION_SETUP` and before running `POST_INSTANTIATION_SETUP`. You also must perform any required instantiation actions manually if you use the `MAINTAIN_GLOBAL`, `MAINTAIN_SCHEMAS`, and `MAINTAIN_TABLES` procedures and set the `instantiation` parameter to `DBMS_STREAMS_ADM.INSTANTIATION_NONE`.

In these cases, you can use any instantiation method. For example, you can use Recovery Manager (RMAN) to perform a database instantiation using the RMAN `DUPLICATE` or `CONVERT DATABASE` command or a tablespace instantiation using the RMAN `TRANSPORT TABLESPACE` command. If the `bi_directional` parameter is set to `true`, then make sure the instantiation SCN values are set properly at the source database as well as the destination database.

Note:

- The `MAINTAIN_SIMPLE_TTS` and `MAINTAIN_TTS` procedures do not provide these instantiation options. These procedures always perform an instantiation by cloning the tablespace or tablespace set, transferring the files required for instantiation to the destination database, and attaching the tablespace or tablespace set at the destination database.
- If one of these procedures performs an instantiation, then the database objects, tablespace, or tablespaces set being configured for replication must exist at the source database, but they must not exist at the destination database.
- If the RMAN `DUPLICATE` or `CONVERT DATABASE` command is used for database instantiation, then the destination database cannot be the capture database.

See Also:

- Chapter 2, "Instantiation and Streams Replication"
- Chapter 10, "Performing Instantiations"

Tasks to Complete Before Configuring Streams Replication

The following sections describe tasks to complete before configuring Streams replication:

- Configure a Streams Administrator on All Databases
- Create One or More Database Links
- Create the Required Directory Objects
- Make Sure Each Source Database Is In ARCHIVELOG Mode
- Configure Log File Copying for Downstream Capture
- Make Sure the Initialization Parameters Are Set Properly

Configure a Streams Administrator on All Databases The Streams administrator at each database must have the required privileges to perform the configuration actions. The examples in this chapter assume that the username of the Streams administrator is `strmadmin` at each database.

> **See Also:** *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

Create One or More Database Links A database link from the source database to the destination database always is required before running one of the procedures. A database link from the destination database to the source database is required in any of the following cases:

- The Streams replication environment will be bi-directional.
- A Data Pump network import will be performed during instantiation.
- The destination database is the capture database for downstream capture of source database changes.
- The RMAN `DUPLICATE` or `CONVERT DATABASE` command will be used for database instantiation.

  This database link is required because the `POST_INSTANTIATION_SETUP` procedure with a non-`NULL` setting for the `instantiation_scn` parameter runs the `SET_GLOBAL_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package at the destination database. The `SET_GLOBAL_INSTANTIATION_SCN` procedure requires the database link. This database link must be created after the RMAN instantiation and before running the `POST_INSTANTIATION_SETUP` procedure.

If a third database is the capture database for downstream capture of source database changes, then the following database links are required:

- A database link is required from the third database to the source database.
- A database link is required from the third database to the destination database.

Each database link should be created in the Streams administrator's schema. For example, if the source database is `stm1.net`, the destination database is `stm2.net`, and the Streams administrator is `strmadmin` at each database, then the following statement creates the database link from the source database to the destination database:

```
CONNECT strmadmin/strmadminpw@stm1.net

CREATE DATABASE LINK stm2.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'stm2.net';
```

If a database link is required from the destination database to the source database, then the following statement creates this database link:

```
CONNECT strmadmin/strmadminpw@stm2.net

CREATE DATABASE LINK stm1.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'stm1.net';
```

If a third database is the capture database, then a database link is required from the third database to the source and destination databases. For example, if the third database is `stm3.net`, then the following statements create the database links from the third database to the source and destination databases:

```
CONNECT strmadmin/strmadminpw@stm3.net

CREATE DATABASE LINK stm1.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'stm1.net';

CREATE DATABASE LINK stm2.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'stm2.net';
```

If an RMAN database instantiation is performed, then the database link at the source database is copied to the destination database during instantiation. This copied database link should be dropped at the destination database. In this case, if the replication is bi-directional, and a database link from the destination database to the source database is required, then this database link should be created after the instantiation.

See Also:

- "Decide Whether Replication Is Bi-Directional" on page 6-9
- "Decide Whether to Configure Local or Downstream Capture for the Source Database" on page 6-7
- "Decide How to Perform Instantiation" on page 6-11

Create the Required Directory Objects A directory object is similar to an alias for a directory on a file system. The following directory objects might be required when you run one of these procedures:

- A script directory object is required if you decided to generate a configuration script. The configuration script is placed in this directory on the computer system where the procedure is run. Use the `script_directory_object` parameter when you run one of these procedures to specify the script directory object.
- A source directory object is required if you decided to perform a Data Pump export dump file instantiation, and you will use one of the following procedures: `MAINTAIN_GLOBAL`, `MAINTAIN_SCHEMAS`, `MAINTAIN_SIMPLE_TTS`, `MAINTAIN_TABLES`, or `MAINTAIN_TTS`. The Data Pump export dump file and log file are placed in this directory on the computer system running the source database. Use the `source_directory_object` parameter when you run one of these procedures to specify the source directory object. This directory object is not required if you will use the `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` procedures.
- A destination directory object is required if you decided to perform a Data Pump export dump file instantiation, and you will use one of the following procedures: `MAINTAIN_GLOBAL`, `MAINTAIN_SCHEMAS`, `MAINTAIN_SIMPLE_TTS`, `MAINTAIN_TABLES`, or `MAINTAIN_TTS`. The Data Pump export dump file is transferred from the computer system running the source database to the computer system running the destination database and placed in this directory on the computer system running the destination database. Use the `destination_directory_object` parameter when you run one of these procedures to specify the destination directory object. This directory object is not required if you will use the `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` procedures.

Each directory object must be created using the SQL statement `CREATE DIRECTORY`, and the user who invokes one of the procedures must have `READ` and `WRITE` privilege on each directory object. For example, the following statement creates a directory

object named db_files_directory that corresponds to the /usr/db_files directory:

```
CONNECT strmadmin/strmadminpw

CREATE DIRECTORY db_files_directory AS '/usr/db_files';
```

Because this directory object was created by the Streams administrator (strmadmin), this user automatically has READ and WRITE privilege on the directory object.

See Also:

- "Decide Whether to Configure Replication Directly or Generate a Script" on page 6-10
- "Decide How to Perform Instantiation" on page 6-11

Make Sure Each Source Database Is In ARCHIVELOG Mode Each source database must be in ARCHIVELOG mode before running one of these procedures (or the script generated by one of these procedures). A source database is a database that will generate changes that will be captured by a capture process. The source database always must be in ARCHIVELOG mode. If the procedure configures a bi-directional replication environment, then the destination database also must be in ARCHIVELOG mode.

See Also:

- "Decide Whether Replication Is Bi-Directional" on page 6-9
- *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode

Configure Log File Copying for Downstream Capture If you decided to use a local capture process at the source database, then log file copying is not required. However, if you decided to use downstream capture for the source database, then configure log file copying from the source database to the capture database before you run the procedure.

Complete the following steps to prepare the source database to copy its redo log files to the capture database, and to prepare the capture database to accept these redo log files:

1. Configure Oracle Net so that the source database can communicate with the capture database.

   See Also: *Oracle Database Net Services Administrator's Guide*

2. Set the following initialization parameters to configure redo transport services to copy archived redo log files from the source database to the capture database:
   - At the source database, set at least one archive log destination in the LOG_ARCHIVE_DEST_*n* initialization parameter to a directory on the computer system running the capture database. To do this, set the following attributes of this parameter:
     - SERVICE - Specify the network service name of the capture database.
     - ARCH or LGWR ASYNC - If you specify ARCH (the default), then the archiver process (ARC*n*) will archive the redo log files to the capture database. If you specify LGWR ASYNC, then the log writer process (LGWR) will archive the redo log files to the capture database. Either ARCH or LGWR ASYNC is acceptable for a capture database destination.

- MANDATORY or OPTIONAL - If you specify MANDATORY, then archiving of a redo log file to the capture database must succeed before the corresponding online redo log at the source database can be overwritten. If you specify OPTIONAL, then successful archiving of a redo log file to the capture database is not required before the corresponding online redo log at the source database can be overwritten. Either MANDATORY or OPTIONAL is acceptable for a capture database destination. If neither the MANDATORY nor the OPTIONAL attribute is specified, then the default is OPTIONAL.
- NOREGISTER - Specify this attribute so that the capture database location is not recorded in the capture database control file.
- TEMPLATE - Specify a directory and format template for archived redo logs at the capture database. The TEMPLATE attribute overrides the LOG_ARCHIVE_FORMAT initialization parameter settings at the capture database. The TEMPLATE attribute is valid only with remote destinations. Make sure the format uses all of the following variables at each source database: %t, %s, and %r.

The following is an example of an LOG_ARCHIVE_DEST_*n* setting that specifies a capture database:

```
LOG_ARCHIVE_DEST_2='SERVICE=STM2.NET ARCH OPTIONAL NOREGISTER
   TEMPLATE=/usr/oracle/log_for_stm1/stm1_arch_%t_%s_%r.log'
```

If another source database transfers log files to this capture database, then, in the initialization parameter file at this other source database, you can use the TEMPLATE attribute to specify a different directory and format for the log files at the capture database. The log files from each source database are kept separate at the capture database.

Tip: Log files from a remote source database should be kept separate from local database log files. In addition, if the capture database contains log files from multiple source databases, then the log files from each source database should be kept separate from each other.

- At the source database, set the LOG_ARCHIVE_DEST_STATE_*n* initialization parameter that corresponds with the LOG_ARCHIVE_DEST_*n* parameter for the capture database to ENABLE.

  For example, if the LOG_ARCHIVE_DEST_2 initialization parameter is set for the capture database, then set one LOG_ARCHIVE_DEST_STATE_2 parameter in the following way:

  ```
  LOG_ARCHIVE_DEST_STATE_2=ENABLE
  ```

- At the source database, make sure the setting for the LOG_ARCHIVE_CONFIG initialization parameter includes the send value.
- At the downstream database, make sure the setting for the LOG_ARCHIVE_CONFIG initialization parameter includes the receive value.

  See Also: *Oracle Database Reference* and *Oracle Data Guard Concepts and Administration* for more information about these initialization parameters

3. If you reset any initialization parameters while the instance is running at a database in Step 2, then you might want to reset them in the initialization

parameter file as well, so that the new values are retained when the database is restarted.

If you did not reset the initialization parameters while the instance was running, but instead reset them in the initialization parameter file in Step 2, then restart the database. The source database must be open when it sends redo log files to the capture database because the global name of the source database is sent to the capture database only if the source database is open.

> **See Also:** "Decide Whether to Configure Local or Downstream Capture for the Source Database" on page 6-7

Make Sure the Initialization Parameters Are Set Properly Certain initialization parameters are important in a Streams environment. Make sure the initialization parameters are set properly at all databases before running one of the procedures.

> **See Also:** *Oracle Streams Concepts and Administration* for information about initialization parameters that are important in a Streams environment

Configuring Database Replication Using the DBMS_STREAMS_ADM Package

You can use the following procedures in the `DBMS_STREAMS_ADM` package to configure database replication:

- `MAINTAIN_GLOBAL`
- `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP`

The `MAINTAIN_GLOBAL` procedure automatically excludes database objects that are not supported by Streams from the replication environment. The `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` procedures do not automatically exclude database objects. Instead, these procedures enable you to specify which database objects to exclude from the replication environment. Query the `DBA_STREAMS_UNSUPPORTED` data dictionary view to determine which database objects are not supported by Streams. If unsupported database objects are not excluded, then capture errors will result.

The example in this section uses the `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP` procedures to configure database replication. The replication configuration will exclude all database objects that are not supported by Streams. The source database is `stm1.net`, and the destination database is `stm2.net`.

Assume that the following decisions were made about the configuration:

- DDL changes will be maintained.
- Local capture will be configured for the source database.
- The replication environment will be bi-directional.
- An RMAN database instantiation will be performed.
- The procedures will configure the replication environment directly. Configuration scripts will not be generated.

Note: A capture process never captures changes in the `SYS`, `SYSTEM`, or `CTXSYS` schemas. Changes to these schemas are not maintained by Streams in the replication configuration described in this section.

See Also: "Decisions to Make Before Configuring Streams Replication" on page 6-6 for more information about these decisions

Complete the following steps to use the PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP procedures to configure the environment:

1. Complete the required tasks before running the PRE_INSTANTIATION_SETUP procedure. See "Tasks to Complete Before Configuring Streams Replication" on page 6-12 for instructions.

   For this configuration, the following tasks must be completed:

   - Configure a Streams administrator at both databases.
   - Create a database link from the source database stm1.net to the destination database stm2.net.
   - Make sure both databases are in ARCHIVELOG mode.
   - Make sure the initialization parameters are set properly at both databases.

   A database link is required from the destination database to the source database. However, because RMAN will be used for database instantiation, this database link must be created after instantiation. This database link is required because the replication environment will be bi-directional and because RMAN will be used for database instantiation.

2. Connect to the source database as the Streams administrator, and run the PRE_INSTANTIATION_SETUP procedure:

```
CONNECT strmadmin/strmadminpw@stm1.net

DECLARE
  empty_tbs  DBMS_STREAMS_TABLESPACE_ADM.TABLESPACE_SET;
BEGIN
  DBMS_STREAMS_ADM.PRE_INSTANTIATION_SETUP(
    maintain_mode        => 'GLOBAL',
    tablespace_names     => empty_tbs,
    source_database      => 'stm1.net',
    destination_database => 'stm2.net',
    perform_actions      => true,
    bi_directional       => true,
    include_ddl          => true,
    start_processes      => true,
    exclude_schemas      => '*',
    exclude_flags        => DBMS_STREAMS_ADM.EXCLUDE_FLAGS_UNSUPPORTED +
                            DBMS_STREAMS_ADM.EXCLUDE_FLAGS_DML +
                            DBMS_STREAMS_ADM.EXCLUDE_FLAGS_DDL);
END;
/
```

   Notice that the start_processes parameter is set to true. Therefore, each capture process and apply process created during the configuration is started automatically.

   Also, notice the values specified for the exclude_schemas and exclude_flags parameters. The asterisk (*) specified for exclude_schemas indicates that certain database objects in every schema in the database might be excluded from the replication environment. The value specified for the exclude_flags parameter indicates that DML and DDL changes for all unsupported database objects are excluded from the replication environment. Rules are placed in the negative rule sets for the capture processes to exclude these database objects.

Because the procedure is run at the source database, local capture is configured at the source database.

Because this procedure configures a bi-directional replication environment, do not allow DML or DDL changes to the shared database objects at the destination database while the procedure is running.

If this procedure encounters an error and stops, then see "Recovering from Configuration Errors" on page 13-1 for information about either recovering from the error or rolling back the configuration operation.

3. Perform the instantiation. You can use any of the methods described in Chapter 10, "Performing Instantiations" to complete the instantiation. This example uses the RMAN `DUPLICATE` command to perform the instantiation by performing the following steps:

   a. Create a backup of the source database if one does not exist. RMAN requires a valid backup for duplication. In this example, create a backup of `stm1.net` if one does not exist.

   b. Connect to the source database as the Streams administrator and determine the until SCN for the RMAN `DUPLICATE` command:

```
CONNECT strmadmin/strmadminpw@stm1.net

SET SERVEROUTPUT ON SIZE 1000000
DECLARE
  until_scn NUMBER;
BEGIN
  until_scn:= DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
      DBMS_OUTPUT.PUT_LINE('Until SCN: ' || until_scn);
END;
/
```

   Make a note of the until SCN returned. You will use this number in Step e. For this example, assume that the returned until SCN is `45442631`.

   c. Connect to the source database as a system administrator in SQL*Plus and archive the current online redo log:

```
ALTER SYSTEM ARCHIVE LOG CURRENT;
```

   d. Prepare your environment for database duplication, which includes preparing the destination database as an auxiliary instance for duplication. See *Oracle Database Backup and Recovery Advanced User's Guide* for instructions.

   e. Use the RMAN `DUPLICATE` command with the `OPEN RESTRICTED` option to instantiate the source database at the destination database. The `OPEN RESTRICTED` option is required. This option enables a restricted session in the duplicate database by issuing the following SQL statement: `ALTER SYSTEM ENABLE RESTRICTED SESSION`. RMAN issues this statement immediately before the duplicate database is opened.

   You can use the `UNTIL SCN` clause to specify an SCN for the duplication. Use the until SCN determined in Step b for this clause. Archived redo logs must be available for the until SCN specified and for higher SCN values. Therefore, Step c archived the redo log containing the until SCN.

   Make sure you use `TO` *`database_name`* in the `DUPLICATE` command to specify the name of the duplicate database. In this example, the duplicate database is `stm2.net`. Therefore, the `DUPLICATE` command for this example includes `TO stm2.net`.

The following is an example of an RMAN `DUPLICATE` command:

```
rman
RMAN> CONNECT TARGET SYS/change_on_install@stm1.net
RMAN> CONNECT AUXILIARY SYS/change_on_install@stm2.net
RMAN> RUN
      {
        SET UNTIL SCN 45442631;
        ALLOCATE AUXILIARY CHANNEL stm2 DEVICE TYPE sbt;
        DUPLICATE TARGET DATABASE TO stm2
        NOFILENAMECHECK
        OPEN RESTRICTED;
      }
```

f. Connect to the destination database as a system administrator in SQL*Plus, and rename the global name. After an RMAN database instantiation, the destination database has the same global name as the source database. Rename the global name of the destination database back to its original name with the following statement:

```
ALTER DATABASE RENAME GLOBAL_NAME TO stm2.net;
```

g. Connect to the destination database as the Streams administrator, and drop the database link from the source database to the destination database that was cloned from the source database:

```
CONNECT strmadmin/strmadminpw@stm2.net

DROP DATABASE LINK stm2.net;
```

4. While still connected to the destination database as the Streams administrator, create a database link from the destination database to the source database:

```
CREATE DATABASE LINK stm1.net CONNECT TO strmadmin
  IDENTIFIED BY strmadminpw USING 'stm1.net';
```

See Step 1 for information about why this database link is required.

5. Connect to the source database as the Streams administrator, and run the `POST_INSTANTIATION_SETUP` procedure:

```
CONNECT strmadmin/strmadminpw@stm1.net

DECLARE
  empty_tbs  DBMS_STREAMS_TABLESPACE_ADM.TABLESPACE_SET;
BEGIN
  DBMS_STREAMS_ADM.POST_INSTANTIATION_SETUP(
    maintain_mode        => 'GLOBAL',
    tablespace_names     => empty_tbs,
    source_database      => 'stm1.net',
    destination_database => 'stm2.net',
    perform_actions      => true,
    bi_directional       => true,
    include_ddl          => true,
    start_processes      => true,
    instantiation_scn    => 45442630,
    exclude_schemas      => '*',
    exclude_flags        => DBMS_STREAMS_ADM.EXCLUDE_FLAGS_UNSUPPORTED +
                            DBMS_STREAMS_ADM.EXCLUDE_FLAGS_DML +
                            DBMS_STREAMS_ADM.EXCLUDE_FLAGS_DDL);
END;
/
```

The parameter values specified in both the PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP procedures must match, except for the values of the following parameters: perform_actions, script_name, script_directory_object, and start_processes.

Also, notice that the instantiation_scn parameter is set to 45442630. The RMAN DUPLICATE command duplicates the database up to one less than the SCN value specified in the UNTIL SCN clause. Therefore, you should subtract one from the until SCN value that you specified when you ran the DUPLICATE command in Step 3e. In this example, the until SCN was set to 45442631. Therefore, the instantiation_scn parameter should be set to 45442631 - 1, or 45442630.

If the instantiation SCN was set for the shared database objects at the destination database during instantiation, then the instantiation_scn parameter should be set to NULL. For example, the instantiation SCN might be set during a full database export/import.

Because this procedure configures a bi-directional replication environment, do not allow DML or DDL changes to the shared database objects at the destination database while the procedure is running.

If this procedure encounters an error and stops, then see "Recovering from Configuration Errors" on page 13-1 for information about either recovering from the error or rolling back the configuration operation.

6. At the destination database, connect as an administrator with SYSDBA privilege in SQL*Plus and use the ALTER SYSTEM statement to disable the RESTRICTED SESSION:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

7. Configure conflict resolution for the shared database objects if necessary.

   Typically, conflicts are possible in a bi-directional replication environment. If conflicts are possible in the environment created by the PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP procedures, then configure conflict resolution before you allow users to make changes to the shared database objects.

The bi-directional replication environment configured in this example has the following characteristics:

- Database supplemental logging is configured at both databases.
- The stm1.net database has two queues and queue tables with system-generated names. One queue is for the local capture process, and one queue is for the apply process.
- The stm2.net database has two queues and queue tables with system-generated names. One queue is for the local capture process, and one queue is for the apply process.
- At the stm1.net database, a capture process with a system-generated name captures DML and DDL changes to all of the database objects in the database that are supported by Streams.
- At the stm2.net database, a capture process with a system-generated name captures DML and DDL changes to all of the database objects in the database that are supported by Streams.

- A propagation running on the stm1.net database with a system-generated name propagates the captured changes from a queue at the stm1.net database to a queue at the stm2.net database.
- A propagation running on the stm2.net database with a system-generated name propagates the captured changes from a queue at the stm2.net database to a queue at the stm1.net database.
- At the stm1.net database, an apply process with a system-generated name dequeues the changes from its queue and applies them to the database objects.
- At the stm2.net database, an apply process with a system-generated name dequeues the changes from its queue and applies them to the database objects.
- Tags are used to avoid change cycling. Specifically, each apply process uses an apply tag so that redo records for changes applied by the apply process include the tag. Each apply process uses an apply tag that is unique in the replication environment. Each propagation discards changes that have the tag of the apply process running on the same database.

See Also:

- Chapter 3, "Streams Conflict Resolution" and "Managing Streams Conflict Detection and Resolution" on page 9-22
- Chapter 4, "Streams Tags"

Configuring Tablespace Replication Using the DBMS_STREAMS_ADM Package

You can use the following procedures in the DBMS_STREAMS_ADM package to configure tablespace replication:

- MAINTAIN_SIMPLE_TTS
- MAINTAIN_TTS
- PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP

You can use the MAINTAIN_SIMPLE_TTS procedure to configure Streams replication for a simple tablespace, and you can use the MAINTAIN_TTS procedure to configure Streams replication for a set of self-contained tablespaces. These procedures use transportable tablespaces, Data Pump, the DBMS_STREAMS_TABLESPACE_ADM package, and the DBMS_FILE_TRANSFER package to configure the environment.

A self-contained tablespace has no references from the tablespace pointing outside of the tablespace. For example, if an index in the tablespace is for a table in a different tablespace, then the tablespace is not self-contained. A simple tablespace is a self-contained tablespace that uses only one datafile. When there is more than one tablespace in a tablespace set, a self-contained tablespace set has no references from inside the set of tablespaces pointing outside of the set of tablespaces.

These procedures clone the tablespace or tablespaces being configured for replication from the source database to the destination database. The MAINTAIN_SIMPLE_TTS procedure uses the CLONE_SIMPLE_TABLESPACE procedure in the DBMS_STREAMS_TABLESPACE_ADM package, and the MAINTAIN_TTS procedure uses the CLONE_TABLESPACES procedure in the DBMS_STREAMS_TABLESPACE_ADM package. When a tablespace is cloned, it is made read-only automatically until the clone operation is complete.

The example in this section uses the MAINTAIN_TTS procedure to configure a Streams replication environment that maintains the following tablespaces using Streams:

- tbs1
- tbs2

The source database is stm1.net, and the destination database is stm2.net.

Assume that the following decisions were made about the configuration:

- DDL changes to these tablespaces and the database objects in these tablespaces will be maintained.
- A downstream capture process running on the destination database (stm2.net) will capture changes made to the source database (stm1.net).
- The replication environment will be bi-directional.
- The MAINTAIN_TTS procedure will configure the replication environment directly. A configuration script will not be generated.

> **See Also:** "Decisions to Make Before Configuring Streams Replication" on page 6-6 for more information about these decisions

In addition, this example makes the following assumptions:

- The tablespaces tbs1 and tbs2 make a self-contained tablespace set at the source database stm1.net.
- The datafiles for the tablespace set are both in the /orc/dbs directory at the source database stm1.net.
- The stm2.net database does not contain the tablespace set currently.

The MAINTAIN_SIMPLE_TTS and MAINTAIN_TTS procedures automatically exclude database objects that are not supported by Streams from the replication environment by adding rules to the negative rule set of each capture and apply process. The PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP procedures enable you to specify which database objects to exclude from the replication environment.

Query the DBA_STREAMS_UNSUPPORTED data dictionary view to determine which database objects are not supported by Streams. If unsupported database objects are not excluded, then capture errors will result.

Complete the following steps to use the MAINTAIN_TTS procedure to configure the environment:

1. Complete the required tasks before running the MAINTAIN_TTS procedure. See "Tasks to Complete Before Configuring Streams Replication" on page 6-12 for instructions.

   For this configuration, the following tasks must be completed:

   - Configure a Streams administrator at both databases.
   - Create a database link from the source database stm1.net to the destination database stm2.net.
   - Because the replication environment will be bi-directional, and because downstream capture will be configured at the destination database, create a database link from the destination database stm2.net to the source database stm1.net.

- Create the following required directory objects:
  - A source directory object at the source database. This example assumes that this directory object is SOURCE_DIRECTORY.
  - A destination directory object at the destination database. This example assumes that this directory object is DEST_DIRECTORY.
- Make sure both databases are in ARCHIVELOG mode.
- Because the destination database will be the capture database for changes made to the source database, configure log file copying from the source database stm1.net to the destination database stm2.net.
- Make sure the initialization parameters are set properly at both databases.

2. While connected as the Streams administrator to the database that contains the tablespace set, create a directory object for the directory that contains the datafiles for the tablespaces in the tablespace set. For example, the following statement creates a directory object named tbs_directory that corresponds to the /orc/dbs directory:

```
CONNECT strmadmin/strmadminpw@stm1.net

CREATE DIRECTORY tbs_directory AS '/orc/dbs';
```

If the datafiles are in multiple directories, then a directory object must exist for each of these directories, and the user who runs the MAINTAIN_TTS procedure in Step 3 must have READ privilege on these directory objects. In this example, the Streams administrator has this privilege because this user creates the directory object.

3. While connected as the Streams administrator to the destination database, run the MAINTAIN_TTS procedure:

```
CONNECT strmadmin/strmadminpw@stm2.net

DECLARE
  t_names  DBMS_STREAMS_TABLESPACE_ADM.TABLESPACE_SET;
BEGIN
  -- Tablespace names
  t_names(1) := 'TBS1';
  t_names(2) := 'TBS2';
  DBMS_STREAMS_ADM.MAINTAIN_TTS(
     tablespace_names             => t_names,
     source_directory_object      => 'SOURCE_DIRECTORY',
     destination_directory_object => 'DEST_DIRECTORY',
     source_database              => 'stm1.net',
     destination_database         => 'stm2.net',
     perform_actions              => true,
     bi_directional               => true,
     include_ddl                  => true);
END;
/
```

When this procedure completes, the Streams bi-directional replication environment is configured. The procedure automatically generates names for the ANYDATA queues, capture processes, propagations, and apply processes it creates. If you do not want system-generated names for these components, you can specify names by using additional parameters available in the MAINTAIN_TTS procedure. This procedure also starts the queues, capture processes, propagations, and apply processes.

Because the procedure is run at the destination database, downstream capture is configured at the destination database for changes to the source database.

If this procedure encounters an error and stops, then see "Recovering from Configuration Errors" on page 13-1 for information about either recovering from the error or rolling back the configuration operation.

4. Configure conflict resolution for the database objects in the tablespace set if necessary.

   Typically, conflicts are possible in a bi-directional replication environment. If conflicts are possible in the environment created by the `MAINTAIN_TTS` procedure, then configure conflict resolution before you allow users to make changes to the objects in the tablespace set.

The resulting bi-directional replication environment has the following characteristics:

- Supplemental logging is configured for the shared database objects at both databases.
- The `stm1.net` database has a queue and queue table with system-generated names. This queue is for the apply process.
- The `stm2.net` database has three queues and queue tables with system-generated names. One queue is for the downstream capture process, one queue is for the local capture process, and one queue is for the apply process.
- At the `stm2.net` database, a downstream capture process with a system-generated name captures DML and DDL changes made to the source database. Specifically, this downstream capture process captures DML changes made to the tables in the `tbs1` and `tbs2` tablespaces and DDL changes to these tablespaces and the database objects in them.
- At the `stm2.net` database, a local capture process with a system-generated name captures DML and DDL changes made to the destination database. Specifically, this local capture process captures DML changes to the tables in the `tbs1` and `tbs2` tablespaces and DDL changes to these tablespaces and the database objects in them.
- A propagation running on the `stm2.net` database with a system-generated name propagates the changes captured by the downstream capture process from the queue for the downstream capture process to the queue for the apply process within the `stm2.net` database.
- A propagation running on the `stm2.net` database with a system-generated name propagates the changes captured by the local capture process from the queue for the local capture process to the queue in the `stm1.net` database.
- At the `stm1.net` database, an apply process with a system-generated name dequeues the changes from the queue and applies them to the shared database objects.
- At the `stm2.net` database, an apply process with a system-generated name dequeues the changes from its queue and applies them to the shared database objects.
- Tags are used to avoid change cycling. Specifically, each apply process uses an apply tag so that redo records for changes applied by the apply process include the tag. Each apply process uses an apply tag that is unique in the replication environment. Each propagation discards changes that have the tag of the apply process running on the same database.

See Also:

- Chapter 3, "Streams Conflict Resolution" and "Managing Streams Conflict Detection and Resolution" on page 9-22
- Chapter 4, "Streams Tags"

Configuring Schema Replication Using the DBMS_STREAMS_ADM Package

You can use the MAINTAIN_SCHEMAS in the DBMS_STREAMS_ADM package to configure schema replication. The example in this section uses this procedure to configure a Streams replication environment that maintains the hr schema. The source database is stm1.net, and the destination database is stm2.net.

Assume that the following decisions were made about the configuration:

- DDL changes to hr schema and the database objects in the hr schema will be maintained.
- The replication environment will be bi-directional.
- A downstream capture process running on a third database named stm3.net will capture changes made to the source database (stm1.net), and a propagation at stm3.net will propagate these captured changes to the destination database (stm2.net).
- A Data Pump export dump file instantiation will be performed.
- The MAINTAIN_SCHEMAS procedure will configure the replication environment directly. A configuration script will not be generated.

The MAINTAIN_SCHEMAS procedure automatically excludes database objects that are not supported by Streams from the replication environment by adding rules to the negative rule set of each capture and apply process. Query the DBA_STREAMS_UNSUPPORTED data dictionary view to determine which database objects are not supported by Streams. If unsupported database objects are not excluded, then capture errors will result.

See Also: "Decisions to Make Before Configuring Streams Replication" on page 6-6 for more information about these decisions

Complete the following steps to use the MAINTAIN_SCHEMAS procedure to configure the environment:

1. Complete the required tasks before running the MAINTAIN_SCHEMAS procedure. See "Tasks to Complete Before Configuring Streams Replication" on page 6-12 for instructions.

   For this configuration, the following tasks must be completed:

   - Configure a Streams administrator at all three databases.
   - Create a database link from the source database stm1.net to the destination database stm2.net.
   - Because downstream capture will be configured at the third database, create a database link from the third database stm3.net to the source database stm1.net.
   - Because downstream capture will be configured at the third database, create a database link from the third database stm3.net to the destination database stm2.net.

- Because the replication environment will be bi-directional, create a database link from the destination database `stm2.net` to the source database `stm1.net`.
- Create the following required directory objects:
  - A source directory object at the source database. This example assumes that this directory object is `SOURCE_DIRECTORY`.
  - A destination directory object at the destination database. This example assumes that this directory object is `DEST_DIRECTORY`.
- Make sure the source database and destination databases are in `ARCHIVELOG` mode.
- Because a third database (`stm3.net`) will be the capture database for changes made to the source database, configure log file copying from the source database `stm1.net` to the third database `stm3.net`.
- Make sure the initialization parameters are set properly at all databases.

2. Connect to the third database as the Streams administrator and run the `MAINTAIN_SCHEMAS` procedure:

```
CONNECT strmadmin/strmadminpw@stm3.net

BEGIN
  DBMS_STREAMS_ADM.MAINTAIN_SCHEMAS(
    schema_names                 => 'hr',
    source_directory_object      => 'SOURCE_DIRECTORY',
    destination_directory_object => 'DEST_DIRECTORY',
    source_database              => 'stm1.net',
    destination_database         => 'stm2.net',
    perform_actions              => true,
    dump_file_name               => 'export_hr.dmp',
    capture_queue_table          => 'rep_capture_queue_table',
    capture_queue_name           => 'rep_capture_queue',
    capture_queue_user           => NULL,
    apply_queue_table            => 'rep_dest_queue_table',
    apply_queue_name             => 'rep_dest_queue',
    apply_queue_user             => NULL,
    capture_name                 => 'capture_hr',
    propagation_name             => 'prop_hr',
    apply_name                   => 'apply_hr',
    log_file                     => 'export_hr.clg',
    bi_directional               => true,
    include_ddl                  => true,
    instantiation                => DBMS_STREAMS_ADM.INSTANTIATION_SCHEMA);
END;
/
```

Because this procedure configures a bi-directional replication environment, do not allow DML or DDL changes to the shared database objects at the destination database while the procedure is running.

Because the procedure is run at the third database, downstream capture is configured at the third database for changes to the source database.

If this procedure encounters an error and stops, then see "Recovering from Configuration Errors" on page 13-1 for information about either recovering from the error or rolling back the configuration operation.

3. Configure conflict resolution for the shared database objects if necessary.

   Typically, conflicts are possible in a bi-directional replication environment. If conflicts are possible in the environment created by the `MAINTAIN_SCHEMAS` procedure, then configure conflict resolution before you allow users to make changes to the shared database objects.

The bi-directional replication environment configured in this example has the following characteristics:

- Supplemental logging is configured for the shared database objects at the source and destination databases.
- The `stm1.net` database has a queue named `rep_dest_queue` which uses a queue table named `rep_dest_queue_table`. This queue is for the apply process.
- The `stm2.net` database has a queue named `rep_capture_queue` which uses a queue table named `rep_capture_queue_table`. This queue is for the local capture process.
- The `stm2.net` database has a queue named `rep_dest_queue` which uses a queue table named `rep_dest_queue_table`. This queue is for the apply process.
- The `stm3.net` database has a queue named `rep_capture_queue` which uses a queue table named `rep_capture_queue_table`. This queue is for the downstream capture process.
- At the `stm3.net` database, a downstream capture process named `capture_hr` captures DML and DDL changes to the `hr` schema and the database objects in the schema at the source database.
- At the `stm2.net` database, a local capture process named `capture_hr` captures DML and DDL changes to the `hr` schema and the database objects in the schema at the destination database.
- A propagation running on the `stm3.net` database named `prop_hr` propagates the captured changes from the queue in the `stm3.net` database to the queue in the `stm2.net` database.
- A propagation running on the `stm2.net` database named `prop_hr` propagates the captured changes from the queue in the `stm2.net` database to the queue in the `stm1.net` database.
- At the `stm1.net` database, an apply process named `apply_hr` dequeues the changes from `rep_dest_queue` and applies them to the database objects.
- At the `stm2.net` database, an apply process named `apply_hr` dequeues the changes from `rep_dest_queue` and applies them to the database objects.
- Tags are used to avoid change cycling. Specifically, each apply process uses an apply tag so that redo records for changes applied by the apply process include the tag. Each apply process uses an apply tag that is unique in the replication environment. Each propagation discards changes that have the tag of the apply process running on the same database.

See Also:

- Chapter 3, "Streams Conflict Resolution" and "Managing Streams Conflict Detection and Resolution" on page 9-22
- Chapter 4, "Streams Tags"

Configuring Table Replication Using the DBMS_STREAMS_ADM Package

You can use the MAINTAIN_TABLES in the DBMS_STREAMS_ADM package to configure table replication. The example in this section uses this procedure to configure a Streams replication environment that maintains the tables in the hr schema. The source database is stm1.net, and the destination database is stm2.net.

Assume that the following decisions were made about the configuration:

- The replication environment should maintain DDL changes to the following tables in the hr schema:
  - departments
  - employees
- The replication environment should not maintain DDL changes to the following tables in the hr schema:
  - countries
  - regions
  - locations
  - jobs
  - job_history
- Local capture will be configured for the source database.
- The replication environment will be single source, not bi-directional.
- A Data Pump network import instantiation will be performed.
- The MAINTAIN_TABLES procedure will not configure the replication environment directly. Instead, a configuration script will be generated, and this script will be modified so that DDL changes to the following tables are maintained: departments and employees.

> **See Also:** "Decisions to Make Before Configuring Streams Replication" on page 6-6 for more information about these decisions

Complete the following steps to use the MAINTAIN_TABLES procedure to configure the environment:

1. Complete the required tasks before running the MAINTAIN_TABLES procedure. See "Tasks to Complete Before Configuring Streams Replication" on page 6-12 for instructions.

   For this configuration, the following tasks must be completed:

   - Configure a Streams administrator at both databases.
   - Create a database link from the source database stm1.net to the destination database stm2.net.
   - Because the MAINTAIN_TABLES procedure will perform a Data Pump network import instantiation, create a database link from the destination database stm2.net to the source database stm1.net.
   - Create a script directory object at the source database. This example assumes that this directory object is SCRIPT_DIRECTORY.
   - Make sure the source database stm1.net is in ARCHIVELOG mode.
   - Make sure the initialization parameters are set properly at both databases.

2. Connect to the source database as the Streams administrator and run the MAINTAIN_TABLES procedure:

```
CONNECT strmadmin/strmadminpw@stm1.net

DECLARE
  tables DBMS_UTILITY.UNCL_ARRAY;
  BEGIN
    tables(1) := 'hr.departments';
    tables(2) := 'hr.employees';
    tables(3) := 'hr.countries';
    tables(4) := 'hr.regions';
    tables(5) := 'hr.locations';
    tables(6) := 'hr.jobs';
    tables(7) := 'hr.job_history';
    DBMS_STREAMS_ADM.MAINTAIN_TABLES(
      table_names                 => tables,
      source_directory_object     => NULL,
      destination_directory_object => NULL,
      source_database             => 'stm1.net',
      destination_database        => 'stm2.net',
      perform_actions             => false,
      script_name                 => 'configure_rep.sql',
      script_directory_object     => 'SCRIPT_DIRECTORY',
      bi_directional              => false,
      include_ddl                 => false,
      instantiation      => DBMS_STREAMS_ADM.INSTANTIATION_TABLE_NETWORK);
END;
/
```

   The configure_rep.sql script generated by the procedure uses default values for the parameters that are not specified in the procedure call. The script uses system-generated names for the ANYDATA queues, queue tables, capture process, propagation, and apply process it creates. You can specify different names by using additional parameters available in the MAINTAIN_TABLES procedure. Notice that the include_ddl parameter is set to false. Therefore, the script does not configure the replication environment to maintain DDL changes to the tables.

3. Modify the configure_rep.sql script:

   a. Navigate to the directory that corresponds with the SCRIPT_DIRECTORY directory object on the computer system running the source database.

   b. Open the configure_rep.sql script in a text editor. Consider making a backup of this script before modifying it.

   c. In the script, find the ADD_TABLE_RULES and ADD_TABLE_PROPAGATION_RULES procedure calls that create the table rules for the hr.departments and hr.employees tables. For example, the procedure calls for the capture process look similar to the following:

```
dbms_streams_adm.add_table_rules(
    table_name => '"HR"."DEPARTMENTS"',
    streams_type => 'CAPTURE',
    streams_name => '"STM1$CAP"',
    queue_name => '"STRMADMIN"."STM1$CAPQ"',
    include_dml => TRUE,
    include_ddl => FALSE,
    include_tagged_lcr => TRUE,
    source_database => 'STM1.NET',
```

```
    inclusion_rule => TRUE,
    and_condition => get_compatible);

dbms_streams_adm.add_table_rules(
    table_name => '"HR"."EMPLOYEES"',
    streams_type => 'CAPTURE',
    streams_name => '"STM1$CAP"',
    queue_name => '"STRMADMIN"."STM1$CAPQ"',
    include_dml => TRUE,
    include_ddl => FALSE,
    include_tagged_lcr => TRUE,
    source_database => 'STM1.NET',
    inclusion_rule => TRUE,
    and_condition => get_compatible);
```

d. In the procedure calls that you found in Step c, change the setting of the `include_ddl` parameter to `TRUE`. For example, the procedure calls for the capture process should look similar to the following after the modification:

```
dbms_streams_adm.add_table_rules(
    table_name => '"HR"."DEPARTMENTS"',
    streams_type => 'CAPTURE',
    streams_name => '"STM1$CAP"',
    queue_name => '"STRMADMIN"."STM1$CAPQ"',
    include_dml => TRUE,
    include_ddl => TRUE,
    include_tagged_lcr => TRUE,
    source_database => 'STM1.NET',
    inclusion_rule => TRUE,
    and_condition => get_compatible);

dbms_streams_adm.add_table_rules(
    table_name => '"HR"."EMPLOYEES"',
    streams_type => 'CAPTURE',
    streams_name => '"STM1$CAP"',
    queue_name => '"STRMADMIN"."STM1$CAPQ"',
    include_dml => TRUE,
    include_ddl => TRUE,
    include_tagged_lcr => TRUE,
    source_database => 'STM1.NET',
    inclusion_rule => TRUE,
    and_condition => get_compatible);
```

Remember to change the procedure calls for all capture processes, propagations, and apply processes.

e. Save and close the `configure_rep.sql` script.

4. At the source database, connect as the Streams administrator, and run the configuration script:

```
CONNECT strmadmin/strmadminpw@stm1.net

SET ECHO ON
SPOOL configure_rep.out
@configure_rep.sql
```

The script prompts you to supply information about the database names and the Streams administrators. When this configuration script completes, the Streams single-source replication environment is configured. The script also starts the queues, capture process, propagations, and apply process.

The resulting single-source replication environment has the following characteristics:

- At the source database, supplemental logging is configured for the shared database objects.
- The source database `stm1.net` has a queue and queue table with system-generated names.
- The destination database `stm2.net` has a queue and queue table with system-generated names.
- At the source database, a capture process with a system-generated name captures DML changes to all of the tables in the `hr` schema and DDL changes to the `hr.departments` and `hr.employees` tables.
- A propagation running on the source database with a system-generated name propagates the captured changes from the queue at the source database to the queue at the destination database.
- At the destination database, an apply process with a system-generated name dequeues the changes from the queue and applies them to the tables at the destination database.

7

Flexible Streams Replication Configuration

This chapter describes flexible methods for configuring Streams replication between two or more databases. This chapter includes step-by-step instructions for configuring each Streams component to build a single-source or multiple-source replication environment.

If possible, consider using a simple method for configuring Streams replication described in Chapter 6, "Simple Streams Replication Configuration". You can either use the Streams tool in Enterprise Manager or a single procedure in the `DBMS_STREAMS_ADM` package configure all of the Streams components in a replication environment with two databases. Also, you can use a simple method and still meet custom requirements for your replication environment in one of the following ways:

- You can use a simple method to generate a configuration script and modify the script to meet your requirements.
- You can use a simple method to configure Streams replication between two databases and add new database objects or databases to the environment by following the instructions in Chapter 8, "Adding to a Streams Replication Environment".

However, if you require more flexibility in your Streams replication configuration than what is available with the simple methods, then you can follow the instructions in this chapter to configure the environment.

This chapter contains these topics:

- Creating a New Streams Single-Source Environment
- Creating a New Streams Multiple-Source Environment

Note:

- The instructions in the following sections assume you will use the `DBMS_STREAMS_ADM` package to configure your Streams environment. If you use other packages, then extra steps might be necessary for each task.
- Certain types of database objects are not supported by Streams. When you configure a Streams environment, ensure that no capture process attempts to capture changes to an unsupported database object. To list unsupported database objects, query the `DBA_STREAMS_UNSUPPORTED` data dictionary view.

Creating a New Streams Single-Source Environment

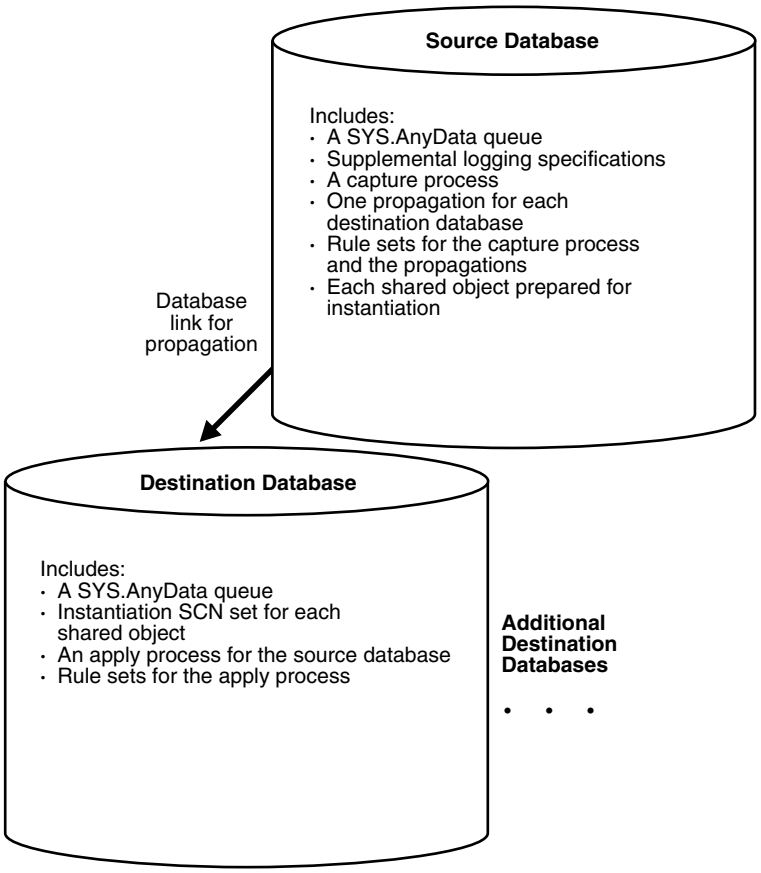
This section lists the general steps to perform when creating a new single-source Streams environment. A single-source environment is one in which there is only one source database for shared data. There can be more than one source database in a single-source environment, but no two source databases capture any of the same data.

Before starting capture processes and configuring propagations in a new Streams environment, make sure any propagations or apply processes that will receive LCRs are configured to handle these LCRs. That is, the propagations or apply processes should exist, and each one should be associated with rule sets that handle the LCRs appropriately. If these propagations and apply processes are not configured properly to handle these LCRs, then LCRs can be lost.

This example assumes that the shared database objects are read-only at the destination databases. If the shared objects are read/write at the destination databases, then the replication environment will not stay synchronized because Streams is not configured to replicate the changes made to the shared objects at the destination databases.

Figure 7–1 shows an example Streams single-source replication environment.

Figure 7–1 Example Streams Single-Source Environment



You can create a Streams environment that is more complicated than the one shown in Figure 7–1. For example, a single-source Streams environment can use downstream capture and directed networks.

In general, if you are configuring a new Streams single-source environment in which changes for shared objects are captured at one database and then propagated and applied at remote databases, then you should configure the environment in the following order:

1. Complete the necessary tasks to prepare each database in your environment for Streams:
   - Configure a Streams administrator
   - Set initialization parameters relevant to Streams
   - For each database that will run a capture process, prepare the database to run a capture process
   - Configure network connectivity and database links

   Some of these tasks might not be required at certain databases.

   See Also: *Oracle Streams Concepts and Administration* for more information about preparing a database for Streams

2. Create any necessary `ANYDATA` queues that do not already exist. When you create a capture process or apply process, you associate the process with a specific `ANYDATA` queue. When you create a propagation, you associate it with a specific source queue and destination queue. See "Creating an ANYDATA Queue to Stage LCRs" on page 9-7 for instructions.
3. Specify supplemental logging at each source database for any shared object. See "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for instructions.
4. At each database, create the required capture processes, propagations, and apply processes for your environment. You can create them in any order.
   - Create one or more capture processes at each database that will capture changes. Make sure each capture process uses rule sets that are appropriate for capturing changes. Do not start the capture processes you create. Oracle recommends that you use only one capture process for each source database. See "Creating a Capture Process" on page 9-1 for instructions.

     When you use a procedure in the `DBMS_STREAMS_ADM` package to add the capture process rules, it automatically runs the `PREPARE_TABLE_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, or `PREPARE_GLOBAL_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package for the specified table, specified schema, or entire database, respectively, if the capture process is a local capture process or a downstream capture process with a database link to the source database.

     You must run the appropriate procedure to prepare for instantiation manually if any of the following conditions is true:

     – You use the `DBMS_RULE_ADM` package to add or modify rules.
     – You use an existing capture process and do not add capture process rules for any shared object.
     – You use a downstream capture process with no database link to the source database.

     If you must prepare for instantiation manually, then see "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.

- Create all propagations that propagate the captured LCRs from a source queue to a destination queue. Make sure each propagation uses rule sets that are appropriate for propagating changes. See "Creating a Propagation that Propagates LCRs" on page 9-8 for instructions.
- Create one or more apply processes at each database that will apply changes. Make sure each apply process uses rule sets that are appropriate for applying changes. Do not start the apply processes you create. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.

5. Either instantiate, or set the instantiation SCN for, each database object for which changes are applied by an apply process. If the database objects do not exist at a destination database, then instantiate them using export/import, transportable tablespaces, or RMAN. If the database objects already exist at a destination database, then set the instantiation SCNs for them manually.

   - To instantiate database objects using export/import, first export them at the source database. Next, import them at the destination database. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for information. Also, see "Instantiating Objects in a Streams Replication Environment" on page 10-3 for information about instantiating objects using export/import, transportable tablespaces, and RMAN.

     If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`.

     If you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.

   - To set the instantiation SCN for a table, schema, or database manually, run the appropriate procedure or procedures in the `DBMS_APPLY_ADM` package at the destination database:

     – `SET_TABLE_INSTANTIATION_SCN`

     – `SET_SCHEMA_INSTANTIATION_SCN`

     – `SET_GLOBAL_INSTANTIATION_SCN`

     When you run one of these procedures, you must ensure that the shared objects at the destination database are consistent with the source database as of the instantiation SCN.

     If you run `SET_GLOBAL_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each schema at the destination database and for the tables owned by these schemas.

     If you run `SET_SCHEMA_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each table in the schema.

     If you set the `recursive` parameter to `true` in the `SET_GLOBAL_INSTANTIATION_SCN` procedure or the `SET_SCHEMA_INSTANTIATION_SCN` procedure, then a database link from the destination database to the source database is required. This database link must have the same name as the global name of the source database and must be accessible to the user who executes the procedure. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Alternatively, you can perform a metadata export/import to set the instantiation SCNs for existing database objects. If you choose this option, then make sure no rows are imported. Also, make sure the shared objects at all of the destination databases are consistent with the source database that performed the export at the time of the export. If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

6. Start each apply process you created in Step 4 using the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package.
7. Start each capture process you created in Step 4 using the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

When you are configuring the environment, remember that capture processes and apply processes are stopped when they are created, but propagations are scheduled to propagate LCRs immediately when they are created. The capture process must be created before the relevant objects are instantiated at a remote destination database. You must create the propagations and apply processes before starting the capture process, and you must instantiate the objects before running the whole stream.

See Also:

- Chapter 14, "Simple Single-Source Replication Example" and Chapter 15, "Single-Source Heterogeneous Replication Example" for detailed examples that set up single-source environments

Creating a New Streams Multiple-Source Environment

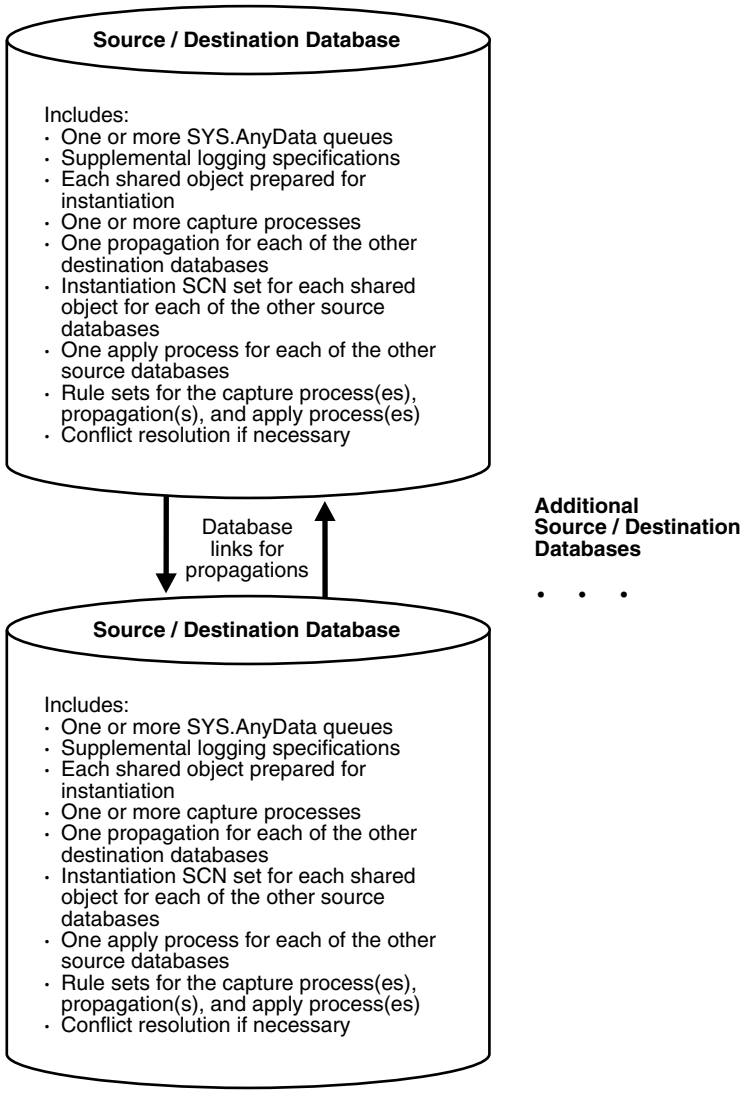
This section lists the general steps to perform when creating a new multiple-source Streams environment. A multiple-source environment is one in which there is more than one source database for any of the shared data.

This example uses the following terms:

- **Populated database:** A database that already contains the shared database objects before you create the new multiple-source environment. You must have at least one populated database to create the new Streams environment.
- **Export database:** A populated database on which you perform an export of the shared database objects. This export is used to instantiate the shared database objects at the import databases. You might not have an export database if all of the databases in the environment are populated databases.
- **Import database:** A database that does not contain the shared database objects before you create the new multiple-source environment. You instantiate the shared database objects at an import database by performing an import of these database objects. You might not have any import databases if all of the databases in the environment are populated databases.

Figure 7–2 shows an example multiple-source Streams environment.

Figure 7–2 Example Streams Multiple-Source Environment



You can create a Streams environment that is more complicated than the one shown in Figure 7–2. For example, a multiple-source Streams environment can use downstream capture and directed networks.

Complete the following steps to create a new multiple-source environment:

Note: Make sure no changes are made to the objects being shared at a database you are adding to the Streams environment until the instantiation at the database is complete.

---

1. Complete the necessary tasks to prepare each database in your environment for Streams:
   - Configure a Streams administrator
   - Set initialization parameters relevant to Streams
   - For each database that will run a capture process, prepare the database to run a capture process
   - Configure network connectivity and database links

   Some of these tasks might not be required at certain databases.

   See Also: *Oracle Streams Concepts and Administration* for more information about preparing a database for Streams

2. At each populated database, specify any necessary supplemental logging for the shared objects. See "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for instructions.
3. Create any necessary `ANYDATA` queues that do not already exist. When you create a capture process or apply process, you associate the process with a specific `ANYDATA` queue. When you create a propagation, you associate it with a specific source queue and destination queue. See "Creating an ANYDATA Queue to Stage LCRs" on page 9-7 for instructions.
4. At each database, create the required capture processes, propagations, and apply processes for your environment. You can create them in any order.
   - Create one or more capture processes at each database that will capture changes. Make sure each capture process uses rule sets that are appropriate for capturing changes. Do not start the capture processes you create. Oracle recommends that you use only one capture process for each source database. See "Creating a Capture Process" on page 9-1 for instructions.

     When you a procedure in the `DBMS_STREAMS_ADM` package to add the capture process rules, it automatically runs the `PREPARE_TABLE_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, or `PREPARE_GLOBAL_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package for the specified table, specified schema, or entire database, respectively, if the capture process is a local capture process or a downstream capture process with a database link to the source database.

     You must run the appropriate procedure to prepare for instantiation manually if any of the following conditions is true:
     - You use the `DBMS_RULE_ADM` package to add or modify rules.
     - You use an existing capture process and do not add capture process rules for any shared object.
     - You use a downstream capture process with no database link to the source database.

     If you must prepare for instantiation manually, then see "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.
   - Create all propagations that propagate the captured LCRs from a source queue to a destination queue. Make sure each propagation uses rule sets that are appropriate for propagating changes. See "Creating a Propagation that Propagates LCRs" on page 9-8 for instructions.

- Create one or more apply processes at each database that will apply changes. Make sure each apply process uses rule sets that are appropriate for applying changes. Do not start the apply processes you create. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.

After completing these steps, complete the steps in each of the following sections that apply to your environment. You might need to complete the steps in only one of these sections or in both of these sections:

- For each populated database, complete the steps in "Configuring Populated Databases When Creating a Multiple-Source Environment" on page 7-8. These steps are required only if your environment has more than one populated database.
- For each import database, complete the steps in "Adding Shared Objects to Import Databases When Creating a New Environment" on page 7-9.

Configuring Populated Databases When Creating a Multiple-Source Environment

After completing the steps in "Creating a New Streams Multiple-Source Environment" on page 7-5, complete the following steps for the populated databases if your environment has more than one populated database:

1. For each populated database, set the instantiation SCN at each of the other populated databases in the environment that will be a destination database of the populated source database. These instantiation SCNs must be set, and only the changes made at a particular populated database that are committed after the corresponding SCN for that database will be applied at another populated database.

   For each populated database, you can set these instantiation SCNs in one of the following ways:

   a. Perform a metadata only export of the shared objects at the populated database and import the metadata at each of the other populated databases. Such an import sets the required instantiation SCNs for the populated database at the other populated databases. Make sure no rows are imported. Also, make sure the shared objects at each populated database performing a metadata import are consistent with the populated database that performed the metadata export at the time of the export.

      If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

   b. Set the instantiation SCNs manually at each of the other populated databases. Do this for each of the shared objects. Make sure the shared objects at each populated database are consistent with the instantiation SCNs you set at that database. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Adding Shared Objects to Import Databases When Creating a New Environment

After completing the steps in "Creating a New Streams Multiple-Source Environment" on page 7-5, complete the following steps for the import databases:

1. Pick the populated database that you will use as the export database. Do not perform the instantiations yet.
2. For each import database, set the instantiation SCNs at all of the other databases in the environment that will be a destination database of the import database. In this case, the import database will be the source database for these destination databases. The databases where you set the instantiation SCNs can include populated databases and other import databases.
   a. If one or more schemas will be created at an import database during instantiation or by a subsequent shared DDL change, then run the `SET_GLOBAL_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for this import database at all of the other databases in the environment.
   b. If a schema exists at an import database, and one or more tables will be created in the schema during instantiation or by a subsequent shared DDL change, then run the `SET_SCHEMA_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for the schema at all of the other databases in the environment for the import database. Do this for each such schema.

   See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

   Because you are running these procedures before any tables are instantiated at the import databases, and because the local capture processes are configured already for these import databases, you will not need to run the `SET_TABLE_INSTANTIATION_SCN` procedure for each table created during the instantiation. Instantiation SCNs will be set automatically for these tables at all of the other databases in the environment that will be destination databases of the import database.
3. At the export database you chose in Step 1, perform an export of the shared objects. Next, perform an import of the shared objects at each import database. See "Instantiating Objects in a Streams Replication Environment" on page 10-3 and *Oracle Database Utilities* for information about using export/import.

   If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`.

   If you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.
4. For each populated database, except for the export database, set the instantiation SCNs at each import database that will be a destination database of the populated source database. These instantiation SCNs must be set, and only the changes made at a populated database that are committed after the corresponding SCN for that database will be applied at an import database.

You can set these instantiation SCNs in one of the following ways:

a. Perform a metadata only export at each populated database and import the metadata at each import database. Each import sets the required instantiation SCNs for the populated database at the import database. In this case, ensure that the shared objects at the import database are consistent with the populated database at the time of the export.

If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

b. For each populated database, set the instantiation SCN manually for each shared object at each import database. Make sure the shared objects at each import database are consistent with the populated database as of the corresponding instantiation SCN. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Complete the Multiple-Source Environment Configuration

Before completing the steps in this section, you should have completed the following tasks:

- "Creating a New Streams Multiple-Source Environment" on page 7-5
- "Configuring Populated Databases When Creating a Multiple-Source Environment" on page 7-8, if your environment has more than one populated database
- "Adding Shared Objects to Import Databases When Creating a New Environment" on page 7-9, if your environment has one or more import databases

When all of the previous configuration steps are finished, complete the following steps:

1. At each database, configure conflict resolution if conflicts are possible. See "Managing Streams Conflict Detection and Resolution" on page 9-22 for instructions.
2. Start each apply process in the environment using the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package.
3. Start each capture process the environment using the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

See Also: Chapter 16, "Multiple-Source Replication Example" for a detailed example that creates a multiple-source environment

8

Adding to a Streams Replication Environment

This chapter contains instructions for adding database objects and databases to an existing Streams replication environment.

This chapter contains these topics:

- Adding Shared Objects to an Existing Single-Source Environment
- Adding a New Destination Database to a Single-Source Environment
- Adding Shared Objects to an Existing Multiple-Source Environment
- Adding a New Database to an Existing Multiple-Source Environment

Note:

- The instructions in the following sections assume you will use the `DBMS_STREAMS_ADM` package to configure your Streams environment. If you use other packages, then extra steps might be necessary for each task.
- Certain types of database objects are not supported by Streams. When you add databases and objects to a Streams environment, ensure that no capture process attempts to capture changes to an unsupported database object. To list unsupported database objects, query the `DBA_STREAMS_UNSUPPORTED` data dictionary view.

See Also:

- Chapter 6, "Simple Streams Replication Configuration"
- Chapter 7, "Flexible Streams Replication Configuration"

Adding Shared Objects to an Existing Single-Source Environment

You add existing database objects to an existing single-source environment by adding the necessary rules to the appropriate capture processes, propagations, and apply processes. Before creating or altering capture or propagation rules in a running Streams environment, make sure any propagations or apply processes that will receive LCRs as a result of the new or altered rules are configured to handle these LCRs. That is, the propagations or apply processes should exist, and each one should be associated with rule sets that handle the LCRs appropriately. If these propagations and apply processes are not configured properly to handle these LCRs, then LCRs can be lost.

For example, suppose you want to add a table to a Streams environment that already captures, propagates, and applies changes to other tables. Assume only one capture process will capture changes to this table, and only one apply process will apply changes to this table. In this case, you must add one or more table rules to the following rule sets:

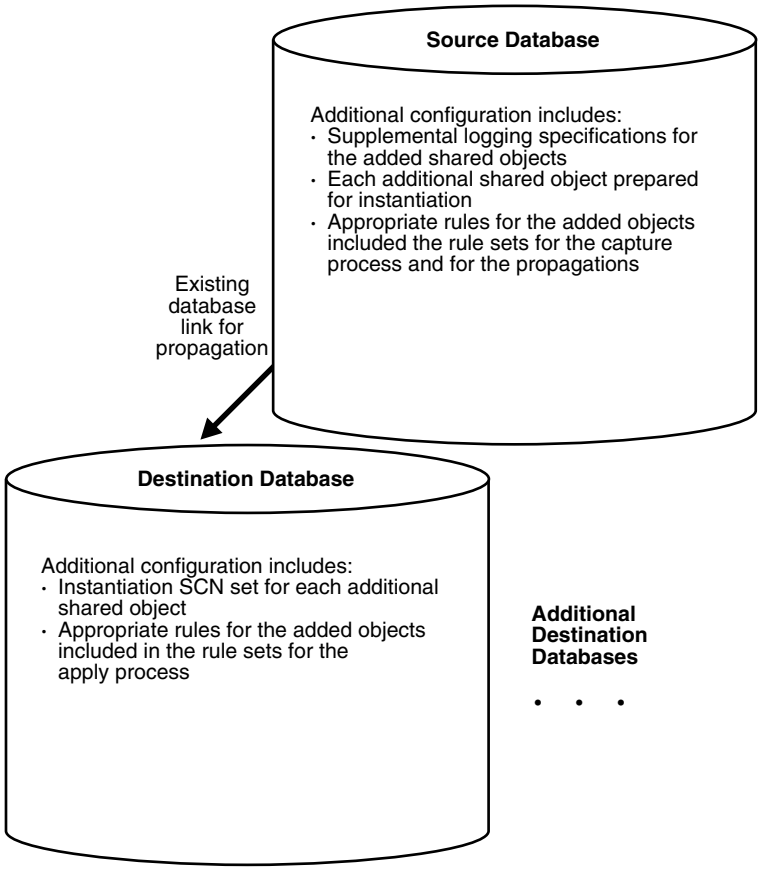
- The positive rule set for the apply process that will apply changes to the table
- The positive rule set for each propagation that will propagate changes to the table
- The positive rule set for the capture process that will capture changes to the table

If you perform administrative steps in the wrong order, you can lose LCRs. For example, if you add the rule to a capture process rule set first, without stopping the capture process, then the propagation will not propagate the changes if it does not have a rule that instructs it to do so, and the changes can be lost.

This example assumes that the shared database objects are read-only at the destination databases. If the shared objects are read/write at the destination databases, then the replication environment will not stay synchronized because Streams is not configured to replicate the changes made to the shared objects at the destination databases.

Figure 8–1 shows the additional configuration steps that must be completed to add shared database objects to a single-source Streams environment.

Figure 8–1 Example of Adding Shared Objects to a Single-Source Environment



To avoid losing LCRs, you should complete the configuration in the following order:

1. At each source database where shared objects are being added, specify supplemental logging for the added shared objects. See "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for instructions.
2. Either stop the capture process, one of the propagations, or the apply processes:
   - Use the `STOP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package to stop a capture process.
   - Use the `STOP_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to stop a propagation.
   - Use the `STOP_APPLY` procedure in the `DBMS_APPLY_ADM` package to stop an apply process.

   See Also: *Oracle Streams Concepts and Administration* for more information about completing these tasks

3. Add the relevant rules to the rule sets for the apply processes. To add rules to the rule set for an apply process, you can run one of the following procedures:

   - DBMS_STREAMS_ADM.ADD_TABLE_RULES
   - DBMS_STREAMS_ADM.ADD_SUBSET_RULES
   - DBMS_STREAMS_ADM.ADD_SCHEMA_RULES
   - DBMS_STREAMS_ADM.ADD_GLOBAL_RULES

   Excluding the ADD_SUBSET_RULES procedure, these procedures can add rules to the positive or negative rule set for an apply process. The ADD_SUBSET_RULES procedure can add rules only to the positive rule set for an apply process.

4. Add the relevant rules to the rule sets for the propagations. To add rules to the rule set for a propagation, you can run one of the following procedures:

   - DBMS_STREAMS_ADM.ADD_TABLE_PROPAGATION_RULES
   - DBMS_STREAMS_ADM.ADD_SUBSET_PROPAGATION_RULES
   - DBMS_STREAMS_ADM.ADD_SCHEMA_PROPAGATION_RULES
   - DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES

   Excluding the ADD_SUBSET_PROPAGATION_RULES procedure, these procedures can add rules to the positive or negative rule set for a propagation. The ADD_SUBSET_PROPAGATION_RULES procedure can add rules only to the positive rule set for a propagation.

5. Add the relevant rules to the rule sets used by the capture process. To add rules to a rule set for an existing capture process, you can run one of the following procedures and specify the existing capture process:

   - DBMS_STREAMS_ADM.ADD_TABLE_RULES
   - DBMS_STREAMS_ADM.ADD_SUBSET_RULES
   - DBMS_STREAMS_ADM.ADD_SCHEMA_RULES
   - DBMS_STREAMS_ADM.ADD_GLOBAL_RULES

   Excluding the ADD_SUBSET_RULES procedure, these procedures can add rules to the positive or negative rule set for a capture process. The ADD_SUBSET_RULES procedure can add rules only to the positive rule set for a capture process.

   When you a procedure in the DBMS_STREAMS_ADM package to add the capture process rules, it automatically runs the PREPARE_TABLE_INSTANTIATION, PREPARE_SCHEMA_INSTANTIATION, or PREPARE_GLOBAL_INSTANTIATION procedure in the DBMS_CAPTURE_ADM package for the specified table, specified schema, or entire database, respectively, if the capture process is a local capture process or a downstream capture process with a database link to the source database.

   You must run the appropriate procedure to prepare for instantiation manually if any of the following conditions is true:

   - You use DBMS_RULE_ADM to create or modify rules in a capture process rule set.
   - You do not add rules for the added objects to a capture process rule set, because the capture process already captures changes to these objects. In this case, rules for the objects can be added to propagations and apply processes in the environment, but not to the capture process.

- You use a downstream capture process with no database link to the source database.

If you must prepare for instantiation manually, then see "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.

6. At each destination database, either instantiate, or set the instantiation SCN for, each database object you are adding to the Streams environment. If the database objects do not exist at a destination database, then instantiate them using export/import, transportable tablespaces, or RMAN. If the database objects already exist at a destination database, then set the instantiation SCNs for them manually.

   - To instantiate database objects using export/import, first export them at the source database. Next, import them at the destination database. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for information. Also, see "Instantiating Objects in a Streams Replication Environment" on page 10-3 for information about instantiating objects using export/import, transportable tablespaces, and RMAN.

     If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`.

     If you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.

   - To set the instantiation SCN for a table, schema, or database manually, run the appropriate procedure or procedures in the `DBMS_APPLY_ADM` package at a destination database:

     – `SET_TABLE_INSTANTIATION_SCN`

     – `SET_SCHEMA_INSTANTIATION_SCN`

     – `SET_GLOBAL_INSTANTIATION_SCN`

     When you run one of these procedures at a destination database, you must ensure that every added object at the destination database is consistent with the source database as of the instantiation SCN.

     If you run `SET_GLOBAL_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each schema at the destination database and for the tables owned by these schemas.

     If you run `SET_SCHEMA_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each table in the schema.

     If you set the `recursive` parameter to `true` in the `SET_GLOBAL_INSTANTIATION_SCN` procedure or the `SET_SCHEMA_INSTANTIATION_SCN` procedure, then a database link from the destination database to the source database is required. This database link must have the same name as the global name of the source database and must be accessible to the user who executes the procedure. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

     Alternatively, you can perform a metadata export/import to set the instantiation SCNs for existing database objects. If you choose this option, then make sure no rows are imported. Also, make sure every added object at

the importing destination database is consistent with the source database that performed the export at the time of the export. If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

7. Start any Streams client you stopped in Step 2:
   - Use the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package to start a capture process.
   - Use the `START_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package start a propagation.
   - Use the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package to start an apply process.

   See Also: *Oracle Streams Concepts and Administration* for more information about completing these tasks

You must stop the capture process, disable one of the propagation jobs, or stop the apply process in Step 2 to ensure that the table or schema is instantiated before the first LCR resulting from the added rule(s) reaches the apply process. Otherwise, LCRs could be lost or could result in apply errors, depending on whether the apply process rule(s) have been added.

If you are certain that the added table is not being modified at the source database during this procedure, and that there are no LCRs for the table already in the stream or waiting to be captured, then you can perform Step 7 before Step 6 to reduce the amount of time that a Streams process or propagation job is stopped.

See Also: "Add Objects to an Existing Streams Replication Environment" on page 15-5 for a detailed example that adds objects to an existing single-source environment

Adding a New Destination Database to a Single-Source Environment

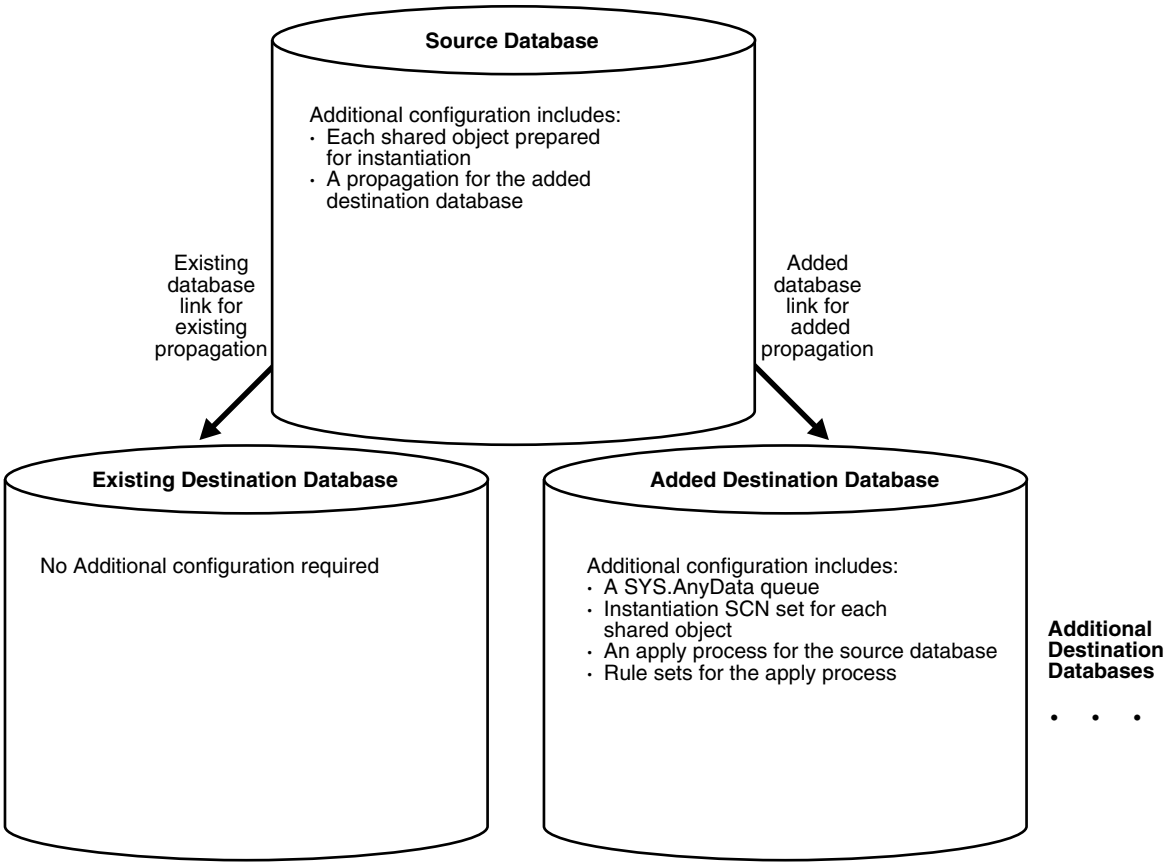
You add a destination database to an existing single-source environment by creating one or more new apply processes at the new destination database and, if necessary, configuring one or more propagations to propagate changes to the new destination database. You might also need to add rules to existing propagations in the stream that propagates to the new destination database.

As in the example that describes "Adding Shared Objects to an Existing Single-Source Environment" on page 8-2, before creating or altering propagation rules in a running Streams environment, make sure any propagations or apply processes that will receive LCRs as a result of the new or altered rules are configured to handle these LCRs. Otherwise, LCRs can be lost.

This example assumes that the shared database objects are read-only at the destination databases. If the shared objects are read/write at the destination databases, then the replication environment will not stay synchronized because Streams is not configured to replicate the changes made to the shared objects at the destination databases.

Figure 8–2 shows the additional configuration steps that must be completed to add a destination database to a single-source Streams environment.

Figure 8–2 Example of Adding a Destination to a Single-Source Environment



To avoid losing LCRs, you should complete the configuration in the following order:

1. Complete the necessary tasks to prepare each database in your environment for Streams:
   - Configure a Streams administrator
   - Set initialization parameters relevant to Streams
   - Configure network connectivity and database links

   Some of these tasks might not be required at certain databases.

   See Also: *Oracle Streams Concepts and Administration* for more information about preparing a database for Streams

2. Create any necessary `ANYDATA` queues that do not already exist at the destination database. When you create an apply process, you associate the apply process with a specific `ANYDATA` queue. See "Creating an ANYDATA Queue to Stage LCRs" on page 9-7 for instructions.
3. Create one or more apply processes at the new destination database to apply the changes from its source database. Make sure each apply process uses rule sets that are appropriate for applying changes. Do not start any of the apply processes at the new database. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.

Keeping the apply processes stopped prevents changes made at the source databases from being applied before the instantiation of the new database is completed, which would otherwise lead to incorrect data and errors.

4. Configure any necessary propagations to propagate changes from the source databases to the new destination database. Make sure each propagation uses rule sets that are appropriate for propagating changes. See "Creating a Propagation that Propagates LCRs" on page 9-8.

5. At the source database, prepare for instantiation each database object for which changes will be applied by an apply process at the new destination database. Run either the `PREPARE_TABLE_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, or `PREPARE_GLOBAL_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package for the specified table, specified schema, or entire database, respectively. See "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.

6. At the new destination database, either instantiate, or set the instantiation SCNs for, each database object for which changes will be applied by an apply process. If the database objects do not already exist at the new destination database, then instantiate them using export/import, transportable tablespaces, or RMAN. If the database objects exist at the new destination database, then set the instantiation SCNs for them.

   - To instantiate database objects using export/import, first export them at the source database. Next, import them at the destination database. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for information. Also, see "Instantiating Objects in a Streams Replication Environment" on page 10-3 for information about instantiating objects using export/import, transportable tablespaces, and RMAN.

     If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`.

     If you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.

   - To set the instantiation SCN for a table, schema, or database manually, run the appropriate procedure or procedures in the `DBMS_APPLY_ADM` package at the new destination database:

     – `SET_TABLE_INSTANTIATION_SCN`

     – `SET_SCHEMA_INSTANTIATION_SCN`

     – `SET_GLOBAL_INSTANTIATION_SCN`

     When you run one of these procedures, you must ensure that the shared objects at the new destination database are consistent with the source database as of the instantiation SCN.

     If you run `SET_GLOBAL_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each schema at the destination database and for the tables owned by these schemas.

     If you run `SET_SCHEMA_INSTANTIATION_SCN` at a destination database, then set the `recursive` parameter for this procedure to `true` so that the instantiation SCN also is set for each table in the schema.

If you set the `recursive` parameter to `true` in the `SET_GLOBAL_INSTANTIATION_SCN` procedure or the `SET_SCHEMA_INSTANTIATION_SCN` procedure, then a database link from the destination database to the source database is required. This database link must have the same name as the global name of the source database and must be accessible to the user who executes the procedure. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Alternatively, you can perform a metadata export/import to set the instantiation SCNs for existing database objects. If you choose this option, then make sure no rows are imported. Also, make sure the shared objects at the importing destination database are consistent with the source database that performed the export at the time of the export. If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

7. Start the apply processes you created in Step 3 using the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package.

See Also: "Add a Database to an Existing Streams Replication Environment" on page 15-6 for detailed example that adds a database to an existing single-source environment

Adding Shared Objects to an Existing Multiple-Source Environment

You add existing database objects to an existing multiple-source environment by adding the necessary rules to the appropriate capture processes, propagations, and apply processes.

This example uses the following terms:

- **Populated database:** A database that already contains the shared database objects being added to the multiple-source environment. You must have at least one populated database to add the objects to the environment.
- **Export database:** A populated database on which you perform an export of the database objects you are adding to the environment. This export is used to instantiate the added database objects at the import databases. You might not have an export database if all of the databases in the environment are populated databases.
- **Import database:** A database that does not contain the shared database objects before they are added to the multiple-source environment. You instantiate the shared database objects at an import database by performing an import of these database objects. You might not have any import databases if all of the databases in the environment are populated databases.

Before creating or altering capture or propagation rules in a running Streams environment, make sure any propagations or apply processes that will receive LCRs as a result of the new or altered rules are configured to handle these LCRs. That is, the propagations or apply processes should exist, and each one should be associated with rule sets that handle the LCRs appropriately. If these propagations and apply processes are not configured properly to handle these LCRs, then LCRs can be lost.

For example, suppose you want to add a new table to a Streams environment that already captures, propagates, and applies changes to other tables. Assume multiple capture processes in the environment will capture changes to this table, and multiple

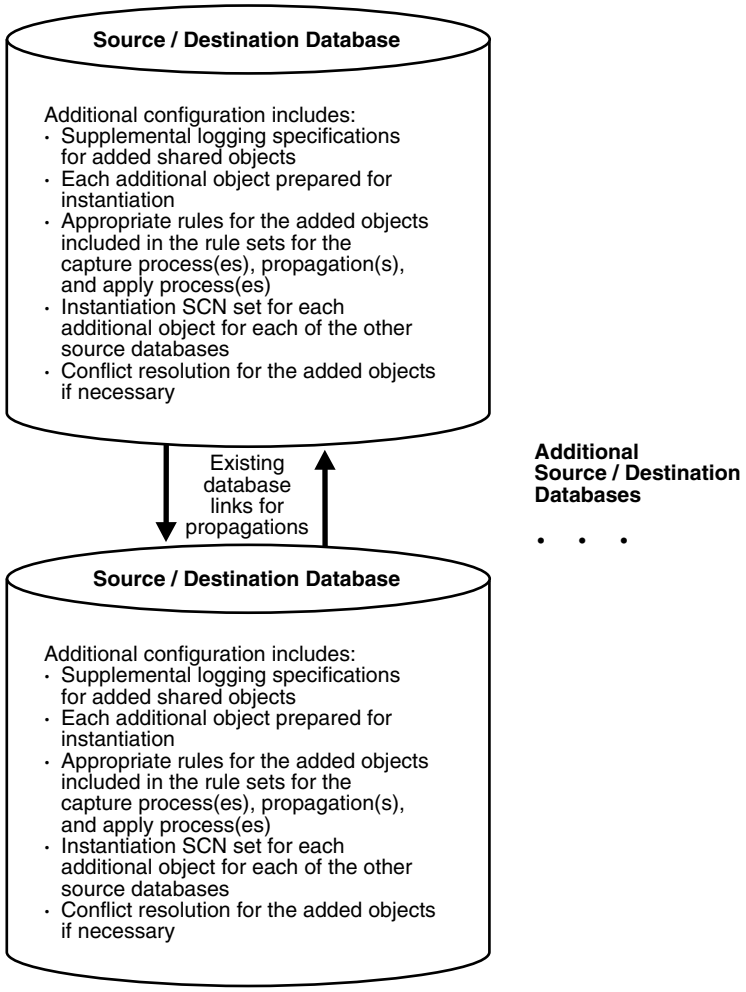
apply processes will apply changes to this table. In this case, you must add one or more table rules to the following rule sets:

- The positive rule set for each apply process that will apply changes to the table.
- The positive rule set for each propagation that will propagate changes to the table
- The positive rule set for each capture process that will capture changes to the table

If you perform administrative steps in the wrong order, you can lose LCRs. For example, if you add the rule to a capture process rule set first, without stopping the capture process, then the propagation will not propagate the changes if it does not have a rule that instructs it to do so, and the changes can be lost.

Figure 8–3 shows the additional configuration steps that must be completed to add shared database objects to a multiple-source Streams environment.

Figure 8–3 Example of Adding Shared Objects to a Multiple-Source Environment



To avoid losing LCRs, you should complete the configuration in the following order:

1. At each populated database, specify any necessary supplemental logging for the objects being added to the environment. See "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for instructions.
2. Either stop all of the capture processes that will capture changes to the added objects, stop all of the propagations that will propagate changes to the added objects, or stop all of the apply process that will apply changes to the added objects:
   - Use the `STOP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package to stop a capture process.
   - Use the `STOP_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to stop a propagation.
   - Use the `STOP_APPLY` procedure in the `DBMS_APPLY_ADM` package to stop an apply process.

   See Also: *Oracle Streams Concepts and Administration* for more information about completing these tasks
3. Add the relevant rules to the rule sets for the apply processes that will apply changes to the added objects. To add rules to the rule set for an apply process, you can run one of the following procedures:
   - `DBMS_STREAMS_ADM.ADD_TABLE_RULES`
   - `DBMS_STREAMS_ADM.ADD_SUBSET_RULES`
   - `DBMS_STREAMS_ADM.ADD_SCHEMA_RULES`
   - `DBMS_STREAMS_ADM.ADD_GLOBAL_RULES`

   Excluding the `ADD_SUBSET_RULES` procedure, these procedures can add rules to the positive or negative rule set for an apply process. The `ADD_SUBSET_RULES` procedure can add rules only to the positive rule set for an apply process.
4. Add the relevant rules to the rule sets for the propagations that will propagate changes to the added objects. To add rules to the rule set for a propagation, you can run one of the following procedures:
   - `DBMS_STREAMS_ADM.ADD_TABLE_PROPAGATION_RULES`
   - `DBMS_STREAMS_ADM.ADD_SUBSET_PROPAGATION_RULES`
   - `DBMS_STREAMS_ADM.ADD_SCHEMA_PROPAGATION_RULES`
   - `DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES`

   Excluding the `ADD_SUBSET_PROPAGATION_RULES` procedure, these procedures can add rules to the positive or negative rule set for a propagation. The `ADD_SUBSET_PROPAGATION_RULES` procedure can add rules only to the positive rule set for a propagation.

5. Add the relevant rules to the rule sets used by each capture process that will capture changes to the added objects. To add rules to a rule set for an existing capture process, you can run one of the following procedures and specify the existing capture process:

   - DBMS_STREAMS_ADM.ADD_TABLE_RULES
   - DBMS_STREAMS_ADM.ADD_SUBSET_RULES
   - DBMS_STREAMS_ADM.ADD_SCHEMA_RULES
   - DBMS_STREAMS_ADM.ADD_GLOBAL_RULES

   Excluding the ADD_SUBSET_RULES procedure, these procedures can add rules to the positive or negative rule set for a capture process. The ADD_SUBSET_RULES procedure can add rules only to the positive rule set for a capture process.

   When you a procedure in the DBMS_STREAMS_ADM package to add the capture process rules, it automatically runs the PREPARE_TABLE_INSTANTIATION, PREPARE_SCHEMA_INSTANTIATION, or PREPARE_GLOBAL_INSTANTIATION procedure in the DBMS_CAPTURE_ADM package for the specified table, specified schema, or entire database, respectively, if the capture process is a local capture process or a downstream capture process with a database link to the source database.

   You must run the appropriate procedure to prepare for instantiation manually if any of the following conditions is true:

   - You use DBMS_RULE_ADM to create or modify rules in a capture process rule set.
   - You do not add rules for the added objects to a capture process rule set, because the capture process already captures changes to these objects. In this case, rules for the objects can be added to propagations and apply processes in the environment, but not to the capture process.
   - You use a downstream capture process with no database link to the source database.

   If you must prepare for instantiation manually, then see "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.

After completing these steps, complete the steps in each of the following sections that apply to your environment. You might need to complete the steps in only one of these sections or in both of these sections:

- For each populated database, complete the steps in "Configuring Populated Databases When Adding Shared Objects" on page 8-13. These steps are required only if your environment has more than one populated database.
- For each import database, complete the steps in "Adding Shared Objects to Import Databases in an Existing Environment" on page 8-13.

Configuring Populated Databases When Adding Shared Objects

After completing the steps in "Adding Shared Objects to an Existing Multiple-Source Environment" on page 8-9, complete the following steps for each populated database if your environment has more than one populated database:

1. For each populated database, set the instantiation SCN for each added object at the other populated databases in the environment. These instantiation SCNs must be set, and only the changes made at a particular populated database that are committed after the corresponding SCN for that database will be applied at another populated database.

   For each populated database, you can set these instantiation SCNs for each added object in one of the following ways:

   a. Perform a metadata only export of the added objects at the populated database and import the metadata at each of the other populated databases. Such an import sets the required instantiation SCNs for the database at the other databases. Make sure no rows are imported. Also, make sure the shared objects at each of the other populated databases are consistent with the populated database that performed the export at the time of the export.

      If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

   b. Set the instantiation SCNs manually for the added objects at each of the other populated databases. Make sure every added object at each populated database is consistent with the instantiation SCNs you set at that database. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Adding Shared Objects to Import Databases in an Existing Environment

After completing the steps in "Adding Shared Objects to an Existing Multiple-Source Environment" on page 8-9, complete the following steps for the import databases:

1. Pick the populated database that you will use as the export database. Do not perform the instantiations yet.
2. For each import database, set the instantiation SCNs for the added objects at all of the other databases in the environment that will be a destination database of the import database. In this case, the import database will be the source database for these destination databases. The databases where you set the instantiation SCNs might be populated databases and other import databases.

   a. If one or more schemas will be created at an import database during instantiation or by a subsequent shared DDL change, then run the `SET_GLOBAL_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for this import database at all of the other databases in the environment.

   b. If a schema exists at an import database, and one or more tables will be created in the schema during instantiation or by a subsequent shared DDL change, then run the `SET_SCHEMA_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for the schema for this import database at each of the other databases in the environment. Do this for each such schema.

   See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Because you are running these procedures before any tables are instantiated at the import databases, and because the local capture processes are configured already for these import databases, you will not need to run the SET_TABLE_INSTANTIATION_SCN procedure for each table created during instantiation. Instantiation SCNs will be set automatically for these tables at all of the other databases in the environment that will be destination databases of the import database.

3. At the export database you chose in Step 1, perform an export of the shared objects. Next, perform an import of the shared objects at each import database. See "Instantiating Objects in a Streams Replication Environment" on page 10-3 and *Oracle Database Utilities* for information about using export/import.

   If you use the original Export utility, then set the OBJECT_CONSISTENT export parameter to y. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as FLASHBACK_SCN or FLASHBACK_TIME.

   If you use the original Import utility, then set the STREAMS_INSTANTIATION import parameter to y.

4. For each populated database, except for the export database, set the instantiation SCNs for the added objects at each import database that will be a destination database of the populated source database. These instantiation SCNs must be set, and only the changes made at a populated database that are committed after the corresponding SCN for that database will be applied at an import database.

   For each populated database, you can set these instantiation SCNs for the added objects in one of the following ways:

   a. Perform a metadata only export of the added objects at the populated database and import the metadata at each import database. Each import sets the required instantiation SCNs for the populated database at the import database. In this case, ensure that every added object at the import database is consistent with the populated database at the time of the export.

      If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

   b. Set the instantiation SCNs manually for the added objects at each import database. Make sure every added object at each import database is consistent with the populated database as of the corresponding instantiation SCN. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

Complete the Adding Objects to a Multiple-Source Environment Configuration

Before completing the configuration, you should have completed the following tasks:

- "Adding Shared Objects to an Existing Multiple-Source Environment" on page 8-9
- "Configuring Populated Databases When Adding Shared Objects" on page 8-13, if your environment has more than one populated database
- "Adding Shared Objects to Import Databases in an Existing Environment" on page 8-13, if your environment had import databases

When all of the previous configuration steps are finished, complete the following steps:

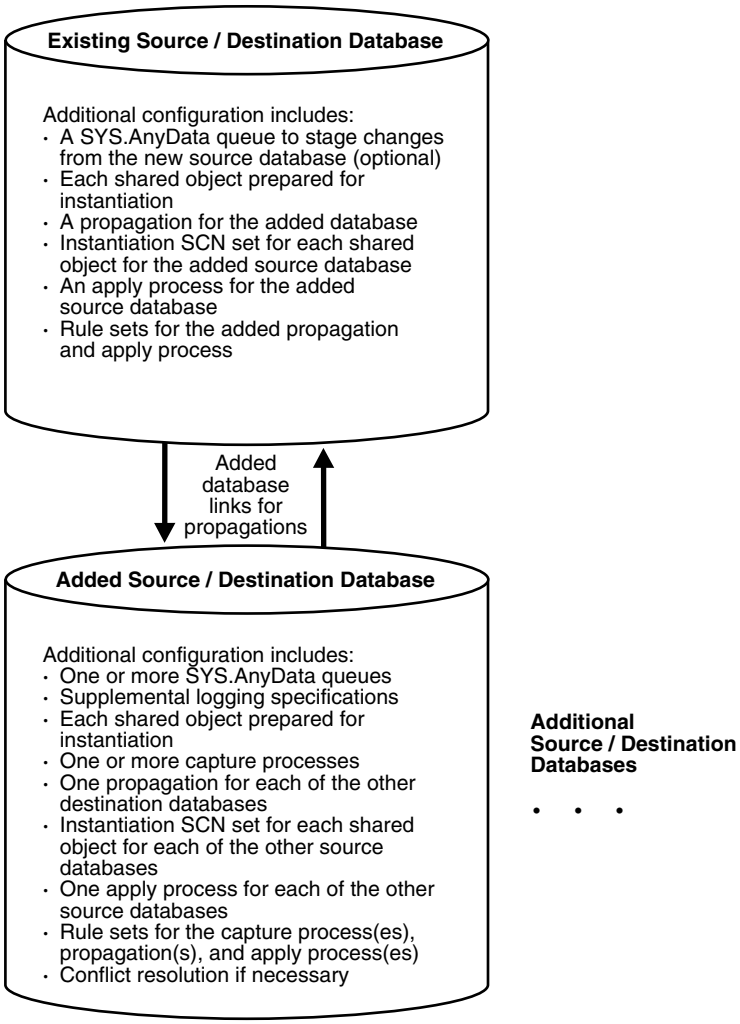
1. At each database, configure conflict resolution for the added database objects if conflicts are possible. See "Managing Streams Conflict Detection and Resolution" on page 9-22 for instructions.
2. Start each Streams client you stopped in Step 2 on page 8-11 in "Adding Shared Objects to an Existing Multiple-Source Environment":
   - Use the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package to start a capture process.
   - Use the `START_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to start a propagation.
   - Use the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package to start an apply process.

     See Also: *Oracle Streams Concepts and Administration* for more information about completing these tasks

Adding a New Database to an Existing Multiple-Source Environment

Figure 8–4 shows the additional configuration steps that must be completed to add a source/destination database to a multiple-source Streams environment.

Figure 8–4 Example of Adding a Database to a Multiple-Source Environment



Complete the following steps to add a new source/destination database to an existing multiple-source Streams environment:

> **Note:** Make sure no changes are made to the objects being shared at the database you are adding to the Streams environment until the instantiation at the database is complete.

1. Complete the necessary tasks to prepare each database in your environment for Streams:
   - Configure a Streams administrator
   - Set initialization parameters relevant to Streams
   - For each database that will run a capture process, prepare the database to run a capture process
   - Configure network connectivity and database links

   Some of these tasks might not be required at certain databases.

   See Also: *Oracle Streams Concepts and Administration* for more information about preparing a database for Streams

2. Create any necessary `ANYDATA` queues that do not already exist. When you create a capture process or apply process, you associate the process with a specific `ANYDATA` queue. When you create a propagation, you associate it with a specific source queue and destination queue. See "Creating an ANYDATA Queue to Stage LCRs" on page 9-7 for instructions.
3. Create one or more apply processes at the new database to apply the changes from its source databases. Make sure each apply process uses rule sets that are appropriate for applying changes. Do not start any apply process at the new database. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.

   Keeping the apply processes stopped prevents changes made at the source databases from being applied before the instantiation of the new database is completed, which would otherwise lead to incorrect data and errors.
4. If the new database will be a source database, then, at all databases that will be destination databases for the changes made at the new database, create one or more apply processes to apply changes from the new database. Make sure each apply process uses rule sets that are appropriate for applying changes. Do not start any of these new apply processes. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.
5. Configure propagations at the databases that will be source databases of the new database to send changes to the new database. Make sure each propagation uses rule sets that are appropriate for propagating changes. See "Creating a Propagation that Propagates LCRs" on page 9-8.
6. If the new database will be a source database, then configure propagations at the new database to send changes from the new database to each of its destination databases. Make sure each propagation uses rule sets that are appropriate for propagating changes. See "Creating a Propagation that Propagates LCRs" on page 9-8.
7. If the new database will be a source database, and the shared objects already exist at the new database, then specify any necessary supplemental logging for the shared objects at the new database. See "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3 for instructions.

8. At each source database for the new database, prepare for instantiation each database object for which changes will be applied by an apply process at the new database. Run either the PREPARE_TABLE_INSTANTIATION, PREPARE_SCHEMA_INSTANTIATION, or PREPARE_GLOBAL_INSTANTIATION procedure in the DBMS_CAPTURE_ADM package for the specified table, specified schema, or entire database, respectively. See "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.
9. If the new database will be a source database, then create one or more capture processes to capture the relevant changes. See "Creating a Capture Process" on page 9-1 for instructions. Oracle recommends that you use only one capture process for each source database.

   When you use a procedure in the DBMS_STREAMS_ADM package to add the capture process rules, it automatically runs the PREPARE_TABLE_INSTANTIATION, PREPARE_SCHEMA_INSTANTIATION, or PREPARE_GLOBAL_INSTANTIATION procedure in the DBMS_CAPTURE_ADM package for the specified table, specified schema, or entire database, respectively, if the capture process is a local capture process or a downstream capture process with a database link to the source database.

   You must run the appropriate procedure to prepare for instantiation manually if any of the following conditions is true:

   - You use the DBMS_RULE_ADM package to add or modify rules.
   - You use an existing capture process and do not add capture process rules for any shared object.
   - You use a downstream capture process with no database link to the source database.

   If you must prepare for instantiation manually, then see "Preparing Database Objects for Instantiation at a Source Database" on page 10-1 for instructions.
10. If the new database will be a source database, then start any capture process you created in Step 9 using the START_CAPTURE procedure in the DBMS_CAPTURE_ADM package.

After completing these steps, complete the steps in the appropriate section:

- If the objects that are to be shared with the new database already exist at the new database, then complete the steps in "Configuring Databases If the Shared Objects Already Exist at the New Database" on page 8-18.
- If the objects that are to be shared with the new database do not already exist at the new database, complete the steps in "Adding Shared Objects to a New Database" on page 8-20.

Configuring Databases If the Shared Objects Already Exist at the New Database

After completing the steps in "Adding a New Database to an Existing Multiple-Source Environment" on page 8-16, complete the following steps if the objects that are to be shared with the new database already exist at the new database:

1. For each source database of the new database, set the instantiation SCNs at the new database. These instantiation SCNs must be set, and only the changes made at a source database that are committed after the corresponding SCN for that database will be applied at the new database.

For each source database of the new database, you can set these instantiation SCNs in one of the following ways:

a. Perform a metadata only export of the shared objects at the source database and import the metadata at the new database. The import sets the required instantiation SCNs for the source database at the new database. Make sure no rows are imported. In this case, ensure that the shared objects at the new database are consistent with the source database at the time of the export.

If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

b. Set the instantiation SCNs manually at the new database for the shared objects. Make sure the shared objects at the new database are consistent with the source database as of the corresponding instantiation SCN. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

2. For the new database, set the instantiation SCNs at each destination database of the new database. These instantiation SCNs must be set, and only the changes made at the new source database that are committed after the corresponding SCN will be applied at a destination database. If the new database is not a source database, then do not complete this step.

You can set these instantiation SCNs for the new database in one of the following ways:

a. Perform a metadata only export at the new database and import the metadata at each destination database. Make sure no rows are imported. The import sets the required instantiation SCNs for the new database at each destination database. In this case, ensure that the shared objects at each destination database are consistent with the new database at the time of the export.

If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

b. Set the instantiation SCNs manually at each destination database for the shared objects. Make sure the shared objects at each destination database are consistent with the new database as of the corresponding instantiation SCN. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

3. At the new database, configure conflict resolution if conflicts are possible. See "Managing Streams Conflict Detection and Resolution" on page 9-22 for instructions.

4. Start the apply processes that you created at the new database in Step 3 on page 8-17 using the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package.

5. Start the apply processes that you created at each of the other destination databases in Step 4 on page 8-17. If the new database is not a source database, then do not complete this step.

Adding Shared Objects to a New Database

After completing the steps in "Adding a New Database to an Existing Multiple-Source Environment" on page 8-16, complete the following steps if the objects that are to be shared with the new database do not already exist at the new database:

1. If the new database is a source database for other databases, then, at each destination database of the new source database, set the instantiation SCNs for the new database.

   a. If one or more schemas will be created at the new database during instantiation or by a subsequent shared DDL change, then run the `SET_GLOBAL_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for the new database at each destination database of the new database.

   b. If a schema exists at the new database, and one or more tables will be created in the schema during instantiation or by a subsequent shared DDL change, then run the `SET_SCHEMA_INSTANTIATION_SCN` procedure in the `DBMS_APPLY_ADM` package for the schema at each destination database of the new database. Do this for each such schema.

   See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

   Because you are running these procedures before any tables are instantiated at the new database, and because the local capture process is configured already at the new database, you will not need to run the `SET_TABLE_INSTANTIATION_SCN` procedure for each table created during instantiation. Instantiation SCNs will be set automatically for these tables at all of the other databases in the environment that will be destination databases of the new database.

   If the new database will not be a source database, then do not complete this step, and continue with the next step.

2. Pick one source database from which to instantiate the shared objects at the new database using export/import. First, perform an export of the shared objects. Next, perform an import of the shared objects at the new database. See "Instantiating Objects in a Streams Replication Environment" on page 10-3 and *Oracle Database Utilities* for information about using export/import.

   If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`.

   If you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.

3. For each source database of the new database, except for the source database that performed the export for instantiation in Step 2, set the instantiation SCNs at the new database. These instantiation SCNs must be set, and only the changes made at a source database that are committed after the corresponding SCN for that database will be applied at the new database.

For each source database, you can set these instantiation SCNs in one of the following ways:

a. Perform a metadata only export at the source database and import the metadata at the new database. The import sets the required instantiation SCNs for the source database at the new database. In this case, ensure that the shared objects at the new database are consistent with the source database at the time of the export.

If you are sharing DML changes only, then table level export/import is sufficient. If you are sharing DDL changes also, then additional considerations apply. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for more information about performing a metadata export/import.

b. Set the instantiation SCNs manually at the new database for the shared objects. Make sure the shared objects at the new database are consistent with the source database as of the corresponding instantiation SCN. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29 for instructions.

4. At the new database, configure conflict resolution if conflicts are possible. See "Managing Streams Conflict Detection and Resolution" on page 9-22 for instructions.
5. Start the apply processes that you created in Step 3 on page 8-17 at the new database using the `START_APPLY` procedure in the `DBMS_APPLY_ADM` package.
6. Start the apply processes that you created in Step 4 on page 8-17 at each of the other destination databases. If the new database is not a source database, then do not complete this step.

Part III

Administering Streams Replication

This part describes managing and monitoring a Streams replication and contains the following chapters:

9

Managing Capture, Propagation, and Apply

This chapter contains instructions for managing Streams capture processes, propagations, and Streams apply processes in a Streams replication environment. This chapter also includes instructions for managing Streams tags, and for performing database point-in-time recovery at a destination database in a Streams environment.

This chapter contains these topics:

Managing Capture for Streams Replication

The following sections describe management tasks for a capture process in a Streams replication environment:

You also might need to perform other management tasks.

> **See Also:** *Oracle Streams Replication Administrator's Guide* for more information about managing a capture process

Creating a Capture Process

A capture process typically starts the process of replicating a database change by capturing the change, converting the change into a logical change record (LCR), and enqueuing the change into an `ANYDATA` queue. From there, the LCR can be propagated to other databases and applied at these database to complete the replication process.

You can create a capture process that captures changes to the local source database, or you can create a capture process that captures changes remotely at a downstream database. If a capture process runs on a downstream database, then redo data from the source database is copied to the downstream database, and the capture process captures changes in the redo data at the downstream database.

You can use any of the following procedures to create a local capture process:

- DBMS_STREAMS_ADM.ADD_TABLE_RULES
- DBMS_STREAMS_ADM.ADD_SUBSET_RULES
- DBMS_STREAMS_ADM.ADD_SCHEMA_RULES
- DBMS_STREAMS_ADM.ADD_GLOBAL_RULES
- DBMS_CAPTURE_ADM.CREATE_CAPTURE

> **Note:** To create a capture process, a user must be granted DBA role.

The following example runs the ADD_SCHEMA_RULES procedure in the DBMS_STREAMS_ADM package to create a local capture process:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name        => 'hr',
    streams_type       => 'capture',
    streams_name       => 'strep01_capture',
    queue_name         => 'strep01_queue',
    include_dml        => true,
    include_ddl        => true,
    include_tagged_lcr => false,
    source_database    => NULL,
    inclusion_rule     => true);
END;
/
```

Running this procedure performs the following actions:

- Creates a capture process named strep01_capture. The capture process is created only if it does not already exist. If a new capture process is created, then this procedure also sets the start SCN to the point in time of creation.
- Associates the capture process with an existing queue named strep01_queue.
- Creates a positive rule set and associates it with the capture process, if the capture process does not have a positive rule set, because the inclusion_rule parameter is set to true. The rule set uses the SYS.STREAMS$_EVALUATION_CONTEXT evaluation context. The rule set name is system generated.
- Creates two rules. One rule evaluates to TRUE for DML changes to the hr schema and the database objects in the hr schema, and the other rule evaluates to TRUE for DDL changes to the hr schema and the database objects in the hr schema. The rule names are system generated.
- Adds the two rules to the positive rule set associated with the capture process. The rules are added to the positive rule set because the inclusion_rule parameter is set to true.
- Specifies that the capture process captures a change in the redo log only if the change has a NULL tag, because the include_tagged_lcr parameter is set to false. This behavior is accomplished through the system-created rules for the capture process.

- Creates a capture process that captures local changes to the source database because the `source_database` parameter is set to `NULL`. For a local capture process, you can also specify the global name of the local database for this parameter.
- Prepares all of the database objects in the `hr` schema, and all of the database objects added to the `hr` schema in the future, for instantiation.

Attention: When a capture process is started or restarted, it might need to scan redo log files with a `FIRST_CHANGE#` value that is lower than start SCN. Removing required redo log files before they are scanned by a capture process causes the capture process to abort. You can query the `DBA_CAPTURE` data dictionary view to determine the first SCN, start SCN, and required checkpoint SCN. A capture process needs the redo log file that includes the required checkpoint SCN, and all subsequent redo log files.

See Also: *Oracle Streams Concepts and Administration* for more information about creating a capture process, including information about creating a downstream capture process, and for more information about the first SCN and start SCN for a capture process

Managing Supplemental Logging in a Streams Replication Environment

Supplemental logging must be specified for certain columns at a source database for changes to the columns to be applied successfully at a destination database. The following sections illustrate how to manage supplemental logging at a source database:

- Specifying Table Supplemental Logging Using Unconditional Log Groups
- Specifying Table Supplemental Logging Using Conditional Log Groups
- Dropping a Supplemental Log Group
- Specifying Database Supplemental Logging of Key Columns
- Dropping Database Supplemental Logging of Key Columns

Note:

- LOB, `LONG`, `LONG RAW`, and user-defined type columns cannot be part of a supplemental log group.
- In addition to the methods described in this section, supplemental logging can also be enabled when database objects are prepared for instantiation.

See Also:

- "Supplemental Logging for Streams Replication" on page 1-8 for information about when supplemental logging is required
- "Monitoring Supplemental Logging" on page 12-2
- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

Specifying Table Supplemental Logging Using Unconditional Log Groups

The following sections describe creating an unconditional log group:

- Specifying an Unconditional Supplemental Log Group for Primary Key Column(s)
- Specifying an Unconditional Supplemental Log Group for All Table Columns
- Specifying an Unconditional Supplemental Log Group that Includes Selected Columns

Specifying an Unconditional Supplemental Log Group for Primary Key Column(s) To specify an unconditional supplemental log group that only includes the primary key column(s) for a table, use an `ALTER TABLE` statement with the `PRIMARY KEY` option in the `ADD SUPPLEMENTAL LOG DATA` clause.

For example, the following statement adds the primary key column of the `hr.regions` table to an unconditional log group:

```
ALTER TABLE hr.regions ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
```

The log group has a system-generated name.

Specifying an Unconditional Supplemental Log Group for All Table Columns To specify an unconditional supplemental log group that includes all of the columns in a table, use an `ALTER TABLE` statement with the `ALL` option in the `ADD SUPPLEMENTAL LOG DATA` clause.

For example, the following statement adds all of the columns in the `hr.departments` table to an unconditional log group:

```
ALTER TABLE hr.regions ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

The log group has a system-generated name.

Specifying an Unconditional Supplemental Log Group that Includes Selected Columns To specify an unconditional supplemental log group that contains columns that you select, use an `ALTER TABLE` statement with the `ALWAYS` specification for the `ADD SUPPLEMENTAL LOG GROUP` clause.These log groups can include key columns, if necessary.

For example, the following statement adds the `department_id` column and the `manager_id` column of the `hr.departments` table to an unconditional log group named `log_group_dep_pk`:

```
ALTER TABLE hr.departments ADD SUPPLEMENTAL LOG GROUP log_group_dep_pk
  (department_id, manager_id) ALWAYS;
```

The `ALWAYS` specification makes this log group an unconditional log group.

Specifying Table Supplemental Logging Using Conditional Log Groups

The following sections describe creating a conditional log group:

- Specifying a Conditional Log Group Using the ADD SUPPLEMENTAL LOG DATA Clause
- Specifying a Conditional Log Group Using the ADD SUPPLEMENTAL LOG GROUP Clause

Specifying a Conditional Log Group Using the ADD SUPPLEMENTAL LOG DATA Clause You can use the following options in the ADD SUPPLEMENTAL LOG DATA clause of an ALTER TABLE statement:

- The FOREIGN KEY option creates a conditional log group that includes the foreign key column(s) in the table.
- The UNIQUE option creates a conditional log group that includes the unique key column(s) and bitmap index column(s) in the table.

If you specify more than one option in a single ALTER TABLE statement, then a separate conditional log group is created for each option.

For example, the following statement creates two conditional log groups:

```
ALTER TABLE hr.employees ADD SUPPLEMENTAL LOG DATA
  (UNIQUE, FOREIGN KEY) COLUMNS;
```

One conditional log group includes the unique key columns and bitmap index columns for the table, and the other conditional log group includes the foreign key columns for the table. Both log groups have a system-generated name.

> **Note:** Specifying the UNIQUE option does not enable supplemental logging of bitmap join index columns.

Specifying a Conditional Log Group Using the ADD SUPPLEMENTAL LOG GROUP Clause To specify a conditional supplemental log group that includes any columns you choose to add, you can use the ADD SUPPLEMENTAL LOG GROUP clause in the ALTER TABLE statement. To make the log group conditional, do not include the ALWAYS specification.

For example, suppose the min_salary and max_salary columns in the hr.jobs table are included in a column list for conflict resolution at a destination database. The following statement adds the min_salary and max_salary columns to a conditional log group named log_group_jobs_cr:

```
ALTER TABLE hr.jobs ADD SUPPLEMENTAL LOG GROUP log_group_jobs_cr
  (min_salary, max_salary);
```

Dropping a Supplemental Log Group

To drop a conditional or unconditional supplemental log group, use the DROP SUPPLEMENTAL LOG GROUP clause in the ALTER TABLE statement. For example, to drop a supplemental log group named log_group_jobs_cr, run the following statement:

```
ALTER TABLE hr.jobs DROP SUPPLEMENTAL LOG GROUP log_group_jobs_cr;
```

Specifying Database Supplemental Logging of Key Columns

You also have the option of specifying supplemental logging for all primary key, unique key, bitmap index, and foreign key columns in a source database. You might choose this option if you configure a capture process to capture changes to an entire database. To specify supplemental logging for all primary key, unique key, bitmap index, and foreign key columns in a source database, issue the following SQL statement:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA
   (PRIMARY KEY, UNIQUE, FOREIGN KEY) COLUMNS;
```

If your primary key, unique key, bitmap index, and foreign key columns are the same at all source and destination databases, then running this command at the source database provides the supplemental logging needed for primary key, unique key, bitmap index, and foreign key columns at all destination databases. When you specify the PRIMARY KEY option, all columns of a row's primary key are placed in the redo log file any time the table is modified (unconditional logging). When you specify the UNIQUE option, any columns in a row's unique key and bitmap index are placed in the redo log file if any column belonging to the unique key or bitmap index is modified (conditional logging). When you specify the FOREIGN KEY option, all columns of a row's foreign key are placed in the redo log file if any column belonging to the foreign key is modified (conditional logging).

You can omit one or more of these options. For example, if you do not want to supplementally log all of the foreign key columns in the database, then you can omit the FOREIGN KEY option, as in the following example:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA
   (PRIMARY KEY, UNIQUE) COLUMNS;
```

In additional to PRIMARY KEY, UNIQUE, and FOREIGN KEY, you can also use the ALL option. The ALL option specifies that, when a row is changed, all the columns of that row (except for LOB, LONG, LONG RAW, and user-defined type columns) are placed in the redo log file (unconditional logging).

Supplemental logging statements are cumulative. If you issue two consecutive ALTER DATABASE ADD SUPPLEMENTAL LOG DATA commands, each with a different identification key, then both keys are supplementally logged.

Note: Specifying the UNIQUE option does not enable supplemental logging of bitmap join index columns.

Dropping Database Supplemental Logging of Key Columns

To drop supplemental logging for all primary key, unique key, bitmap index, and foreign key columns in a source database, issue the ALTER DATABASE DROP SUPPLEMENTAL LOG DATA statement. To drop database supplemental logging for all primary key, unique key, bitmap index, and foreign key columns, issue the following SQL statement:

```
ALTER DATABASE DROP SUPPLEMENTAL LOG DATA
  (PRIMARY KEY, UNIQUE, FOREIGN KEY) COLUMNS;
```

Note: Dropping database supplemental logging of key columns does not affect any existing table-level supplemental log groups.

Managing Staging and Propagation for Streams Replication

The following sections describe management tasks for LCR staging and propagation in a Streams replication environment:

- Creating an ANYDATA Queue to Stage LCRs
- Creating a Propagation that Propagates LCRs

You also might need to perform other management tasks.

> **See Also:** *Oracle Streams Replication Administrator's Guide* for more information about managing message staging and propagation

Creating an ANYDATA Queue to Stage LCRs

In a Streams replication environment, ANYDATA queues stage LCRs that encapsulate captured changes. These queues can be used by capture processes, propagations, and apply processes as an LCR goes through a stream from a source database to a destination database.

You use the SET_UP_QUEUE procedure in the DBMS_STREAMS_ADM package to create an ANYDATA queue. This procedure enables you to specify the following for the ANYDATA queue it creates:

- The queue table for the queue
- A storage clause for the queue table
- The queue name
- A queue user that will be configured as a secure queue user of the queue and granted ENQUEUE and DEQUEUE privileges on the queue
- A comment for the queue

This procedure creates a queue that is both a secure queue and a transactional queue and starts the newly created queue.

For example, to create an ANYDATA queue named strep01_queue in the strmadmin schema with a queue table named strep01_queue_table, run the following procedure:

```
BEGIN
  DBMS_STREAMS_ADM.SET_UP_QUEUE(
    queue_table  => 'strmadmin.strep01_queue_table',
    queue_name   => 'strmadmin.strep01_queue');
END;
/
```

You can also use procedures in the DBMS_AQADM package to create an ANYDATA queue.

> **See Also:** *Oracle Streams Concepts and Administration* for information about managing ANYDATA queues

Creating a Propagation that Propagates LCRs

To replicate LCRs between databases, you must propagate the LCRs from the database where they were first staged in a queue to the database where they are applied. To accomplish this goal, you can use any number of separate propagations.

You can use any of the following procedures to create a propagation:

- DBMS_STREAMS_ADM.ADD_TABLE_PROPAGATION_RULES
- DBMS_STREAMS_ADM.ADD_SUBSET_PROPAGATION_RULES
- DBMS_STREAMS_ADM.ADD_SCHEMA_PROPAGATION_RULES
- DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES
- DBMS_PROPAGATION_ADM.CREATE_PROPAGATION

The following tasks must be completed before you create a propagation:

- Create a source queue and a destination queue for the propagation, if they do not exist. See "Creating an ANYDATA Queue to Stage LCRs" on page 9-7 for instructions.
- Create a database link between the database containing the source queue and the database containing the destination queue. See *Oracle Streams Concepts and Administration* for more information about creating database links for propagations.

The following example runs the ADD_SCHEMA_PROPAGATION_RULES procedure in the DBMS_STREAMS_ADM package to create a propagation:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_PROPAGATION_RULES(
    schema_name             => 'hr',
    streams_name            => 'strep01_propagation',
    source_queue_name       => 'strmadmin.strep01_queue',
    destination_queue_name  => 'strmadmin.strep02_queue@rep2.net',
    include_dml             => true,
    include_ddl             => true,
    include_tagged_lcr      => false,
    source_database         => 'rep1.net',
    inclusion_rule          => true,
    queue_to_queue          => true);
END;
/
```

Running this procedure performs the following actions:

- Creates a propagation named strep01_propagation. The propagation is created only if it does not already exist.
- Specifies that the propagation propagates LCRs from strep01_queue in the current database to strep02_queue in the rep2.net database.
- Specifies that the propagation uses the rep2.net database link to propagate the LCRs, because the destination_queue_name parameter contains @rep2.net.

- Creates a positive rule set and associates it with the propagation, if the propagation does not have a positive rule set, because the `inclusion_rule` parameter is set to `true`. The rule set uses the evaluation context `SYS.STREAMS$_EVALUATION_CONTEXT`. The rule set name is system generated.
- Creates two rules. One rule evaluates to `TRUE` for row LCRs that contain the results of DML changes to the tables in the `hr` schema, and the other rule evaluates to `TRUE` for DDL LCRs that contain DDL changes to the `hr` schema or to the database objects in the `hr` schema. The rule names are system generated.
- Adds the two rules to the positive rule set associated with the propagation. The rules are added to the positive rule setbecause the `inclusion_rule` parameter is set to `true`.
- Specifies that the propagation propagates an LCR only if it has a `NULL` tag, because the `include_tagged_lcr` parameter is set to `false`. This behavior is accomplished through the system-created rules for the propagation.
- Specifies that the source database for the LCRs being propagated is `rep1.net`, which might or might not be the current database. This propagation does not propagate LCRs in the source queue that have a different source database.
- Creates a propagation job for the queue-to-queue propagation.

Note: To use queue-to-queue propagation, the compatibility level must be 10.2.0 or higher for each database that contains a queue involved in the propagation.

See Also: *Oracle Streams Concepts and Administration* for information about creating propagations

Managing Apply for Streams Replication

The following sections describe management tasks for an apply process in a Streams replication environment:

- Creating an Apply Process that Applies LCRs
- Managing the Substitute Key Columns for a Table
- Managing a DML Handler
- Managing a DDL Handler
- Using Virtual Dependency Definitions
- Managing Streams Conflict Detection and Resolution

You also might need to perform other management tasks.

See Also: *Oracle Streams Concepts and Administration* for more information about managing an apply process

Creating an Apply Process that Applies LCRs

When an apply process applies an LCR or sends an LCR to an apply handler that executes it, the replication process for the LCR is complete. That is, the database change that is encapsulated in the LCR is shared with the database where the LCR is applied.

You can use any of the following procedures to create an apply process that applies LCRs:

- DBMS_STREAMS_ADM.ADD_TABLE_RULES
- DBMS_STREAMS_ADM.ADD_SUBSET_RULES
- DBMS_STREAMS_ADM.ADD_SCHEMA_RULES
- DBMS_STREAMS_ADM.ADD_GLOBAL_RULES
- DBMS_APPLY_ADM.CREATE_APPLY

Note: To create an apply process, a user must be granted DBA role.

Before you create an apply process, create an ANYDATA queue to associate with the apply process, if one does not exist.

The following example runs the ADD_SCHEMA_RULES procedure in the DBMS_STREAMS_ADM package to create an apply process:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name        => 'hr',
    streams_type       => 'apply',
    streams_name       => 'strep01_apply',
    queue_name         => 'strep02_queue',
    include_dml        => true,
    include_ddl        => true,
    include_tagged_lcr => false,
    source_database    => 'rep1.net',
    inclusion_rule     => true);
END;
/
```

Running this procedure performs the following actions:

- Creates an apply process named strep01_apply that applies captured LCRs to the local database. The apply process is created only if it does not already exist. To create an apply process that applies user-enqueued LCRs, you must use the CREATE_CAPTURE procedure in the DBMS_CAPTURE_ADM package.
- Associates the apply process with an existing queue named strep02_queue.
- Creates a positive rule set and associates it with the apply process, if the apply process does not have a positive rule set, because the inclusion_rule parameter is set to true. The rule set uses the SYS.STREAMS$_EVALUATION_CONTEXT evaluation context. The rule set name is system generated.
- Creates two rules. One rule evaluates to TRUE for row LCRs that contain the results of DML changes to the tables in the hr schema, and the other rule evaluates to TRUE for DDL LCRs that contain DDL changes to the hr schema or to the database objects in the hr schema. The rule names are system generated.

- Adds the rules to the positive rule set associated with the apply process because the `inclusion_rule` parameter is set to `true`.
- Sets the `apply_tag` for the apply process to a value that is the hexadecimal equivalent of `'00'` (double zero). Redo entries generated by the apply process have a tag with this value.
- Specifies that the apply process applies an LCR only if it has a `NULL` tag, because the `include_tagged_lcr` parameter is set to `false`. This behavior is accomplished through the system-created rule for the apply process.
- Specifies that the LCRs applied by the apply process originate at the `rep1.net` source database. The rules in the apply process rule sets determine which LCRs are dequeued by the apply process. If the apply process dequeues an LCR with a source database that is different than `rep1.net`, then an error is raised.

Note: Depending on the configuration of the apply process you create, supplemental logging might be required at the source database on columns in the tables for which an apply process applies changes.

See Also: *Oracle Streams Concepts and Administration* for information about creating apply processes

Managing the Substitute Key Columns for a Table

This section contains instructions for setting and removing the substitute key columns for a table.

See Also:

- "Substitute Key Columns" on page 1-19
- "Displaying the Substitute Key Columns Specified at a Destination Database" on page 12-7

Setting Substitute Key Columns for a Table

When an apply process applies changes to a table, substitute key columns can either replace the primary key columns for a table that has a primary key or act as the primary key columns for a table that does not have a primary key. Set the substitute key columns for a table using the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package. This setting applies to all of the apply processes that apply local changes to the database.

For example, to set the substitute key columns for the `hr.employees` table to the `first_name`, `last_name`, and `hire_date` columns, replacing the `employee_id` column, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_KEY_COLUMNS(
    object_name         => 'hr.employees',
    column_list         => 'first_name,last_name,hire_date');
END;
/
```

Note:

- You must specify an unconditional supplemental log group at the source database for all of the columns specified as substitute key columns in the `column_list` or `column_table` parameter at the destination database. In this example, you would specify an unconditional supplemental log group including the `first_name`, `last_name`, and `hire_date` columns in the `hr.employees` table.
- If an apply process applies changes to a remote non-Oracle database, then it can use different substitute key columns for the same table. You can run the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package to specify substitute key columns for changes that will be applied to a remote non-Oracle database by setting the `apply_database_link` parameter to a non-`NULL` value.

See Also:

- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3
- "Apply Process Configuration in an Oracle to Non-Oracle Environment" on page 5-3 for information about setting a setting key columns for a table in a remote non-Oracle database

Removing the Substitute Key Columns for a Table

You remove the substitute key columns for a table by specifying `NULL` for the `column_list` or `column_table` parameter in the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package. If the table has a primary key, then the table's primary key is used by any apply process for local changes to the database after you remove the substitute primary key.

For example, to remove the substitute key columns for the `hr.employees` table, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_KEY_COLUMNS(
    object_name  => 'hr.employees',
    column_list  => NULL);
END;
/
```

Managing a DML Handler

This section contains instructions for creating, setting, and removing a DML handler.

See Also: "Apply Processing Options for LCRs" on page 1-12

Creating a DML Handler

A DML handler must have the following signature:

```
PROCEDURE user_procedure (
   parameter_name   IN  ANYDATA);
```

Here, *user_procedure* stands for the name of the procedure and *parameter_name* stands for the name of the parameter passed to the procedure. The parameter passed to the procedure is an ANYDATA encapsulation of a row LCR.

The following restrictions apply to the user procedure:

- Do not execute COMMIT or ROLLBACK statements. Doing so can endanger the consistency of the transaction that contains the LCR.
- If you are manipulating a row using the EXECUTE member procedure for the row LCR, then do not attempt to manipulate more than one row in a row operation. You must construct and execute manually any DML statements that manipulate more than one row.
- If the command type is UPDATE or DELETE, then row operations resubmitted using the EXECUTE member procedure for the LCR must include the entire key in the list of old values. The key is the primary key or the smallest unique key that has at least one NOT NULL column, unless a substitute key has been specified by the SET_KEY_COLUMNS procedure. If there is no specified key, then the key consists of all non LOB, non LONG, and non LONG RAW columns.
- If the command type is INSERT, then row operations resubmitted using the EXECUTE member procedure for the LCR should include the entire key in the list of new values. Otherwise, duplicate rows are possible. The key is the primary key or the smallest unique key that has at least one NOT NULL column, unless a substitute key has been specified by the SET_KEY_COLUMNS procedure. If there is no specified key, then the key consists of all non LOB, non LONG, and non LONG RAW columns.

A DML handler can be used for any customized processing of row LCRs. For example, the handler can modify an LCR and then execute it using the EXECUTE member procedure for the LCR. When you execute a row LCR in a DML handler, the apply process applies the LCR without calling the DML handler again.

You can also use a DML handler for recording the history of DML changes. For example, a DML handler can insert information about an LCR it processes into a table and then apply the LCR using the EXECUTE member procedure. To create such a DML handler, first create a table to hold the history information:

```
CREATE TABLE strmadmin.history_row_lcrs(
  timestamp             DATE,
  source_database_name  VARCHAR2(128),
  command_type          VARCHAR2(30),
  object_owner          VARCHAR2(32),
  object_name           VARCHAR2(32),
  tag                   RAW(10),
  transaction_id        VARCHAR2(10),
  scn                   NUMBER,
  commit_scn            NUMBER,
  old_values            SYS.LCR$_ROW_LIST,
  new_values            SYS.LCR$_ROW_LIST)
    NESTED TABLE old_values STORE AS old_values_ntab
    NESTED TABLE new_values STORE AS new_values_ntab;
```

```
CREATE OR REPLACE PROCEDURE history_dml(in_any IN ANYDATA)
 IS
  lcr   SYS.LCR$_ROW_RECORD;
  rc    PLS_INTEGER;
 BEGIN
  -- Access the LCR
  rc := in_any.GETOBJECT(lcr);
  -- Insert information about the LCR into the history_row_lcrs table
  INSERT INTO strmadmin.history_row_lcrs VALUES
   (SYSDATE, lcr.GET_SOURCE_DATABASE_NAME(), lcr.GET_COMMAND_TYPE(),
    lcr.GET_OBJECT_OWNER(), lcr.GET_OBJECT_NAME(), lcr.GET_TAG(),
    lcr.GET_TRANSACTION_ID(), lcr.GET_SCN(), lcr.GET_COMMIT_SCN,
    lcr.GET_VALUES('old'), lcr.GET_VALUES('new', 'n'));
  --  Apply row LCR
  lcr.EXECUTE(true);
END;
/
```

Note:

- You must specify an unconditional supplemental log group at the source database for any columns needed by a DML handler at the destination database. This example DML handler does not require any additional supplemental logging because it simply records information about the row LCR and does not manipulate the row LCR in any other way.
- To test a DML handler before using it, or to debug a DML handler, you can construct row LCRs and run the DML handler procedure outside the context of an apply process.

See Also:

- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3
- "Executing Row LCRs" on page 11-7 for an example that constructs and executes row LCRs outside the context of an apply process
- Chapter 11, "Managing Logical Change Records (LCRs)" for information about and restrictions regarding DML handlers and LOB, `LONG`, and `LONG RAW` datatypes
- *Oracle Streams Concepts and Administration* for an example of a precommit handler that can be used with this DML handler to record commit information for applied transactions

Setting a DML Handler

A DML handler processes each row LCR dequeued by any apply process that contains a specific operation on a specific table. You can specify multiple DML handlers on the same table, to handle different operations on the table. All apply processes that apply changes to the specified table in the local database use the specified DML handler.

Set the DML handler using the `SET_DML_HANDLER` procedure in the `DBMS_APPLY_ADM` package. For example, the following procedure sets the DML handler for `UPDATE` operations on the `hr.locations` table. Therefore, when any apply process that applies changes locally dequeues a row LCR containing an `UPDATE` operation on the

hr.locations table, the apply process sends the row LCR to the history_dml PL/SQL procedure in the strmadmin schema for processing. The apply process does not apply a row LCR containing such a change directly.

In this example, the apply_name parameter is set to NULL. Therefore, the DML handler is a general DML handler that is used by all of the apply processes in the database.

```
BEGIN
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name         => 'hr.locations',
    object_type         => 'TABLE',
    operation_name      => 'UPDATE',
    error_handler       => false,
    user_procedure      => 'strmadmin.history_dml',
    apply_database_link => NULL,
    apply_name          => NULL);
END;
/
```

Note: If an apply process applies changes to a remote non-Oracle database, then it can use a different DML handler for the same table. You can run the SET_DML_HANDLER procedure in the DBMS_APPLY_ADM package to specify a DML handler for changes that will be applied to a remote non-Oracle database by setting the apply_database_link parameter to a non-NULL value.

See Also: "DML Handlers in an Oracle to Non-Oracle Heterogeneous Environment" on page 5-4

Unsetting a DML Handler

You unset a DML handler using the SET_DML_HANDLER procedure in the DBMS_APPLY_ADM package. When you run that procedure, set the user_procedure parameter to NULL for a specific operation on a specific table. After the DML handler is unset, any apply process that applies changes locally will apply a row LCR containing such a change directly.

For example, the following procedure unsets the DML handler for UPDATE operations on the hr.locations table:

```
BEGIN
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name    => 'hr.locations',
    object_type    => 'TABLE',
    operation_name => 'UPDATE',
    error_handler  => false,
    user_procedure => NULL,
    apply_name     => NULL);
END;
/
```

Managing a DDL Handler

This section contains instructions for creating, specifying, and removing the DDL handler for an apply process.

Note: All applied DDL LCRs commit automatically. Therefore, if a DDL handler calls the `EXECUTE` member procedure of a DDL LCR, then a commit is performed automatically.

See Also:

- "Apply Processing Options for LCRs" on page 1-12
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `EXECUTE` member procedure for LCR types

Creating a DDL Handler for an Apply Process

A DDL handler must have the following signature:

```
PROCEDURE handler_procedure (
   parameter_name   IN  ANYDATA);
```

Here, *`handler_procedure`* stands for the name of the procedure and *`parameter_name`* stands for the name of the parameter passed to the procedure. The parameter passed to the procedure is an `ANYDATA` encapsulation of a DDL LCR.

A DDL handler can be used for any customized processing of DDL LCRs. For example, the handler can modify the LCR and then execute it using the `EXECUTE` member procedure for the LCR. When you execute a DDL LCR in a DDL handler, the apply process applies the LCR without calling the DDL handler again.

You can also use a DDL handler to record the history of DDL changes. For example, a DDL handler can insert information about an LCR it processes into a table and then apply the LCR using the `EXECUTE` member procedure.

To create such a DDL handler, first create a table to hold the history information:

```
CREATE TABLE strmadmin.history_ddl_lcrs(
  timestamp            DATE,
  source_database_name  VARCHAR2(128),
  command_type          VARCHAR2(30),
  object_owner          VARCHAR2(32),
  object_name           VARCHAR2(32),
  object_type           VARCHAR2(18),
  ddl_text              CLOB,
  logon_user            VARCHAR2(32),
  current_schema        VARCHAR2(32),
  base_table_owner      VARCHAR2(32),
  base_table_name       VARCHAR2(32),
  tag                   RAW(10),
  transaction_id        VARCHAR2(10),
  scn                   NUMBER);
```

```
CREATE OR REPLACE PROCEDURE history_ddl(in_any IN ANYDATA)
 IS
   lcr       SYS.LCR$_DDL_RECORD;
   rc        PLS_INTEGER;
   ddl_text  CLOB;
 BEGIN
   -- Access the LCR
   rc := in_any.GETOBJECT(lcr);
   DBMS_LOB.CREATETEMPORARY(ddl_text, true);
   lcr.GET_DDL_TEXT(ddl_text);
   --  Insert DDL LCR information into history_ddl_lcrs table
   INSERT INTO strmadmin.history_ddl_lcrs VALUES(
     SYSDATE, lcr.GET_SOURCE_DATABASE_NAME(), lcr.GET_COMMAND_TYPE(),
     lcr.GET_OBJECT_OWNER(), lcr.GET_OBJECT_NAME(), lcr.GET_OBJECT_TYPE(),
     ddl_text, lcr.GET_LOGON_USER(), lcr.GET_CURRENT_SCHEMA(),
     lcr.GET_BASE_TABLE_OWNER(), lcr.GET_BASE_TABLE_NAME(), lcr.GET_TAG(),
     lcr.GET_TRANSACTION_ID(), lcr.GET_SCN());
   --  Apply DDL LCR
   lcr.EXECUTE();
   -- Free temporary LOB space
   DBMS_LOB.FREETEMPORARY(ddl_text);
END;
/
```

Setting the DDL Handler for an Apply Process

A DDL handler processes all DDL LCRs dequeued by an apply process. Set the DDL handler for an apply process using the `ddl_handler` parameter in the `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package. For example, the following procedure sets the DDL handler for an apply process named `strep01_apply` to the `history_ddl` procedure in the `strmadmin` schema.

```
BEGIN
  DBMS_APPLY_ADM.ALTER_APPLY(
    apply_name  => 'strep01_apply',
    ddl_handler => 'strmadmin.history_ddl');
END;
/
```

Removing the DDL Handler for an Apply Process

A DDL handler processes all DDL LCRs dequeued by an apply process. You remove the DDL handler for an apply process by setting the `remove_ddl_handler` parameter to `true` in the `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package. For example, the following procedure removes the DDL handler from an apply process named `strep01_apply`.

```
BEGIN
  DBMS_APPLY_ADM.ALTER_APPLY(
    apply_name         => 'strep01_apply',
    remove_ddl_handler => true);
END;
/
```

Using Virtual Dependency Definitions

A virtual dependency definition is a description of a dependency that is used by an apply process to detect dependencies between transactions being applied at a destination database. Virtual dependency definitions are useful when apply process parallelism is greater than 1 and dependencies are not described by constraints in the data dictionary at the destination database. There are two types of virtual dependency definitions: value dependencies and object dependencies.

A value dependency defines a table constraint, such as a unique key, or a relationship between the columns of two or more tables. An object dependency defines a parent-child relationship between two objects at a destination database.

The following sections describe using virtual dependency definitions:

- Setting and Unsetting Value Dependencies
- Creating and Dropping Object Dependencies

> **See Also:** "Apply Processes and Dependencies" on page 1-14 for more information about virtual dependency definitions

Setting and Unsetting Value Dependencies

Use the `SET_VALUE_DEPENDENCY` procedure in the `DBMS_APPLY_ADM` package to set or unset a value dependency. The following sections describe scenarios for using value dependencies:

- Schema Differences and Value Dependencies
- Undefined Constraints at the Destination Database and Value Dependencies

Schema Differences and Value Dependencies This scenario involves an environment that shares many tables between a source database and destination database, but the schema that owns the tables is different at these two databases. Also, in this replication environment, the source database is in the United States and the destination database is in England. A design firm uses dozens of tables to describe product designs, but the tables use United States measurements (inches, feet, and so on) in the source database and metric measurements in the destination database. The name of the schema that owns the database objects at the source database is `us_designs`, while the name of the schema at the destination database is `uk_designs`. Therefore, the schema name of the shared database objects must be changed before apply, and all of the measurements must be converted from United States measurements to metric measurements. Both databases use the same constraints to enforce dependencies between database objects.

Rule-based transformations could make the required changes, but the goal is to apply multiple LCRs in parallel. Rule-based transformations must apply LCRs serially. So, a DML handler is configured at the destination database to make the required changes to the LCRs, and apply process parallelism is set to 5. In this environment, the destination database has no information about the schema `us_designs` in the LCRs being sent from the source database. Because an apply process calculates dependencies before passing LCRs to apply handlers, the apply process must be informed about the dependencies between LCRs. Value dependencies can be used to describe these dependencies.

In this scenario, suppose a number of tables describe different designs, and each of these tables has a primary key. One of these tables is `design_53`, and the primary key column is `key_53`. Also, a table named `all_designs_summary` includes a summary of all of the individual designs, and this table has a foreign key column for each design

table. The all_designs_summary includes a key_53 column, which is a foreign key of the primary key in the design_53 table. To inform an apply process about the relationship between these tables, run the following procedures to create a value dependency at the destination database:

```
BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'key_53_foreign_key',
    object_name       => 'us_designs.design_53',
    attribute_list    => 'key_53');
END;
/

BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'key_53_foreign_key',
    object_name       => 'us_designs.all_designs_summary',
    attribute_list    => 'key_53');
END;
/
```

Notice that the value dependencies use the schema at the source database (us_designs) because LCRs contain the source database schema. The schema will be changed to uk_designs by the DML handler after the apply process passes the row LCRs to the handler.

To unset a value dependency, run the SET_VALUE_DEPENDENCY procedure, and specify the name of the value dependency in the dependency_name parameter and NULL in the object_name parameter. For example, to unset the key_53_foreign_key value dependency that was set previously, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'key_53_foreign_key',
    object_name       => NULL,
    attribute_list    => NULL);
END;
/
```

Note: "Managing a DML Handler" on page 9-12

Undefined Constraints at the Destination Database and Value Dependencies This scenarios involves an environment in which foreign key constraints are used for shared tables at the source database, but no constraints are used for these tables at the destination database. In the replication environment, the destination database is used as a data warehouse where data is written to the database far more often than it is queried. To optimize write operations, no constraints are defined at the destination database.

In such an environment, an apply processes running on the destination database must be informed about the constraints to apply transactions consistently. Value dependencies can be used to inform the apply process about these constraints.

For example, assume that the orders and order_items tables in the oe schema are shared between the source database and the destination database in this environment. On the source database, the order_id column is a primary key in the orders table, and the order_id column in the order_items table is a foreign key that matches the primary key column in the orders table. At the destination database, these

constraints have been removed. Run the following procedures to create a value dependency at the destination database that informs apply processes about the relationship between the columns in these tables:

```
BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'order_id_foreign_key',
    object_name       => 'oe.orders',
    attribute_list    => 'order_id');
END;
/

BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'order_id_foreign_key',
    object_name       => 'oe.order_items',
    attribute_list    => 'order_id');
END;
/
```

Also, in this environment, the following actions should be performed so that apply processes can apply transactions consistently:

- Value dependencies should be set for each column that has a unique key or bitmap index at the source database.
- The DBMS_APPLY_ADM.SET_KEY_COLUMNS procedure should set substitute key columns for the columns that are primary key columns at the source database.

To unset the value dependency that was set previously, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_VALUE_DEPENDENCY(
    dependency_name   => 'order_id_foreign_key',
    object_name       => NULL,
    attribute_list    => NULL);
END;
/
```

Note: "Managing the Substitute Key Columns for a Table" on page 9-11

Creating and Dropping Object Dependencies

Use the CREATE_OBJECT_DEPENDENCY and DROP_OBJECT_DEPENDENCY procedures in the DBMS_APPLY_ADM package to create or drop an object dependency. The following sections provide detailed instructions for creating and dropping object dependencies.

Creating an Object Dependency An object dependency can be used when row LCRs for a particular table always should be applied before the row LCRs for another table, and the destination database's data dictionary does not contain a constraint to enforce this relationship. When you define an object dependency, the table whose row LCRs should be applied first is the parent table and the table whose row LCRs should be applied second is the child table.

For example, consider a Streams replication environment with the following characteristics:

- The following tables in the `ord` schema are shared between a source and destination database:
  - The `customers` table contains information about customers, including each customer's shipping address.
  - The `orders` table contains information about each order.
  - The `order_items` table contains information about the items ordered in each order.
  - The `ship_orders` table contains information about orders that are ready to ship, but it does not contain detailed information about the customer or information about individual items to ship with each order.
- The `ship_orders` table has no relationships, defined by constraints, with the other tables.
- Information about orders is entered into the source database and propagated to the destination database, where it is applied.
- The destination database site is a warehouse where orders are shipped to customers. At this site, a DML handler uses the information in the `ship_orders`, `customers`, `orders`, and `order_items` tables to generate a report that includes the customer's shipping address and the items to ship.

The information in the report generated by the DML handler must be consistent with the time when the ship order record was created. An object dependency at the destination database can accomplish this goal. In this case, the `ship_orders` table is the parent table of the following child tables: `customers`, `orders`, and `order_items`. Because `ship_orders` is the parent of these tables, any changes to these tables made after a record in the `ship_orders` table was entered will not be applied until the DML handler has generated the report for the ship order.

To create these object dependencies, run the following procedures at the destination database:

```
BEGIN
  DBMS_APPLY_ADM.CREATE_OBJECT_DEPENDENCY(
    object_name         =>  'ord.customers',
    parent_object_name  =>  'ord.ship_orders');
END;
/

BEGIN
  DBMS_APPLY_ADM.CREATE_OBJECT_DEPENDENCY(
    object_name         =>  'ord.orders',
    parent_object_name  =>  'ord.ship_orders');
END;
/

BEGIN
  DBMS_APPLY_ADM.CREATE_OBJECT_DEPENDENCY(
    object_name         =>  'ord.order_items',
    parent_object_name  =>  'ord.ship_orders');
END;
/
```

See Also: "Managing a DML Handler" on page 9-12

Dropping an Object Dependency To drop the object dependencies created in "Creating an Object Dependency" on page 9-20, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.DROP_OBJECT_DEPENDENCY(
    object_name         =>  'ord.customers',
    parent_object_name  =>  'ord.ship_orders');
END;
/

BEGIN
  DBMS_APPLY_ADM.DROP_OBJECT_DEPENDENCY(
    object_name         =>  'ord.orders',
    parent_object_name  =>  'ord.ship_orders');
END;
/

BEGIN
  DBMS_APPLY_ADM.DROP_OBJECT_DEPENDENCY(
    object_name         =>  'ord.order_items',
    parent_object_name  =>  'ord.ship_orders');
END;
/
```

Managing Streams Conflict Detection and Resolution

This section describes the following tasks:

- Setting an Update Conflict Handler
- Modifying an Existing Update Conflict Handler
- Removing an Existing Update Conflict Handler
- Stopping Conflict Detection for Nonkey Columns

See Also:

- Chapter 3, "Streams Conflict Resolution"
- "Displaying Information About Update Conflict Handlers" on page 12-12

Setting an Update Conflict Handler

Set an update conflict handler using the `SET_UPDATE_CONFLICT_HANDLER` procedure in the `DBMS_APPLY_ADM` package. You can use one of the following prebuilt methods when you create an update conflict resolution handler:

- `OVERWRITE`
- `DISCARD`
- `MAXIMUM`
- `MINIMUM`

For example, suppose a Streams environment captures changes to the `hr.jobs` table at `dbs1.net` and propagates these changes to the `dbs2.net` destination database, where they are applied. In this environment, applications can perform DML changes

on the hr.jobs table at both databases, but, if there is a conflict for a particular DML change, then the change at the dbs1.net database should always overwrite the change at the dbs2.net database. In this environment, you can accomplish this goal by specifying an OVERWRITE handler at the dbs2.net database.

To specify an update conflict handler for the hr.jobs table in the hr schema at the dbs2.net database, run the following procedure at dbs2.net:

```
DECLARE
  cols DBMS_UTILITY.NAME_ARRAY;
  BEGIN
    cols(1) := 'job_title';
    cols(2) := 'min_salary';
    cols(3) := 'max_salary';
    DBMS_APPLY_ADM.SET_UPDATE_CONFLICT_HANDLER(
      object_name       => 'hr.jobs',
      method_name       => 'OVERWRITE',
      resolution_column => 'job_title',
      column_list       => cols);
END;
/
```

All apply processes running on a database that apply changes to the specified table locally use the specified update conflict handler.

Note:

- The resolution_column is not used for OVERWRITE and DISCARD methods, but one of the columns in the column_list still must be specified.
- You must specify a conditional supplemental log group at the source database for all of the columns in the column_list at the destination database. In this example, you would specify a conditional supplemental log group including the job_title, min_salary, and max_salary columns in the hr.jobs table at the dbs1.net database.
- Prebuilt update conflict handlers do not support LOB, LONG, LONG RAW, and user-defined type columns. Therefore, you should not include these types of columns in the column_list parameter when running the procedure SET_UPDATE_CONFLICT_HANDLER.

See Also:

- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3
- Chapter 16, "Multiple-Source Replication Example" for an example Streams environment that illustrates using the MAXIMUM prebuilt method for time-based conflict resolution

Modifying an Existing Update Conflict Handler

You can modify an existing update conflict handler by running the SET_UPDATE_CONFLICT_HANDLER procedure in the DBMS_APPLY_ADM package. To update an existing conflict handler, specify the same table and resolution column as the existing conflict handler.

To modify the update conflict handler created in "Setting an Update Conflict Handler" on page 9-22, you specify the hr.jobs table and the job_title column as the resolution column. You can modify this update conflict handler by specifying a different type of prebuilt method or a different column list, or both. However, if you want to change the resolution column for an update conflict handler, then you must remove and re-create the handler.

For example, suppose the environment changes, and you want changes from dbs1.net to be discarded in the event of a conflict, whereas previously changes from dbs1.net overwrote changes at dbs2.net. You can accomplish this goal by specifying a DISCARD handler at the dbs2.net database.

To modify the existing update conflict handler for the hr.jobs table in the hr schema at the dbs2.net database, run the following procedure:

```
DECLARE
  cols DBMS_UTILITY.NAME_ARRAY;
  BEGIN
    cols(1) := 'job_title';
    cols(2) := 'min_salary';
    cols(3) := 'max_salary';
    DBMS_APPLY_ADM.SET_UPDATE_CONFLICT_HANDLER(
      object_name       => 'hr.jobs',
      method_name       => 'DISCARD',
      resolution_column => 'job_title',
      column_list       => cols);
END;
/
```

Removing an Existing Update Conflict Handler

You can remove an existing update conflict handler by running the SET_UPDATE_CONFLICT_HANDLER procedure in the DBMS_APPLY_ADM package. To remove a an existing conflict handler, specify NULL for the method, and specify the same table, column list, and resolution column as the existing conflict handler.

For example, suppose you want to remove the update conflict handler created in "Setting an Update Conflict Handler" on page 9-22 and then modified in "Modifying an Existing Update Conflict Handler" on page 9-23. To remove this update conflict handler, run the following procedure:

```
DECLARE
  cols DBMS_UTILITY.NAME_ARRAY;
  BEGIN
    cols(1) := 'job_title';
    cols(2) := 'min_salary';
    cols(3) := 'max_salary';
    DBMS_APPLY_ADM.SET_UPDATE_CONFLICT_HANDLER(
      object_name       => 'hr.jobs',
      method_name       => NULL,
      resolution_column => 'job_title',
      column_list       => cols);
END;
/
```

Stopping Conflict Detection for Nonkey Columns

You can stop conflict detection for nonkey columns using the COMPARE_OLD_VALUES procedure in the DBMS_APPLY_ADM package.

For example, suppose you configure a time column for conflict resolution for the hr.employees table, as described in "MAXIMUM" on page 3-8. In this case, you can decide to stop conflict detection for the other nonkey columns in the table. After adding the time column and creating the trigger as described in that section, add the columns in the hr.employees table to the column list for an update conflict handler:

```
DECLARE
  cols  DBMS_UTILITY.NAME_ARRAY;
BEGIN
  cols(1)  := 'first_name';
  cols(2)  := 'last_name';
  cols(3)  := 'email';
  cols(4)  := 'phone_number';
  cols(5)  := 'hire_date';
  cols(6)  := 'job_id';
  cols(7)  := 'salary';
  cols(8)  := 'commission_pct';
  cols(9)  := 'manager_id';
  cols(10) := 'department_id';
  cols(11) := 'time';
  DBMS_APPLY_ADM.SET_UPDATE_CONFLICT_HANDLER(
    object_name       => 'hr.employees',
    method_name       => 'MAXIMUM',
    resolution_column => 'time',
    column_list       => cols);
END;
/
```

This example does not include the primary key for the table in the column list because it assumes that the primary key is never updated. However, other key columns are included in the column list.

To stop conflict detection for all nonkey columns in the table for both UPDATE and DELETE operations at a destination database, run the following procedure:

```
DECLARE
  cols DBMS_UTILITY.LNAME_ARRAY;
  BEGIN
    cols(1) := 'first_name';
    cols(2) := 'last_name';
    cols(3) := 'email';
    cols(4) := 'phone_number';
    cols(5) := 'hire_date';
    cols(6) := 'job_id';
    cols(7) := 'salary';
    cols(8) := 'commission_pct';
  DBMS_APPLY_ADM.COMPARE_OLD_VALUES(
    object_name  => 'hr.employees',
    column_table => cols,
    operation    => '*',
    compare      => false);
END;
/
```

The asterisk (*) specified for the operation parameter means that conflict detection is stopped for both UPDATE and DELETE operations. After you run this procedure, all

apply processes running on the database that apply changes to the specified table locally do not detect conflicts on the specified columns. Therefore, in this example, the `time` column is the only column used for conflict detection.

Note: The example in this section sets an update conflict handler before stopping conflict detection for nonkey columns. However, an update conflict handler is not required before you stop conflict detection for nonkey columns.

See Also:

- "Control Over Conflict Detection for Nonkey Columns" on page 3-4
- "Displaying Information About Conflict Detection" on page 12-11
- Chapter 16, "Multiple-Source Replication Example" for a detailed example that uses time-based conflict resolution
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `COMPARE_OLD_VALUES` procedure

Managing Streams Tags

You can set or get the value of the tags generated by the current session or by an apply process. The following sections describe how to set and get tag values.

- Managing Streams Tags for the Current Session
- Managing Streams Tags for an Apply Process

See Also:

- Chapter 4, "Streams Tags"
- "Monitoring Streams Tags" on page 12-13

Managing Streams Tags for the Current Session

This section contains instructions for setting and getting the tag for the current session.

Setting the Tag Values Generated by the Current Session

You can set the tag for all redo entries generated by the current session using the `SET_TAG` procedure in the `DBMS_STREAMS` package. For example, to set the tag to the hexadecimal value of `'1D'` in the current session, run the following procedure:

```
BEGIN
   DBMS_STREAMS.SET_TAG(
      tag  =>  HEXTORAW('1D'));
END;
/
```

After running this procedure, each redo entry generated by DML or DDL statements in the current session will have a tag value of `1D`. Running this procedure affects only the current session.

Getting the Tag Value for the Current Session

You can get the tag for all redo entries generated by the current session using the GET_TAG procedure in the DBMS_STREAMS package. For example, to get the hexadecimal value of the tags generated in the redo entries for the current session, run the following procedure:

```
SET SERVEROUTPUT ON
DECLARE
   raw_tag RAW(2048);
BEGIN
   raw_tag := DBMS_STREAMS.GET_TAG();
   DBMS_OUTPUT.PUT_LINE('Tag Value = ' || RAWTOHEX(raw_tag));
END;
/
```

You can also display the tag value for the current session by querying the DUAL view:

```
SELECT DBMS_STREAMS.GET_TAG FROM DUAL;
```

Managing Streams Tags for an Apply Process

This section contains instructions for setting and removing the tag for an apply process.

See Also:

- "Tags and an Apply Process" on page 4-5 for conceptual information about how tags are used by an apply process and apply handlers
- "Apply and Streams Replication" on page 1-12
- "Managing Apply for Streams Replication" on page 9-9

Setting the Tag Values Generated by an Apply Process

An apply process generates redo entries when it applies changes to a database or invokes handlers. You can set the default tag for all redo entries generated by an apply process when you create the apply process using the CREATE_APPLY procedure in the DBMS_APPLY_ADM package, or when you alter an existing apply process using the ALTER_APPLY procedure in the DBMS_APPLY_ADM package. In both of these procedures, set the apply_tag parameter to the value you want to specify for the tags generated by the apply process.

For example, to set the value of the tags generated in the redo log by an existing apply process named strep01_apply to the hexadecimal value of '7', run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.ALTER_APPLY(
     apply_name  =>  'strep01_apply',
     apply_tag   =>  HEXTORAW('7'));
END;
/
```

After running this procedure, each redo entry generated by the apply process will have a tag value of 7.

Removing the Apply Tag for an Apply Process

You remove the apply tag for an apply process by setting the `remove_apply_tag` parameter to `true` in the `ALTER_APPLY` procedure in the `DBMS_APPLY_ADM` package. Removing the apply tag means that each redo entry generated by the apply process has a `NULL` tag. For example, the following procedure removes the apply tag from an apply process named `strep01_apply`.

```
BEGIN
  DBMS_APPLY_ADM.ALTER_APPLY(
    apply_name       => 'strep01_apply',
    remove_apply_tag => true);
END;
/
```

Changing the DBID or Global Name of a Source Database

Typically, database administrators change the `DBID` and global name of a database when it is a clone of another database. You can view the `DBID` of a database by querying the `DBID` column in the `V$DATABASE` dynamic performance view, and you can view the global name of a database by querying the `GLOBAL_NAME` static data dictionary view. When you change the `DBID` or global name of a source database, any existing capture processes that capture changes originating at this source database become unusable. The capture processes can be local capture processes or downstream capture processes that capture changes that originated at the source database. Also, any existing apply processes that apply changes from the source database become unusable.

If a capture process is capturing changes generated by a database for which you have changed the `DBID` or global name, then complete the following steps:

1. Shut down the source database.
2. Restart the source database with `RESTRICTED SESSION` enabled using `STARTUP RESTRICT`.
3. Drop the capture process using the `DROP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package. The capture process can be a local capture process at the source database or a downstream capture process at a remote database.
4. At the source database, run the `ALTER SYSTEM SWITCH LOGFILE` statement on the database.
5. If any changes have been captured from the source database, then manually resynchronize the data at all destination databases that apply changes originating at this source database. If the database never captured any changes, then this step is not necessary.
6. Modify any rules that use the source database name as a condition. The source database name should be changed to the new global name of the source database where appropriate in these rules. You might need to modify capture process rules, propagation rules, and apply process rules at the local database and at remote databases in the environment.
7. Drop the apply processes that apply changes from the capture process that you dropped in Step 3. Use the `DROP_APPLY` procedure in the `DBMS_APPLY_ADM` package to drop an apply process.

8. At each destination database that applies changes from the source database, re-create the apply processes you dropped in Step 7. You might want to associate the each apply process with the same rule sets it used before it was dropped. See "Creating an Apply Process that Applies LCRs" on page 9-10 for instructions.
9. Re-create the capture process you dropped in Step 3, if necessary. You might want to associate the capture process with the same rule sets used by the capture process you dropped in Step 3. See "Creating a Capture Process" on page 9-1 for instructions.
10. At the source database, prepare database objects whose changes will be captured by the re-created capture process for instantiation. See "Preparing Database Objects for Instantiation at a Source Database" on page 10-1.
11. At each destination database that applies changes from the source database, set the instantiation SCN for all databases objects to which changes from the source database will be applied. See "Setting Instantiation SCNs at a Destination Database" on page 10-27 for instructions.
12. Disable the restricted session using the `ALTER SYSTEM DISABLE RESTRICTED SESSION` statement.
13. At each destination database that applies changes from the source database, start the apply processes you created in Step 8.
14. At the source database, start the capture process you created in Step 9.

See Also: *Oracle Database Utilities* for more information about changing the `DBID` of a database using the DBNEWID utility

Resynchronizing a Source Database in a Multiple-Source Environment

A multiple-source environment is one in which there is more than one source database for any of the shared data. If a source database in a multiple-source environment cannot be recovered to the current point in time, then you can use the method described in this section to resynchronize the source database with the other source databases in the environment. Some reasons why a database cannot be recovered to the current point in time include corrupted archived redo logs or the media failure of an online redo log group.

For example, a bidirectional Streams environment is one in which exactly two databases share the replicated database objects and data. In this example, assume that database A is the database that must be resynchronized and that database B is the other source database in the environment. To resynchronize database A in this bidirectional Streams environment, complete the following steps:

1. Verify that database B has applied all of the changes sent from database A. You can query the `V$BUFFERED_SUBSCRIBERS` data dictionary view at database B to determine whether the apply process that applies these changes has any unapplied changes in its queue. See the example on viewing propagations dequeuing LCRs from each buffered queue in *Oracle Streams Concepts and Administration* for an example of such a query. Do not continue until all of these changes have been applied.
2. Remove the Streams configuration from database A by running the `REMOVE_STREAMS_CONFIGURATION` procedure in the `DBMS_STREAMS_ADM` package. See *Oracle Database PL/SQL Packages and Types Reference* for more information about this procedure.

3. At database B, drop the apply process that applies changes from database A. Do not drop the rule sets used by this apply process because you will re-create the apply process in a subsequent step.
4. Complete the steps in "Adding a New Database to an Existing Multiple-Source Environment" on page 8-16 to add database A back into the Streams environment.

Performing Database Point-in-Time Recovery in a Streams Environment

Point-in-time recovery is the recovery of a database to a specified noncurrent time, SCN, or log sequence number. The following sections discuss performing point-in-time recovery in a Streams replication environment:

- Performing Point-in-Time Recovery on the Source in a Single-Source Environment
- Performing Point-in-Time Recovery in a Multiple-Source Environment
- Performing Point-in-Time Recovery on a Destination Database

> **See Also:** *Oracle Database Backup and Recovery Advanced User's Guide* for more information about point-in-time recovery

Performing Point-in-Time Recovery on the Source in a Single-Source Environment

A single-source Streams replication environment is one in which there is only one source database for shared data. If database point-in-time recovery is required at the source database in a single-source Streams environment, and any capture processes that capture changes generated at a source database are running, then you must stop these capture processes before you perform the recovery operation. Both local and downstream capture process that capture changes generated at the source database must be stopped. Typically, database administrators reset the log sequence number of a database during point-in-time recovery. The `ALTER DATABASE OPEN RESETLOGS` statement is an example of a statement that resets the log sequence number.

The instructions in this section assume that the single-source replication environment has the following characteristics:

- Only one capture process named `strm01_capture`, which can be a local or downstream capture process
- Only one destination database with the global name `dest.net`
- Only one apply process named `strm01_apply` at the destination database

If point-in-time recovery must be performed on the source database, then you can follow these instructions to recover as many transactions as possible at the source database by using transactions applied at the destination database. These instructions assume that you can identify the transactions applied at the destination database after the source point-in-time SCN and execute these transactions at the source database.

Note: Oracle recommends that you set the apply process parameter `COMMIT_SERIALIZATION` to `FULL` when performing point-in-time recovery in a single-source Streams replication environment.

Complete the following steps to perform point-in-time recovery on the source database in a single-source Streams replication environment:

1. Perform point-in-time recovery on the source database if you have not already done so. Note the point-in-time recovery SCN because it is needed in subsequent steps.
2. Ensure that the source database is in restricted mode.
3. Stop the capture process using the `STOP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.
4. At the source database, perform a data dictionary build:

```
SET SERVEROUTPUT ON
DECLARE
  scn  NUMBER;
BEGIN
  DBMS_CAPTURE_ADM.BUILD(
    first_scn => scn);
  DBMS_OUTPUT.PUT_LINE('First SCN Value = ' || scn);
END;
/
```

   Note the SCN value returned because it is needed in Step 13.

5. At the destination database, wait until all of the transactions from the source database in the apply process queue have been applied. The apply processes should become idle when these transactions have been applied. You can query the `STATE` column in both the `V$STREAMS_APPLY_READER` and `V$STREAMS_APPLY_SERVER`. The state should be `IDLE` for the apply process in both views before you continue.
6. Perform a query at the destination database to determine the highest SCN for a transaction that was applied.

   If the apply process is running, then perform the following query:

```
SELECT HWM_MESSAGE_NUMBER FROM V$STREAMS_APPLY_COORDINATOR
  WHERE APPLY_NAME = 'STRM01_APPLY';
```

   If the apply process is disabled, then perform the following query:

```
SELECT APPLIED_MESSAGE_NUMBER FROM DBA_APPLY_PROGRESS
  WHERE APPLY_NAME = 'STRM01_APPLY';
```

   Note the highest apply SCN returned by the query because it is needed in subsequent steps.

7. If the highest apply SCN obtained in Step 6 is less than the point-in-time recovery SCN noted in Step 1, then proceed to step 8. Otherwise, if the highest apply SCN obtained in Step 6 is greater than or equal to the point-in-time recovery SCN noted in Step 1, then the apply process has applied some transactions from the source database after point-in-time recovery SCN. In this case complete the following steps:

   a. Manually execute transactions applied after the point-in-time SCN at the source database. When you execute these transactions at the source database, make sure you set a Streams tag in the session so that the transactions will not be captured by the capture process. If no such Streams session tag is set, then these changes can be cycled back to the destination database. See "Managing Streams Tags for the Current Session" on page 9-26 for instructions.

b. Disable the restricted session at the source database.

8. If you completed the actions in Step 7, then proceed to Step 12. Otherwise, if the highest apply SCN obtained in Step 6 is less than the point-in-time recovery SCN noted in Step 1, then the apply process has not applied any transactions from the source database after point-in-time recovery SCN. In this case, complete the following steps:

a. Disable the restricted session at the source database.

b. Ensure that the apply process is running at the destination database.

c. Set the `maximum_scn` capture process parameter of the original capture process to the point-in-time recovery SCN using the `SET_PARAMETER` procedure in the `DBMS_CAPTURE_ADM` package.

d. Set the start SCN of the original capture process to the oldest SCN of the apply process. You can determine the oldest SCN of a running apply process by querying the `OLDEST_SCN_NUM` column in the `V$STREAMS_APPLY_READER` dynamic performance view at the destination database. To set the start SCN of the capture process, specify the `start_scn` parameter when you run the `ALTER_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

e. Ensure that the capture process writes information to the alert log by running the following procedure:

```
BEGIN
  DBMS_CAPTURE_ADM.SET_PARAMETER(
    capture_name => 'strm01_capture',
    parameter    => 'write_alert_log',
    value        => 'Y');
END;
/
```

f. Start the original capture process using the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

g. Ensure that the original capture process has captured all changes up to the `maximum_scn` setting by querying the `CAPTURED_SCN` column in the `DBA_CAPTURE` data dictionary view. When the value returned by the query is equal to or greater than the `maximum_scn` value, the capture process should stop automatically. When the capture process is stopped, proceed to the next step.

h. Find the value of the `LAST_ENQUEUE_MESSAGE_NUMBER` in the alert log. Note this value because it is needed in subsequent steps.

i. At the destination database, wait until all the changes are applied. You can monitor the applied changes for the apply process `strm01_apply` by running the following queries at the destination database:

```
SELECT DEQUEUED_MESSAGE_NUMBER
  FROM V$STREAMS_APPLY_READER
  WHERE APPLY_NAME = 'STRM01_APPLY' AND
        DEQUEUED_MESSAGE_NUMBER = last_enqueue_message_number;
```

Substitute the `LAST_ENQUEUE_MESSAGE_NUMBER` found in the alert log in Step h for *last_enqueue_message_number* on the last line of the query. When this query returns a row, all of the changes from the capture database have been applied at the destination database.

Also, ensure that the state of the apply process reader server and each apply server is IDLE. For example, run the following queries for an apply process named strm01_apply:

```
SELECT STATE FROM V$STREAMS_APPLY_READER
  WHERE APPLY_NAME = 'STRM01_APPLY';

SELECT STATE FROM V$STREAMS_APPLY_SERVER
  WHERE APPLY_NAME = 'STRM01_APPLY';
```

When both of these queries return IDLE, move on to the next step.

9. At the destination database, drop the apply process using the DROP_APPLY procedure in the DBMS_APPLY_ADM package.
10. At the destination database, create a new apply process. The new apply process should use the same queue and rule sets used by the original apply process.
11. At the destination database, start the new apply process using the START_APPLY procedure in the DBMS_APPLY_ADM package.
12. Drop the original capture process using the DROP_CAPTURE procedure in the DBMS_CAPTURE_ADM package.
13. Create a new capture process using the CREATE_CAPTURE procedure in the DBMS_CAPTURE_ADM package to replace the capture process you dropped in Step 12. Specify the SCN returned by the data dictionary build in Step 4 for both the first_scn and start_scn parameters. The new capture process should use the same queue and rule sets as the original capture process.
14. Start the new capture process using the START_CAPTURE procedure in the DBMS_CAPTURE_ADM package.

Performing Point-in-Time Recovery in a Multiple-Source Environment

A multiple-source environment is one in which there is more than one source database for any of the shared data. If database point-in-time recovery is required at a source database in a multiple-source Streams environment, then you can use another source database in the environment to recapture the changes made to the recovered source database after the point-in-time recovery.

For example, in a multiple-source Streams environment, one source database can become unavailable at time T2 and undergo point in time recovery to an earlier time T1. After recovery to T1, transactions performed at the recovered database between T1 and T2 are lost at the recovered database. However, before the recovered database became unavailable, assume that these transactions were propagated to another source database and applied. In this case, this other source database can be used to restore the lost changes to the recovered database.

Specifically, to restore changes made to the recovered database after the point-in-time recovery, you configure a capture process to recapture these changes from the redo logs at the other source database, a propagation to propagate these changes from the database where changes are recaptured to the recovered database, and an apply process at the recovered database to apply these changes.

Changes originating at the other source database that were applied at the recovered database between T1 and T2 also have been lost and must be recovered. To accomplish this, alter the capture process at the other source database to start capturing changes at an earlier SCN. This SCN is the oldest SCN for the apply process at the recovered database.

The following SCN values are required to restore lost changes to the recovered database:

- **Point-in-time SCN**: The SCN for the point-in-time recovery at the recovered database.
- **Instantiation SCN**: The SCN value to which the instantiation SCN must be set for each database object involved in the recovery at the recovered database while changes are being reapplied. At the other source database, this SCN value corresponds to one less than the *commit SCN* of the first transaction that was applied at the other source database and lost at the recovered database.
- **Start SCN**: The SCN value to which the start SCN is set for the capture process created to recapture changes at the other source database. This SCN value corresponds to the *earliest SCN* at which the apply process at the other source database started applying a transaction that was lost at the recovered database. This capture process can be a local or downstream capture process that uses the other source database for its source database.
- **Maximum SCN**: The SCN value to which the `maximum_scn` parameter for the capture process created to recapture lost changes should be set. The capture process stops capturing changes when it reaches this SCN value. The current SCN for the other source database is used for this value.

You should record the point-in-time SCN when you perform point-in-time recovery on the recovered database. You can use the `GET_SCN_MAPPING` procedure in the `DBMS_STREAMS_ADM` package to determine the other necessary SCN values.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the `GET_SCN_MAPPING` procedure

Performing Point-in-Time Recovery on a Destination Database

If database point-in-time recovery is required at a destination database in a Streams environment, then you must reapply the captured changes that had already been applied after the point-in-time recovery.

For each relevant capture process, you can choose either of the following methods to perform point-in-time recovery at a destination database in a Streams environment:

- Reset the start SCN for the existing capture process that captures the changes that are applied at the destination database.
- Create a new capture process to capture the changes that must be reapplied at the destination database.

Resetting the start SCN for the capture process is simpler than creating a new capture process. However, if the capture process captures changes that are applied at multiple destination databases, then the changes are resent to all the destination databases, including the ones that did not perform point-in-time recovery. If a change is already applied at a destination database, then it is discarded by the apply process, but you might not want to use the network and computer resources required to resend the changes to multiple destination databases. In this case, you can create and temporarily use a new capture process and a new propagation that propagates changes only to the destination database that was recovered.

The following sections provide instructions for each task:

- Resetting the Start SCN for the Existing Capture Process to Perform Recovery
- Creating a New Capture Process to Perform Recovery

If there are multiple apply processes at the destination database where you performed point-in-time recovery, then complete one of the tasks in this section for each apply process.

Neither of these methods should be used if any of the following conditions are true regarding the destination database you are recovering:

- A propagation propagates user-enqueued messages to the destination database. Both of these methods reapply only captured LCRs at the destination database, not user-enqueued LCRs.
- In a directed networks configuration, the destination database is used to propagate LCRs from a capture process to other databases, but the destination database does not apply LCRs from this capture process.
- The oldest message number for an apply process at the destination database is lower than the first SCN of a capture process that captures changes for this apply process. The following query at a destination database lists the oldest message number (oldest SCN) for each apply process:

  ```
  SELECT APPLY_NAME, OLDEST_MESSAGE_NUMBER FROM DBA_APPLY_PROGRESS;
  ```

  The following query at a source database lists the first SCN for each capture process:

  ```
  SELECT CAPTURE_NAME, FIRST_SCN FROM DBA_CAPTURE;
  ```

- The archived log files that contain the intended start SCN are no longer available.

If any of these conditions are true in your environment, then you cannot use the methods described in this section. Instead, you must manually resynchronize the data at all destination databases.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about SCN values relating to a capture process and directed networks

Resetting the Start SCN for the Existing Capture Process to Perform Recovery

If you decide to reset the start SCN for the existing capture process to perform point-in-time recovery, then complete the following steps:

1. If the destination database is also a source database in a multiple-source Streams environment, then complete the actions described in "Performing Point-in-Time Recovery in a Multiple-Source Environment" on page 9-33.
2. If you are not using directed networks between the source database and destination database, then drop the propagation that propagates changes from the source queue at the source database to the destination queue at the destination database. Use the `DROP_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to drop the propagation.

   If you are using directed networks, and there are intermediate databases between the source database and destination database, then drop the propagation at each intermediate database in the path to the destination database, including the propagation at the source database.

   Do not drop the rule sets used by the propagations you drop.

Note: You must drop the appropriate propagation(s). Disabling them is not sufficient. You will re-create the propagation(s) in Step 7, and dropping them now ensures that only LCRs created after resetting the start SCN for the capture process are propagated.

See Also: *Oracle Streams Concepts and Administration* for more information about directed networks

3. Perform the point-in-time recovery at the destination database.
4. Query for the oldest message number (oldest SCN) from the source database for the apply process at the destination database. Make a note of the results of the query. The oldest message number is the earliest system change number (SCN) that might need to be applied.

   The following query at a destination database lists the oldest message number for each apply process:

   ```
   SELECT APPLY_NAME, OLDEST_MESSAGE_NUMBER FROM DBA_APPLY_PROGRESS;
   ```

5. Stop the existing capture process using the `STOP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.
6. Reset the start SCN of the existing capture process.

   To reset the start SCN for an existing capture process, run the `ALTER_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package and set the `start_scn` parameter to the value you recorded from the query in Step 4. For example, to reset the start SCN for a capture process named `strm01_capture` to the value `829381993`, run the following `ALTER_CAPTURE` procedure:

   ```
   BEGIN
     DBMS_CAPTURE_ADM.ALTER_CAPTURE(
       capture_name  =>  'strm01_capture',
       start_scn     =>  829381993);
   END;
   /
   ```

7. If you are not using directed networks between the source database and destination database, then create a new propagation to propagate changes from the source queue to the destination queue using the `CREATE_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package. Specify any rule sets used by the original propagation when you create the propagation.

   If you are using directed networks, and there are intermediate databases between the source database and destination database, then create a new propagation at each intermediate database in the path to the destination database, including the propagation at the source database.
8. Start the existing capture process using the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

Creating a New Capture Process to Perform Recovery

If you decide to create a new capture process to perform point-in-time recovery, then complete the following steps:

1. If the destination database is also a source database in a multiple-source Streams environment, then complete the actions described in "Performing Point-in-Time Recovery in a Multiple-Source Environment" on page 9-33.

2. If you are not using directed networks between the source database and destination database, then drop the propagation that propagates changes from the source queue at the source database to the destination queue at the destination database. Use the `DROP_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to drop the propagation.

   If you are using directed networks, and there are intermediate databases between the source database and destination database, then drop the propagation that propagates LCRs between the last intermediate database and the destination database. You do not need to drop the propagations at the other intermediate databases nor at the source database.

   Note: You must drop the appropriate propagation. Disabling it is not sufficient.

   See Also: *Oracle Streams Concepts and Administration* for more information about directed networks

3. Perform the point-in-time recovery at the destination database.

4. Query for the oldest message number (oldest SCN) from the source database for the apply process at the destination database. Make a note of the results of the query. The oldest message number is the earliest system change number (SCN) that might need to be applied.

   The following query at a destination database lists the oldest message number for each apply process:

   ```
   SELECT APPLY_NAME, OLDEST_MESSAGE_NUMBER FROM DBA_APPLY_PROGRESS;
   ```

5. Create a queue at the source database to be used by the capture process using the `SET_UP_QUEUE` procedure in the `DBMS_STREAMS_ADM` package.

   If you are using directed networks, and there are intermediate databases between the source database and destination database, then create a queue at each intermediate database in the path to the destination database, including the new queue at the source database. Do not create a new queue at the destination database.

6. If you are not using directed networks between the source database and destination database, then create a new propagation to propagate changes from the source queue created in Step 5 to the destination queue using the `CREATE_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package. Specify any rule sets used by the original propagation when you create the propagation.

   If you are using directed networks, and there are intermediate databases between the source database and destination database, then create a propagation at each intermediate database in the path to the destination database, including the propagation from the source database to the first intermediate database. These propagations propagate changes captured by the capture process you will create in Step 7 between the queues created in Step 5.

7. Create a new capture process at the source database using the `CREATE_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package. Set the `source_queue`

parameter to the local queue you created in Step 5 and the `start_scn` parameter to the value you recorded from the query in Step 4. Also, specify any rule sets used by the original capture process. If the rule sets used by the original capture process instruct the capture process to capture changes that should not be sent to the destination database that was recovered, then you can create and use smaller, customized rule sets that share some rules with the original rule sets.

8. Start the capture process you created in Step 7 using the `START_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.
9. When the oldest message number of the apply process at the recovered database is approaching the capture number of the original capture process at the source database, stop the original capture process using the `STOP_CAPTURE` procedure in the `DBMS_CAPTURE_ADM` package.

   At the destination database, you can use the following query to determine the oldest message number from the source database for the apply process:

   ```
   SELECT APPLY_NAME, OLDEST_MESSAGE_NUMBER FROM DBA_APPLY_PROGRESS;
   ```

   At the source database, you can use the following query to determine the capture number of the original capture process:

   ```
   SELECT CAPTURE_NAME, CAPTURE_MESSAGE_NUMBER FROM V$STREAMS_CAPTURE;
   ```

10. When the oldest message number of the apply process at the recovered database is beyond the capture number of the original capture process at the source database, drop the new capture process created in Step 7.
11. If you are not using directed networks between the source database and destination database, then drop the new propagation created in Step 6.

    If you are using directed networks, and there are intermediate databases between the source database and destination database, then drop the new propagation at each intermediate database in the path to the destination database, including the new propagation at the source database.
12. If you are not using directed networks between the source database and destination database, then remove the queue created in Step 5.

    If you are using directed networks, and there are intermediate databases between the source database and destination database, then drop the new queue at each intermediate database in the path to the destination database, including the new queue at the source database. Do not drop the queue at the destination database.
13. If you are not using directed networks between the source database and destination database, then create a propagation that propagates changes from the original source queue at the source database to the destination queue at the destination database. Use the `CREATE_PROPAGATION` procedure in the `DBMS_PROPAGATION_ADM` package to create the propagation. Specify any rule sets used by the original propagation when you create the propagation.

    If you are using directed networks, and there are intermediate databases between the source database and destination database, then re-create the propagation from the last intermediate database to the destination database. You dropped this propagation in Step 2.
14. Start the capture process you stopped in Step 9.

All of the steps after Step 8 can be deferred to a later time, or they can be done as soon as the condition described in Step 9 is met.

10

Performing Instantiations

This chapter contains instructions for performing instantiations in a Streams replication environment. Database objects must be instantiated at a destination database before changes to these objects can be replicated.

This chapter contains these topics:

- Preparing Database Objects for Instantiation at a Source Database
- Aborting Preparation for Instantiation at a Source Database
- Instantiating Objects in a Streams Replication Environment
- Setting Instantiation SCNs at a Destination Database

See Also: Chapter 2, "Instantiation and Streams Replication"

Preparing Database Objects for Instantiation at a Source Database

If you use the DBMS_STREAMS_ADM package to create rules for a capture process, then any objects referenced in the system-created rules are prepared for instantiation automatically. If you use the DBMS_RULE_ADM package to create rules for a capture process, then you must prepare the database objects referenced in these rules for instantiation manually. In this case, you should prepare a database object for instantiation after a capture process has been configured to capture changes to the database object.

The following procedures in the DBMS_CAPTURE_ADM package prepare database objects for instantiation:

- PREPARE_TABLE_INSTANTIATION prepares a single table for instantiation.
- PREPARE_SCHEMA_INSTANTIATION prepares for instantiation all of the database objects in a schema and all database objects added to the schema in the future.
- PREPARE_GLOBAL_INSTANTIATION prepares for instantiation all of the objects in a database and all objects added to the database in the future.

If you run one of these procedures while a long running transaction is modifying one or more database objects being prepared for instantiation, then the procedure will wait until the long running transaction is complete before it records the ignore SCN for the objects, which is the SCN below which changes to an object cannot be applied at destination databases. Query the V$STREAMS_TRANSACTION dynamic performance view to monitor long running transactions being processed by a capture process or apply process.

In addition, these procedures can enable supplemental logging for any primary key, unique key, bitmap index, and foreign key columns, or for all columns, in the tables that are being prepared for instantiation. Use the `supplemental_logging` parameter in each of these procedures to specify the columns for which supplemental logging is enabled.

See Also:

- "Capture Process Rules and Preparation for Instantiation" on page 2-3
- "Instantiation SCN and Ignore SCN for an Apply Process" on page 1-27

The following sections contain examples that prepare tables for instantiation and specify different options for supplemental logging:

- Preparing a Table for Instantiation
- Preparing the Database Objects in a Schema for Instantiation
- Preparing All of the Database Objects in a Database for Instantiation

Preparing a Table for Instantiation

To prepare the `hr.regions` table for instantiation and enable supplemental logging for any primary key, unique key, bitmap index, and foreign key columns in the table, run the following procedure:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
    table_name          => 'hr.regions',
    supplemental_logging => 'keys');
END;
/
```

The default value for the `supplemental_logging` parameter is `keys`. Therefore, if this parameter is not specified, then supplemental logging is enabled for any primary key, unique key, bitmap index, and foreign key columns in the table that is being prepared for instantiation.

Preparing the Database Objects in a Schema for Instantiation

To prepare the database objects in the `hr` schema for instantiation and enable supplemental logging for the all columns in the tables in the `hr` schema, run the following procedure:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_SCHEMA_INSTANTIATION(
    schema_name          => 'hr',
    supplemental_logging => 'all');
END;
/
```

After running this procedure, supplemental logging is enabled for all of the columns in the tables in the `hr` schema and for all of the columns in the tables added to the `hr` schema in the future.

Preparing All of the Database Objects in a Database for Instantiation

To prepare all of the database objects in a database for instantiation, run the following procedure:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_GLOBAL_INSTANTIATION(
    supplemental_logging => 'none');
END;
/
```

Because `none` is specified for the `supplemental_logging` parameter, this procedure does not enable supplemental logging for any columns. However, you can specify supplemental logging manually using an `ALTER TABLE` or `ALTER DATABASE` statement.

See Also: "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3

Aborting Preparation for Instantiation at a Source Database

The following procedures in the `DBMS_CAPTURE_ADM` package abort preparation for instantiation:

- `ABORT_TABLE_INSTANTIATION` reverses the effects of `PREPARE_TABLE_INSTANTIATION` and removes any supplemental logging enabled by the `PREPARE_TABLE_INSTANTIATION` procedure.
- `ABORT_SCHEMA_INSTANTIATION` reverses the effects of `PREPARE_SCHEMA_INSTANTIATION` and removes any supplemental logging enabled by the `PREPARE_SCHEMA_INSTANTIATION` and `PREPARE_TABLE_INSTANTIATION` procedures.
- `ABORT_GLOBAL_INSTANTIATION` reverses the effects of `PREPARE_GLOBAL_INSTANTIATION` and removes any supplemental logging enabled by the `PREPARE_GLOBAL_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, and `PREPARE_TABLE_INSTANTIATION` procedures.

These procedures remove data dictionary information related to the potential instantiation of the relevant database objects.

For example, to abort the preparation for instantiation of the `hr.regions` table, run the following procedure:

```
BEGIN
  DBMS_CAPTURE_ADM.ABORT_TABLE_INSTANTIATION(
    table_name  => 'hr.regions');
END;
/
```

Instantiating Objects in a Streams Replication Environment

You can instantiate database objects in a Streams environment in the following ways:

- Instantiating Objects Using Data Pump Export/Import
- Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN
- Instantiating Objects Using Original Export/Import
- Instantiating an Entire Database Using RMAN

You can use Oracle Data Pump, transportable tablespaces, and the original Export/Import utilities to instantiate individual database objects, schemas, or an entire database. You can use RMAN to instantiate the database objects in a tablespace or tablespace set or to instantiate an entire database.

Note: You can use the following procedures in the `DBMS_STREAMS_ADM` package to configure Streams replication: `MAINTAIN_GLOBAL`, `MAINTAIN_SCHEMAS`, `MAINTAIN_SIMPLE_TTS`, `MAINTAIN_TABLES`, and `MAINTAIN_TTS`. If you use one of these procedures, then instantiation is performed automatically for the appropriate database objects. Oracle recommends using one of these procedures to configure replication.

See Also:

- "Overview of Instantiation and Streams Replication" on page 2-1
- "Instantiation SCN and Ignore SCN for an Apply Process" on page 1-27
- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1
- "Setting Instantiation SCNs at a Destination Database" on page 10-27
- Chapter 6, "Simple Streams Replication Configuration" for information about the procedures for configuring replication

Instantiating Objects Using Data Pump Export/Import

The example in this section describes the steps required to instantiate objects in a Streams environment using Oracle Data Pump export/import. This example makes the following assumptions:

- You want to capture changes to all of the database objects in the `hr` schema at a source database and apply these changes at a separate destination database.
- The `hr` schema exists at a source database but does not exist at a destination database. For the purposes of this example, you can drop the `hr` user at the destination database using the following SQL statement:

  ```
  DROP USER hr CASCADE;
  ```

  The Data Pump import re-creates the user and the user's database objects at the destination database.
- You have configured a Streams administrator at the source database and the destination database named `strmadmin`. At each database, the Streams administrator is granted `DBA` role.

Note: The example in this section uses the command line Data Pump utility. You can also use the `DBMS_DATAPUMP` package for Streams instantiations.

See Also:

- *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator
- *Oracle Database Utilities* for more information about Data Pump
- Part IV, "Sample Replication Environments" for examples that use the DBMS_DATAPUMP package for Streams instantiations

Given these assumptions, complete the following steps to instantiate the hr schema using Data Pump export/import:

1. While connected in SQL*Plus to the source database as the Streams administrator, create a directory object to hold the export dump file and export log file:

```
CREATE DIRECTORY DPUMP_DIR AS '/usr/dpump_dir';
```

2. While connected as the Streams administrator strmadmin at the source database, prepare the database objects in the hr schema for instantiation. You can complete this step in one of the following ways:

   - Add rules for the hr schema to the positive rule set for a capture process using a procedure in the DBMS_STREAMS_ADM package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then the procedure that you run prepares the objects in the hr schema for instantiation automatically.

     For example, the following procedure adds rules to the positive rule set of a capture process named strm01_capture and prepares the hr schema, and all of its objects, for instantiation:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name     =>  'hr',
    streams_type    =>  'capture',
    streams_name    =>  'strm01_capture',
    queue_name      =>  'strm01_queue',
    include_dml     =>  true,
    include_ddl     =>  true,
    inclusion_rule  =>  true);
END;
/
```

     If the specified capture process does not exist, then this procedure creates it.

   - Add rules for the hr schema to the positive rule set for a capture process using a procedure in the DBMS_RULE_ADM package, and then prepare the objects for instantiation manually by specifying the hr schema when you run the PREPARE_SCHEMA_INSTANTIATION procedure in the DBMS_CAPTURE_ADM package:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_SCHEMA_INSTANTIATION(
    schema_name  => 'hr');
END;
/
```

     Make sure you add the rules to the positive rule set for the capture process before you prepare the database objects for instantiation.

3. While still connected to the source database as the Streams administrator, determine the current system change number (SCN) of the source database:

   ```
   SELECT DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER FROM DUAL;
   ```

   The SCN value returned by this query is specified for the FLASHBACK_SCN Data Pump export parameter in Step 4. Because the hr schema includes foreign key constraints between tables, the FLASHBACK_SCN export parameter, or a similar export parameter, must be specified during export. In this example, assume that the query returned 876606.

   After you perform this query, make sure no DDL changes are made to the objects being exported until after the export is complete.

4. On a command line, use Data Pump to export the hr schema at the source database.

   Perform the export by connecting as an administrative user who is granted EXP_FULL_DATABASE role. This user also must have READ and WRITE privilege on the directory object created in Step 1. This example connects as the Streams administrator strmadmin.

   The following is an example Data Pump export command:

   ```
   expdp strmadmin/strmadminpw SCHEMAS=hr DIRECTORY=DPUMP_DIR DUMPFILE=hr_schema_
   dp.dmp FLASHBACK_SCN=876606
   ```

   See Also: *Oracle Database Utilities* for information about performing a Data Pump export

5. While connected in SQL*Plus to the destination database the Streams administrator, create a directory object to hold the import dump file and import log file:

   ```
   CREATE DIRECTORY DPUMP_DIR AS '/usr/dpump_dir';
   ```

6. Transfer the Data Pump export dump file hr_schema_dp.dmp to the destination database. You can use the DBMS_FILE_TRANSFER package, binary FTP, or some other method to transfer the file to the destination database. After the file transfer, the export dump file should reside in the directory that corresponds to the directory object created in Step 5.

7. On a command line at the destination database, use Data Pump to import the export dump file hr_schema_dp.dmp. Make sure no changes are made to the tables in the schema being imported at the destination database until the import is complete. Performing the import automatically sets the instantiation SCN for the hr schema and all of its objects at the destination database.

   Perform the import by connecting as an administrative user who is granted IMP_FULL_DATABASE role. This user also must have READ and WRITE privilege on the directory object created in Step 5. This example connects as the Streams administrator strmadmin.

   The following is an example import command:

   ```
   impdp strmadmin/strmadminpw SCHEMAS=hr DIRECTORY=DPUMP_DIR DUMPFILE=hr_schema_
   dp.dmp
   ```

Note: Any table supplemental log groups for the tables exported from the export database are retained when the tables are imported at the import database. You can drop these supplemental log groups if necessary.

See Also: *Oracle Database Utilities* for information about performing a Data Pump import

Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN

The examples in this section describe the steps required to instantiate the database objects in a tablespace using transportable tablespace or RMAN. These instantiation options usually are faster than export/import. The following examples instantiate the database objects in a tablespace:

- "Instantiating Objects Using Transportable Tablespace" on page 10-8 uses the transportable tablespace feature to complete the instantiation. Data Pump exports the tablespace at the source database, and imports the tablespace at the destination database. The tablespace is read-only during the export.
- "Instantiating Objects Using Transportable Tablespace from Backup with RMAN" on page 10-11 uses the RMAN `TRANSPORT TABLESPACE` command to generate a Data Pump export dump file and datafiles for a tablespace or set of tablespaces at the source database while the tablespace or tablespaces remain online. Either Data Pump import or the `ATTACH_TABLESPACES` procedure in the `DBMS_STREAMS_TABLESPACE_ADM` package can add the tablespace or tablespaces to the destination database.

These examples instantiate a tablespace set that includes a tablespace called `jobs_tbs`, and a tablespace called `regions_tbs`. To run the examples, connect to the source database as an administrative user and create the new tablespaces:

```
CREATE TABLESPACE jobs_tbs DATAFILE '/usr/oracle/dbs/jobs_tbs.dbf' SIZE 5 M;

CREATE TABLESPACE regions_tbs DATAFILE '/usr/oracle/dbs/regions_tbs.dbf' SIZE 5 M;
```

Place the new table `hr.jobs_transport` in the `jobs_tbs` tablespace:

```
CREATE TABLE hr.jobs_transport TABLESPACE jobs_tbs AS
  SELECT * FROM hr.jobs;
```

Place the new table `hr.regions_transport` in the `regions_tbs` tablespace:

```
CREATE TABLE hr.regions_transport TABLESPACE regions_tbs AS
  SELECT * FROM hr.regions;
```

Both of the examples make the following assumptions:

- You want to capture all of the changes to the `hr.jobs_transport` and `hr.regions_transport` tables at a source database and apply these changes at a separate destination database.
- The `hr.jobs_transport` exists at a source database, and a single self-contained tablespace named `jobs_tbs` contains the table. The `jobs_tbs` tablespace is stored in a single datafile named `jobs_tbs.dbf`.
- The `hr.regions_transport` exists at a source database, and a single self-contained tablespace named `regions_tbs` contains the table. The `regions_tbs` tablespace is stored in a single datafile named `regions_tbs.dbf`.

- The jobs_tbs and regions_tbs tablespaces do not contain data from any other schemas.
- The hr.jobs_transport table, the hr.regions_transport table, the jobs_tbs tablespace, and the regions_tbs tablespace do not exist at the destination database.
- You have configured a Streams administrator at both the source database and the destination database named strmadmin, and you have granted this Streams administrator DBA role at both databases.

  See Also: *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

Instantiating Objects Using Transportable Tablespace

This example uses transportable tablespace to instantiate the database objects in a tablespace set. In addition to the assumptions listed in "Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN" on page 10-7, this example makes the following assumptions:

- The Streams administrator at the source database is granted the EXP_FULL_DATABASE role to perform the transportable tablespaces export. The DBA role is sufficient because it includes the EXP_FULL_DATABASE role. In this example, the Streams administrator performs the transportable tablespaces export.
- The Streams administrator at the destination database is granted the IMP_FULL_DATABASE role to perform the transportable tablespaces import. The DBA role is sufficient because it includes the IMP_FULL_DATABASE role. In this example, the Streams administrator performs the transportable tablespaces export.

  See Also: *Oracle Database Administrator's Guide* for more information about using transportable tablespaces and for information about limitations that might apply

Complete the following steps to instantiate the database objects in the jobs_tbs tablespace using transportable tablespace:

1. While connected in SQL*Plus to the source database as the Streams administrator strmadmin, create a directory object to hold the export dump file and export log file:

   ```
   CREATE DIRECTORY TRANS_DIR AS '/usr/trans_dir';
   ```

2. While connected as the Streams administrator strmadmin at the source database, prepare the hr.jobs_transport and hr.regions_transport tables for instantiation. You can complete this step in one of the following ways:

   - Add rules for the hr.jobs_transport and hr.regions_transport tables to the positive rule set for a capture process using a procedure in the DBMS_STREAMS_ADM package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then the procedure that you run prepares this table for instantiation automatically.

     For example, the following procedure adds rules to the positive rule set of a capture process named strm01_capture and prepares the hr.jobs_transport table:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name     => 'hr.jobs_transport',
    streams_type   => 'capture',
    streams_name   => 'strm01_capture',
    queue_name     => 'strmadmin.strm01_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

The following procedure adds rules to the positive rule set of a capture process named `strm01_capture` and prepares the `hr.regions_transport` table:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name     => 'hr.regions_transport',
    streams_type   => 'capture',
    streams_name   => 'strm01_capture',
    queue_name     => 'strmadmin.strm01_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

- Add rules for the `hr.jobs_transport` and `hr.regions_transport` tables to the positive rule set for a capture process using a procedure in the `DBMS_RULE_ADM` package, and then prepare these tables for instantiation manually by specifying the table when you run the `PREPARE_TABLE_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
    table_name  => 'hr.jobs_transport');
END;
/

BEGIN
  DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
    table_name  => 'hr.regions_transport');
END;
/
```

Make sure you add the rules to the positive rule set for the capture process before you prepare the database objects for instantiation.

3. While connected the Streams administrator at the source database, make the tablespaces that contain the objects you are instantiating read-only. In this example, the `jobs_tbs` and `regions_tbs` tablespaces contain the database objects.

```
ALTER TABLESPACE jobs_tbs READ ONLY;

ALTER TABLESPACE regions_tbs READ ONLY;
```

4. On a command line, use the Data Pump Export utility to export the `jobs_tbs` and `regions_tbs` tablespaces at the source database using transportable tablespaces export parameters. The following is an example export command that uses transportable tablespaces export parameters:

   ```
   expdp strmadmin/strmadminpw TRANSPORT_TABLESPACES=jobs_tbs, regions_tbs
   DIRECTORY=TRANS_DIR DUMPFILE=tbs_ts.dmp
   ```

   When you run the export command, make sure you connect as an administrative user who was granted `EXP_FULL_DATABASE` role and has `READ` and `WRITE` privileges on the directory object.

   You can also perform an instantiation using transportable tablespaces and the original Export/Import utilities.

   > **See Also:** *Oracle Database Utilities* for information about performing an export

5. While connected to the destination database as the Streams administrator `strmadmin`, create a directory object to hold the import dump file and import log file:

   ```
   CREATE DIRECTORY TRANS_DIR AS '/usr/trans_dir';
   ```

6. Transfer the data files for the tablespaces and the export dump file `tbs_ts.dmp` to the destination database. You can use the `DBMS_FILE_TRANSFER` package, binary FTP, or some other method to transfer these files to the destination database. After the file transfer, the export dump file should reside in the directory that corresponds to the directory object created in Step 5.

7. On a command line at the destination database, use the Data Pump Import utility to import the export dump file `tbs_ts.dmp` using transportable tablespaces import parameters. Performing the import automatically sets the instantiation SCN for the `hr.jobs_transport` and `hr.regions_transport` tables at the destination database.

   The following is an example import command:

   ```
   impdp strmadmin/strmadminpw DIRECTORY=TRANS_DIR DUMPFILE=tbs_ts.dmp
   TRANSPORT_DATAFILES=/usr/orc/dbs/jobs_tbs.dbf,/usr/orc/dbs/regions_tbs.dbf
   ```

   When you run the import command, make sure you connect as an administrative user who was granted `IMP_FULL_DATABASE` role and has `READ` and `WRITE` privileges on the directory object.

   > **See Also:** *Oracle Database Utilities* for information about performing an import

8. If necessary, at both the source database and the destination database, connect as the Streams administrator and put the tablespaces into read/write mode:

   ```
   ALTER TABLESPACE jobs_tbs READ WRITE;

   ALTER TABLESPACE regions_tbs READ WRITE;
   ```

Note: Any table supplemental log groups for the tables exported from the export database are retained when tables are imported at the import database. You can drop these supplemental log groups if necessary.

Instantiating Objects Using Transportable Tablespace from Backup with RMAN

The RMAN TRANSPORT TABLESPACE command uses Data Pump and an RMAN-managed auxiliary instance to export the database objects in a tablespace or tablespace set while the tablespace or tablespace set remains online in the source database. The RMAN TRANSPORT TABLESPACE command produces a Data Pump export dump file and datafiles, and these files can be used to perform a Data Pump import of the tablespace or tablespaces at the destination database. The ATTACH_TABLESPACES procedure in the DBMS_STREAMS_TABLESPACE_ADM package can also be used to attach the tablespace or tablespaces at the destination database.

In addition to the assumptions listed in "Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN" on page 10-7, this example makes the following assumptions:

- The source database is tts1.net.
- The destination database is tts2.net.

See Also: *Oracle Database Backup and Recovery Advanced User's Guide* for instructions on using the RMAN TRANSPORT TABLESPACE command

Complete the following steps to instantiate the database objects in the jobs_tbs and regions_tbs tablespaces using transportable tablespaces and RMAN:

1. Create a backup of the source database that includes the tablespaces being instantiated, if a backup does not exist. RMAN requires a valid backup for tablespace cloning. In this example, create a backup of the source database that includes the jobs_tbs and regions_tbs tablespaces if one does not exist.
2. Optionally, connect in SQL*Plus to the source database as the Streams administrator strmadmin, and create a directory object to hold the export dump file and export log file:

   ```
   CONNECT strmadmin/strmadminpw@tts1.net

   CREATE DIRECTORY SOURCE_DIR AS '/usr/db_files';
   ```

   This step is optional because the RMAN TRANSPORT TABLESPACE command creates a directory object named STREAMS_DIROBJ_DPDIR on the auxiliary instance if the DATAPUMP DIRECTORY parameter is omitted when you run this command in Step 6.
3. While connected as the Streams administrator strmadmin at the source database tts1.net, prepare the hr.jobs_transport and hr.regions_transport tables for instantiation. You can complete this step in one of the following ways:
   - Add rules for the hr.jobs_transport and hr.regions_transport tables to the positive rule set for a capture process using a procedure in the DBMS_STREAMS_ADM package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then the procedure that you run prepares this table for instantiation automatically.

For example, the following procedure adds rules to the positive rule set of a capture process named `strm01_capture` and prepares the `hr.jobs_transport` table:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name     => 'hr.jobs_transport',
    streams_type   => 'capture',
    streams_name   => 'strm01_capture',
    queue_name     => 'strmadmin.strm01_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

The following procedure adds rules to the positive rule set of a capture process named `strm01_capture` and prepares the `hr.regions_transport` table:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name     => 'hr.regions_transport',
    streams_type   => 'capture',
    streams_name   => 'strm01_capture',
    queue_name     => 'strmadmin.strm01_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

- Add rules for the `hr.jobs_transport` and `hr.regions_transport` tables to the positive rule set for a capture process using a procedure in the `DBMS_RULE_ADM` package, and then prepare these tables for instantiation manually by specifying the table when you run the `PREPARE_TABLE_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package:

```
BEGIN
  DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
    table_name  => 'hr.jobs_transport');
END;
/

BEGIN
  DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
    table_name  => 'hr.regions_transport');
END;
/
```

Make sure you add the rules to the positive rule set for the capture process before you prepare the database objects for instantiation.

4. Determine the until SCN for the RMAN TRANSPORT TABLESPACE command:

```
SET SERVEROUTPUT ON SIZE 1000000
DECLARE
  until_scn NUMBER;
BEGIN
  until_scn:= DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
      DBMS_OUTPUT.PUT_LINE('Until SCN: ' || until_scn);
END;
/
```

   Make a note of the until SCN returned. You will use this number in Step 6. For this example, assume that the returned until SCN is 7661956.

   Optionally, you can skip this step. In this case, do not specify the until clause in the RMAN TRANSPORT TABLESPACE command in Step 6. When no until clause is specified, RMAN uses the last archived redo log file to determine the until SCN automatically.

5. Connect to the source database as a system administrator in SQL*Plus and archive the current online redo log:

```
ALTER SYSTEM ARCHIVE LOG CURRENT;
```

6. At the source database tts1.net, use the RMAN TRANSPORT TABLESPACE command to generate the dump file for the tablespace set:

```
RMAN> CONNECT TARGET SYS/change_on_install@tts1.net
RMAN> RUN
      {
        TRANSPORT TABLESPACE 'jobs_tbs', 'regions_tbs'
        UNTIL SCN 7661956
        AUXILIARY DESTINATION '/usr/aux_files'
        DATAPUMP DIRECTORY SOURCE_DIR
        DUMP FILE 'jobs_regions_tbs.dmp'
        EXPORT LOG 'jobs_regions_tbs.log'
        IMPORT SCRIPT 'jobs_regions_tbs_imp.sql'
        TABLESPACE DESTINATION '/orc/dbs';
      }
```

   The TRANSPORT TABLESPACE command places the files in the following directories on the computer system that runs the source database:

   - The directory that corresponds to the SOURCE_DIR directory object (/usr/db_files) contains the export dump file and export log file.
   - The /orc/dbs directory contains the generated datafiles for the tablespaces and the import script. You use this script to complete the instantiation by attaching the tablespace at the destination database.

7. Modify the import script, if necessary. You might need to modify one or both of the following items in the script:

   - You might want to change the method used to make the exported tablespaces part of the destination database. The import script includes two ways to make the exported tablespaces part of the destination database: a Data Pump import command (impdp), and a script for attaching the tablespaces using the ATTACH_TABLESPACES procedure in the DBMS_STREAMS_TABLESPACE_ADM package.

The default script uses the attach tablespaces method. The Data Pump import command is commented out. If you want to use Data Pump import, then remove the comment symbols (`/*` and `*/`) surrounding the impdp command, and either surround the attach tablespaces script with comments or remove the attach tablespaces script. The attach tablespaces script starts with `SET SERVEROUTPUT ON` and continues to the end of the file.

- You might need to change the directory paths specified in the script. In Step 8, you will transfer the import script (`jobs_regions_tbs_imp.sql`), the Data Pump export dump file (`jobs_regions_tbs.dmp`), and the generated datafile for each tablespace (`jobs_tbs.dbf` and `regions_tbs.dbf`) to one or more directories on the computer system running the destination database. Make sure the directory paths specified in the script are the correct directory paths.

8. Transfer the import script (`jobs_regions_tbs_imp.sql`), the Data Pump export dump file (`jobs_regions_tbs.dmp`), and the generated datafile for each tablespace (`jobs_tbs.dbf` and `regions_tbs.dbf`) to the destination database. You can use the `DBMS_FILE_TRANSFER` package, binary FTP, or some other method to transfer the file to the destination database. After the file transfer, these files should reside in the directories specified in the import script.
9. At the destination database, connect as the Streams administrator in SQL*Plus and run the import script:

```
CONNECT strmadmin/strmadminpw@tts2.net

SET ECHO ON
SPOOL jobs_tbs_imp.out
@jobs_tbs_imp.sql
```

When the script completes, check the `jobs_tbs_imp.out` spool file to ensure that all actions finished successfully.

Instantiating Objects Using Original Export/Import

The example in this section describes the steps required to instantiate objects in a Streams environment using original export/import. This example makes the following assumptions:

- You want to capture changes to all of the tables in the `hr` schema at a source database and apply these changes at a separate destination database.
- The `hr` schema exists at both the source database and the destination database. The `hr` schema at the source database contains seven tables. The `hr` schema at the destination database does not contain any tables. For the purposes of this example, you can drop the tables in the `hr` schema at the destination database using the following SQL statements:

```
DROP TABLE hr.countries CASCADE CONSTRAINTS;
DROP TABLE hr.departments CASCADE CONSTRAINTS;
DROP TABLE hr.employees CASCADE CONSTRAINTS;
DROP TABLE hr.job_history CASCADE CONSTRAINTS;
DROP TABLE hr.jobs CASCADE CONSTRAINTS;
DROP TABLE hr.locations CASCADE CONSTRAINTS;
DROP TABLE hr.regions CASCADE CONSTRAINTS;
```

The import re-creates these tables at the destination database.

- You have configured a Streams administrator at the source database named `strmadmin`.

  See Also: *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

Given these assumptions, complete the following steps to instantiate the `hr` schema using original export/import:

1. While connected in SQL*Plus as the Streams administrator `strmadmin` at the source database, prepare the database objects in the `hr` schema for instantiation. You can complete this step in one of the following ways:

   - Add rules for the `hr` schema to the positive rule set for a capture process using a procedure in the `DBMS_STREAMS_ADM` package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then the procedure that you run prepares the objects in the `hr` schema for instantiation automatically.

     For example, the following procedure adds rules to the positive rule set of a capture process named `strm01_capture` and prepares the `hr` schema, and all of its objects, for instantiation:

     ```
     BEGIN
       DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
         schema_name     =>  'hr',
         streams_type    =>  'capture',
         streams_name    =>  'strm01_capture',
         queue_name      =>  'strm01_queue',
         include_dml     =>  true,
         include_ddl     =>  true,
         inclusion_rule  =>  true);
     END;
     /
     ```

   - Add rules for the `hr` schema to the positive rule set for a capture process using a procedure in the `DBMS_RULE_ADM` package, and then prepare the objects for instantiation manually by specifying the `hr` schema when you run the `PREPARE_SCHEMA_INSTANTIATION` procedure in the `DBMS_CAPTURE_ADM` package:

     ```
     BEGIN
       DBMS_CAPTURE_ADM.PREPARE_SCHEMA_INSTANTIATION(
         schema_name  => 'hr');
     END;
     /
     ```

     Make sure you add the rules to the positive rule set for the capture process before you prepare the database objects for instantiation.

2. On the command line, use the original Export utility to export the tables in the `hr` schema at the source database. Make sure no DDL changes are made to the tables during the export.

   The following is an example export command:

   ```
   exp hr/hr FILE=hr_schema.dmp CONSISTENT=y
   TABLES=countries,departments,employees,jobs,job_history,locations,regions
   ```

Because the hr schema includes foreign key constraints between tables, the CONSISTENT export parameter is set to y to ensure consistency between all of the objects in the schema. The OBJECT_CONISTENT export parameter is not used because the CONSISTENT export parameter provides a more stringent level of consistency.

> **See Also:** *Oracle Database Utilities* for information about performing an export using the original Export utility

3. Transfer the export dump file hr_schema.dmp to the destination database. You can use the DBMS_FILE_TRANSFER package, binary FTP, or some other method to transfer the to the destination database.
4. At the destination database, use the original Import utility to import the export dump file hr_schema.dmp. When you run the import command, make sure you set the STREAMS_INSTANTIATION import parameter to y. This parameter ensures that the import records instantiation SCN information for each object imported. Also, make sure no changes are made to the tables in the schema being imported at the destination database until the import is complete. Performing the import automatically sets the instantiation SCN for each table in the hr schema at the destination database.

   The following is an example import command:

```
imp hr/hr FILE=hr_schema.dmp FULL=y COMMIT=y STREAMS_INSTANTIATION=y
LOG=import.log
```

> **Note:** Any table supplemental log groups for the tables exported from the export database are retained when the tables are imported at the import database. You can drop these supplemental log groups if necessary.

> **See Also:** *Oracle Database Utilities* for information about performing an import using the original Import utility

Instantiating an Entire Database Using RMAN

The examples in this section describe the steps required to instantiate an entire database using the Recovery Manager (RMAN) DUPLICATE command or CONVERT DATABASE command. When you use one of these RMAN commands for full database instantiation, you perform the following general steps:

1. Copy the entire source database to the destination site using the RMAN command.
2. Remove the Streams configuration at the destination site using the REMOVE_STREAMS_CONFIGURATION procedure in the DBMS_STREAMS_ADM package.
3. Configure Streams destination site, including configuration of one or more apply processes to apply changes from the source database.

You can complete this process without stopping any running capture processes or propagations at the source database.

The RMAN DUPLICATE command can be used for instantiation when the source and destination databases are running on the same platform. The RMAN CONVERT DATABASE command can be used for instantiation when the source and destination databases are running on different platforms. Follow the instructions in one of these sections:

- Instantiating an Entire Database on the Same Platform Using RMAN
- Instantiating an Entire Database on Different Platforms Using RMAN

Note:

- If you want to configure a Streams replication environment that replicates all of the supported changes for an entire database, then the PRE_INSTANTIATION_SETUP and POST_INSTANTIATION_SETUP procedures in the DBMS_STREAMS_ADM package can be used. See "Configuring Database Replication Using the DBMS_STREAMS_ADM Package" on page 6-17 for instructions.
- Oracle recommends that you do not use RMAN for instantiation in an environment where distributed transactions are possible. Doing so can cause in-doubt transactions that must be corrected manually. Use export/import or transportable tablespaces for instantiation instead.

See Also: *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

Instantiating an Entire Database on the Same Platform Using RMAN

The example in this section instantiates an entire database using the RMAN DUPLICATE command. The example makes the following assumptions:

- You want to capture all of the changes made to a source database named dpx1.net, propagate these changes to a separate destination database named dpx2.net, and apply these changes at the destination database.
- You have configured a Streams administrator at the source database named strmadmin.
- The dpx1.net and dpx2.net databases run on the same platform.

See Also: *Oracle Database Backup and Recovery Advanced User's Guide* for instructions on using the RMAN DUPLICATE command

Complete the following steps to instantiate an entire database using RMAN when the source and destination databases run on the same platform:

1. Create a backup of the source database if one does not exist. RMAN requires a valid backup for duplication. In this example, create a backup of dpx1.net if one does not exist.
2. While connected in SQL*Plus as the Streams administrator strmadmin at the source database, create an ANYDATA queue to stage the changes from the source database if such a queue does not already exist. This queue will stage changes that will be propagated to the destination database after it has been configured.

   For example, the following procedure creates a queue named streams_queue:

   ```
   EXEC DBMS_STREAMS_ADM.SET_UP_QUEUE();
   ```

Remain connected as the Streams administrator in SQL*Plus at the source database through Step 8.

3. Create a database link from `dpx1.net` to `dpx2.net`:

```
CREATE DATABASE LINK dpx2.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'dpx2.net';
```

4. Create a propagation from the source queue at the source database to the destination queue at the destination database. The destination queue at the destination database does not exist yet, but creating this propagation ensures that LCRs enqueued into the source queue will remain staged there until propagation is possible. In addition to captured LCRs, the source queue will stage internal messages that will populate the Streams data dictionary at the destination database.

   The following procedure creates the `dpx1_to_dpx2` propagation:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES(
    streams_name            => 'dpx1_to_dpx2',
    source_queue_name       => 'strmadmin.streams_queue',
    destination_queue_name  => 'strmadmin.streams_queue@dpx2.net',
    include_dml             => true,
    include_ddl             => true,
    source_database         => 'dpx1.net',
    inclusion_rule          => true,
    queue_to_queue          => true);
END;
/
```

5. Stop the propagation you created in Step 4.

```
BEGIN
  DBMS_PROPAGATION_ADM.STOP_PROPAGATION(
    propagation_name  => 'dpx1_to_dpx2');
END;
/
```

6. Prepare the entire source database for instantiation, if it has not been prepared for instantiation previously. If there is no capture process that captures all of the changes to the source database, then create this capture process using the `ADD_GLOBAL_RULES` procedure in the `DBMS_STREAMS_ADM` package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then running this procedure automatically prepares the entire source database for instantiation. If such a capture process already exists, then make sure the source database has been prepared for instantiation by querying the `DBA_CAPTURE_PREPARED_DATABASE` data dictionary view.

   If you need to create a capture process, then this example creates the `capture_db` capture process if it does not already exist:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_GLOBAL_RULES(
    streams_type   => 'capture',
    streams_name   => 'capture_db',
    queue_name     => 'strmadmin.streams_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

If the capture process already exists and you need to prepare the entire database for instantiation, then run the following procedure:

```
EXEC DBMS_CAPTURE_ADM.PREPARE_GLOBAL_INSTANTIATION();
```

7. If you created a capture process in Step 6, then start the capture process:

```
BEGIN
  DBMS_CAPTURE_ADM.START_CAPTURE(
    capture_name  => 'capture_db');
END;
/
```

8. Determine the until SCN for the RMAN `DUPLICATE` command:

```
SET SERVEROUTPUT ON SIZE 1000000
DECLARE
  until_scn NUMBER;
BEGIN
  until_scn:= DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
      DBMS_OUTPUT.PUT_LINE('Until SCN: ' || until_scn);
END;
/
```

Make a note of the until SCN returned. You will use this number in Step 11. For this example, assume that the returned until SCN is `3050191`.

9. Connect to the source database as a system administrator in SQL*Plus and archive the current online redo log:

```
ALTER SYSTEM ARCHIVE LOG CURRENT;
```

10. Prepare your environment for database duplication, which includes preparing the destination database as an auxiliary instance for duplication. See *Oracle Database Backup and Recovery Advanced User's Guide* for instructions.

11. Use the RMAN `DUPLICATE` command with the `OPEN RESTRICTED` option to instantiate the source database at the destination database. The `OPEN RESTRICTED` option is required. This option enables a restricted session in the duplicate database by issuing the following SQL statement: `ALTER SYSTEM ENABLE RESTRICTED SESSION`. RMAN issues this statement immediately before the duplicate database is opened.

    You can use the `UNTIL SCN` clause to specify an SCN for the duplication. Use the until SCN determined in Step 8 for this clause. The until SCN specified for the RMAN `DUPLICATE` command must be higher than the SCN when the database was prepared for instantiation in Step 6. Also, archived redo logs must be available for the until SCN specified and for higher SCN values. Therefore, Step 9 archived the redo log containing the until SCN.

Make sure you use TO *database_name* in the DUPLICATE command to specify the name of the duplicate database. In this example, the duplicate database name is dpx2. Therefore, the DUPLICATE command for this example includes TO dpx2.

The following is an example of an RMAN DUPLICATE command:

```
rman
RMAN> CONNECT TARGET SYS/change_on_install@dpx1.net
RMAN> CONNECT AUXILIARY SYS/change_on_install@dpx2.net
RMAN> RUN
      {
        SET UNTIL SCN 3050191;
        ALLOCATE AUXILIARY CHANNEL dpx2 DEVICE TYPE sbt;
        DUPLICATE TARGET DATABASE TO dpx2
        NOFILENAMECHECK
        OPEN RESTRICTED;
      }
```

12. At the destination database, connect as an administrative user in SQL*Plus and rename the database global name. After the RMAN DUPLICATE command, the destination database has the same global name as the source database.

```
ALTER DATABASE RENAME GLOBAL_NAME TO DPX2.NET;
```

13. At the destination database, connect as an administrator with SYSDBA privilege in SQL*Plus and run the following procedure:

> **Attention:** Make sure you are connected to the destination database, not the source database, when you run this procedure because it removes the local Streams configuration.

```
EXEC DBMS_STREAMS_ADM.REMOVE_STREAMS_CONFIGURATION();
```

> **Note:** Any supplemental log groups for the tables at the source database are retained at the destination database, and the REMOVE_STREAMS_CONFIGURATION procedure does not drop them. You can drop these supplemental log groups if necessary.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the REMOVE_STREAMS_CONFIGURATION procedure

14. At the destination database, use the ALTER SYSTEM statement to disable the RESTRICTED SESSION:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

15. At the destination database, create the queue specified in Step 4. For example:

    For example, the following procedure creates a queue named streams_queue:

```
EXEC DBMS_STREAMS_ADM.SET_UP_QUEUE();
```

16. At the destination database, connect as the Streams administrator and configure the Streams environment.

    Attention: Do not start any apply processes at the destination database until after you set the global instantiation SCN in Step 18.

    See Also: *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

17. At the destination database, create a database link from the destination database to the source database:

```
CREATE DATABASE LINK dpx1.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'dpx1.net';
```

    This database link is required because the next step runs the `SET_GLOBAL_INSTANTIATION_SCN` procedure with the recursive parameter set to `true`.

18. At the destination database, set the global instantiation SCN for the source database. The RMAN `DUPLICATE` command duplicates the database up to one less than the SCN value specified in the `UNTIL SCN` clause. Therefore, you should subtract one from the until SCN value that you specified when you ran the `DUPLICATE` command in Step 11. In this example, the until SCN was set to `3050191`. Therefore, the instantiation SCN should be set to `3050191 - 1`, or `3050190`.

    For example, to set the global instantiation SCN to `3050190` for the `dpx1.net` source database, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_GLOBAL_INSTANTIATION_SCN(
    source_database_name   =>  'dpx1.net',
    instantiation_scn      =>  3050190,
    recursive              =>  true);
END;
/
```

    Notice that the `recursive` parameter is set to `true` to set the instantiation SCN for all schemas and tables in the destination database.

19. At the destination database, you can start any apply processes that you configured.

20. At the source database, start the propagation you stopped in Step 5:

```
BEGIN
  DBMS_PROPAGATION_ADM.START_PROPAGATION(
    queue_name  => 'dpx1_to_dpx2');
END;
/
```

Instantiating an Entire Database on Different Platforms Using RMAN

The example in this section instantiates an entire database using the RMAN CONVERT DATABASE command. The example makes the following assumptions:

- You want to capture all of the changes made to a source database named cvx1.net, propagate these changes to a separate destination database named cvx2.net, and apply these changes at the destination database.
- You have configured a Streams administrator at the source database named strmadmin.
- The cvx1.net and cvx2.net databases run on different platforms, and the platform combination is supported by the RMAN CONVERT DATABASE command. You can use the DBMS_TDB package to determine whether a platform combination is supported.

The RMAN CONVERT DATABASE command produces converted datafiles, an initialization parameter file (PFILE), and a SQL script. The converted datafiles and PFILE are for use with the destination database, and the SQL script creates the destination database on the destination platform.

> **See Also:** *Oracle Database Backup and Recovery Advanced User's Guide* for instructions on using the RMAN CONVERT DATABASE command

Complete the following steps to instantiate an entire database using RMAN when the source and destination databases run on different platforms:

1. Create a backup of the source database if one does not exist. RMAN requires a valid backup. In this example, create a backup of cvx1.net if one does not exist.
2. While connected in SQL*Plus as the Streams administrator strmadmin at the source database, create an ANYDATA queue to stage the changes from the source database if such a queue does not already exist. This queue will stage changes that will be propagated to the destination database after it has been configured.

   For example, the following procedure creates a queue named streams_queue:

   ```
   EXEC DBMS_STREAMS_ADM.SET_UP_QUEUE();
   ```

   Remain connected as the Streams administrator in SQL*Plus at the source database through Step 7.
3. Create a database link from cvx1.net to cvx2.net:

   ```
   CREATE DATABASE LINK cvx2.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
     USING 'cvx2.net';
   ```

4. Create a propagation from the source queue at the source database to the destination queue at the destination database. The destination queue at the destination database does not exist yet, but creating this propagation ensures that LCRs enqueued into the source queue will remain staged there until propagation is possible. In addition to captured LCRs, the source queue will stage internal messages that will populate the Streams data dictionary at the destination database.

The following procedure creates the cvx1_to_cvx2 propagation:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES(
    streams_name            => 'cvx1_to_cvx2',
    source_queue_name       => 'strmadmin.streams_queue',
    destination_queue_name  => 'strmadmin.streams_queue@cvx2.net',
    include_dml             => true,
    include_ddl             => true,
    source_database         => 'cvx1.net',
    inclusion_rule          => true,
    queue_to_queue          => true);
END;
/
```

5. Stop the propagation you created in Step 4.

```
BEGIN
  DBMS_PROPAGATION_ADM.STOP_PROPAGATION(
    propagation_name  => 'cvx1_to_cvx2');
END;
/
```

6. Prepare the entire source database for instantiation, if it has not been prepared for instantiation previously. If there is no capture process that captures all of the changes to the source database, then create this capture process using the ADD_GLOBAL_RULES procedure in the DBMS_STREAMS_ADM package. If the capture process is a local capture process or a downstream capture process with a database link to the source database, then running this procedure automatically prepares the entire source database for instantiation. If such a capture process already exists, then make sure the source database has been prepared for instantiation by querying the DBA_CAPTURE_PREPARED_DATABASE data dictionary view.

   If you need to create a capture process, then this example creates the capture_db capture process if it does not already exist:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_GLOBAL_RULES(
    streams_type   => 'capture',
    streams_name   => 'capture_db',
    queue_name     => 'strmadmin.streams_queue',
    include_dml    => true,
    include_ddl    => true,
    inclusion_rule => true);
END;
/
```

   If the capture process already exists, and you need to prepare the entire database for instantiation, then run the following procedure:

```
EXEC DBMS_CAPTURE_ADM.PREPARE_GLOBAL_INSTANTIATION();
```

7. If you created a capture process in Step 6, then start the capture process:

```
BEGIN
  DBMS_CAPTURE_ADM.START_CAPTURE(
    capture_name  => 'capture_db');
END;
/
```

8. Connect to the source database as a system administrator in SQL*Plus and archive the current online redo log:

```
ALTER SYSTEM ARCHIVE LOG CURRENT;
```

9. Prepare your environment for database conversion, which includes opening the source database in read-only mode. Complete the following steps:

   a. If the source database is open, then shut it down and start it in read-only mode.

   b. Run the CHECK_DB and CHECK_EXTERNAL functions in the DBMS_TDB package. Check the results to ensure that the conversion is supported by the RMAN CONVERT DATABASE command.

      See Also: *Oracle Database Backup and Recovery Advanced User's Guide* for more information about these steps

10. Determine the current SCN of the source database:

```
SET SERVEROUTPUT ON SIZE 1000000
DECLARE
  current_scn NUMBER;
BEGIN
  current_scn:= DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
      DBMS_OUTPUT.PUT_LINE('Current SCN: ' || current_scn);
END;
/
```

    Make a note of the SCN value returned. You will use this number in Step 20. For this example, assume that the returned value is 46931285.

11. Open RMAN and run the CONVERT DATABASE command.

    Make sure you use NEW DATABASE *database_name* in the CONVERT DATABASE command to specify the name of the destination database. In this example, the destination database name is cvx2. Therefore, the CONVERT DATABASE command for this example includes NEW DATABASE cvx2.

    The following is an example of an RMAN CONVERT DATABASE command for a destination database that is running on the Linux IA (64-bit) platform:

```
rman
RMAN> CONNECT TARGET SYS/change_on_install@cvx1.net
CONVERT DATABASE NEW DATABASE 'cvx2'
          TRANSPORT SCRIPT '/tmp/convertdb/transportscript.sql'
          TO PLATFORM 'Linux IA (64-bit)'
          DB_FILE_NAME_CONVERT '/home/oracle/dbs','/tmp/convertdb';
```

12. Transfer the datafiles, PFILE, and SQL script produced by the RMAN CONVERT DATABASE command to the computer system that will run the destination database.

13. On the computer system that will run the destination database, modify the SQL script so that the destination database always opens with restricted session enabled.

    The following is an example script with the necessary modifications in bold font:

```
-- The following commands will create a new control file and use it
-- to open the database.
-- Data used by Recovery Manager will be lost.
-- The contents of online logs will be lost and all backups will
```

```
-- be invalidated. Use this only if online logs are damaged.

-- After mounting the created controlfile, the following SQL
-- statement will place the database in the appropriate
-- protection mode:
--  ALTER DATABASE SET STANDBY DATABASE TO MAXIMIZE PERFORMANCE

STARTUP NOMOUNT PFILE='init_00gd2lak_1_0.ora'
CREATE CONTROLFILE REUSE SET DATABASE "CVX2" RESETLOGS  NOARCHIVELOG
    MAXLOGFILES 32
    MAXLOGMEMBERS 2
    MAXDATAFILES 32
    MAXINSTANCES 1
    MAXLOGHISTORY 226
LOGFILE
  GROUP 1 '/tmp/convertdb/archlog1'  SIZE 25M,
  GROUP 2 '/tmp/convertdb/archlog2'  SIZE 25M
DATAFILE
  '/tmp/convertdb/systemdf',
  '/tmp/convertdb/sysauxdf',
  '/tmp/convertdb/datafile1',
  '/tmp/convertdb/datafile2',
  '/tmp/convertdb/datafile3'
CHARACTER SET WE8DEC
;

-- NOTE: This ALTER SYSTEM statement is added to enable restricted session.

ALTER SYSTEM ENABLE RESTRICTED SESSION;

-- Database can now be opened zeroing the online logs.
ALTER DATABASE OPEN RESETLOGS;

-- No tempfile entries found to add.
--

set echo off
prompt ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
prompt * Your database has been created successfully!
prompt * There are many things to think about for the new database. Here
prompt * is a checklist to help you stay on track:
prompt * 1. You may want to redefine the location of the directory objects.
prompt * 2. You may want to change the internal database identifier (DBID)
prompt *    or the global database name for this database. Use the
prompt *    NEWDBID Utility (nid).
prompt ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SHUTDOWN IMMEDIATE
-- NOTE: This startup has the UPGRADE parameter.
-- It already has restricted session enabled, so no change is needed.
STARTUP UPGRADE PFILE='init_00gd2lak_1_0.ora'
@@ ?/rdbms/admin/utlirp.sql
SHUTDOWN IMMEDIATE
-- NOTE: The startup below is generated without the RESTRICT clause.
-- Add the RESTRICT clause.
STARTUP RESTRICT PFILE='init_00gd2lak_1_0.ora'
-- The following step will recompile all PL/SQL modules.
-- It may take serveral hours to complete.
@@ ?/rdbms/admin/utlrp.sql
set feedback 6;
```

Other changes to the script might be necessary. For example, the datafile locations and PFILE location might need to be changed to point to the correct locations on the destination database computer system.

14. At the destination database, connect as an administrator with SYSDBA privilege in SQL*Plus and run the following procedure:

    Attention: Make sure you are connected to the destination database, not the source database, when you run this procedure because it removes the local Streams configuration.

```
EXEC DBMS_STREAMS_ADM.REMOVE_STREAMS_CONFIGURATION();
```

    Note: Any supplemental log groups for the tables at the source database are retained at the destination database, and the REMOVE_STREAMS_CONFIGURATION procedure does not drop them. You can drop these supplemental log groups if necessary.

    See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about the REMOVE_STREAMS_CONFIGURATION procedure

15. Connect to the destination database as the Streams administrator, and drop the database link from the source database to the destination database that was cloned from the source database:

```
CONNECT strmadmin/strmadminpw@cvx2.net

DROP DATABASE LINK cvx2.net;
```

16. At the destination database, use the ALTER SYSTEM statement to disable the RESTRICTED SESSION:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

17. At the destination database, create the queue specified in Step 4.

    For example, the following procedure creates a queue named streams_queue:

```
EXEC DBMS_STREAMS_ADM.SET_UP_QUEUE();
```

18. At the destination database, connect as the Streams administrator and configure the Streams environment.

    Attention: Do not start any apply processes at the destination database until after you set the global instantiation SCN in Step 20.

    See Also: *Oracle Streams Concepts and Administration* for information about configuring a Streams administrator

19. At the destination database, create a database link to the source database:

```
CREATE DATABASE LINK cvx1.net CONNECT TO strmadmin IDENTIFIED BY strmadminpw
   USING 'cvx1.net';
```

This database link is required because the next step runs the SET_GLOBAL_INSTANTIATION_SCN procedure with the recursive parameter set to true.

20. At the destination database, set the global instantiation SCN for the source database to the SCN value returned in Step 10.

For example, to set the global instantiation SCN to 46931285 for the cvx1.net source database, run the following procedure:

```
BEGIN
  DBMS_APPLY_ADM.SET_GLOBAL_INSTANTIATION_SCN(
    source_database_name   =>  'cvx1.net',
    instantiation_scn      =>  46931285,
    recursive              =>  true);
END;
/
```

Notice that the recursive parameter is set to true to set the instantiation SCN for all schemas and tables in the destination database.

21. At the destination database, you can start any apply processes that you configured.

22. At the source database, start the propagation you stopped in Step 5:

```
BEGIN
  DBMS_PROPAGATION_ADM.START_PROPAGATION(
    propagation_name  => 'cvx1_to_cvx2');
END;
/
```

Setting Instantiation SCNs at a Destination Database

An instantiation SCN instructs an apply process at a destination database to apply changes to a database object that committed after a specific SCN at a source database. You can set instantiation SCNs in one of the following ways:

- Export the relevant database objects at the source database and import them into the destination database. In this case, the export/import creates the database objects at the destination database, populates them with the data from the source database, and sets the relevant instantiation SCNs. You can use Data Pump export/import or original export/import for instantiations. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for information about the instantiation SCNs that are set for different types of export/import operations.
- Perform a metadata only export/import using Data Pump or original export/import. If you use Data Pump export/import, then set the CONTENT parameter to METADATA_ONLY during export at the source database or import at the destination database, or both. If you use original export/import, then set the ROWS parameter to n during export at the source database or import at the destination database, or both. In either case, instantiation SCNs are set for the database objects, but no data is imported. See "Setting Instantiation SCNs Using Export/Import" on page 10-28 for information about the instantiation SCNs that are set for different types of export/import operations.

- Use transportable tablespaces to copy the objects in one or more tablespaces from a source database to a destination database. An instantiation SCN is set for each schema in these tablespaces and for each database object in these tablespaces that was prepared for instantiation before the export. See "Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN" on page 10-7.
- Set the instantiation SCN using the `SET_TABLE_INSTANTIATION_SCN`, `SET_SCHEMA_INSTANATIATION_SCN`, and `SET_GLOBAL_INSTANTIATION_SCN` procedures in the `DBMS_APPLY_ADM` package. See "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29.

See Also:

- "Instantiation SCN and Ignore SCN for an Apply Process" on page 1-27
- "Overview of Instantiation and Streams Replication" on page 2-1
- "Instantiating Objects in a Streams Replication Environment" on page 10-3

Setting Instantiation SCNs Using Export/Import

This section discusses setting instantiation SCNs by performing an export/import. The information in this section applies to both metadata export/import operations and to export/import operations that import rows. Also, you can use either Data Pump export/import or original export/import.

If you use the original Export utility, then set the `OBJECT_CONSISTENT` export parameter to `y`. Regardless of whether you use Data Pump export or original export, you can specify a more stringent degree of consistency by using an export parameter such as `FLASHBACK_SCN` or `FLASHBACK_TIME`. Also, if you use the original Import utility, then set the `STREAMS_INSTANTIATION` import parameter to `y`.

The following sections describe how the instantiation SCNs are set for different types of export/import operations. These sections refer to **prepared tables**. Prepared tables are tables that have been prepared for instantiation using the `PREPARE_TABLE_INSTANTIATION`, `PREPARE_SCHEMA_INSTANTIATION`, or `PREPARE_GLOBAL_INSTANTIATION` procedures in the `DBMS_CAPTURE_ADM` package. A table must be a prepared table before export in order for an instantiation SCN to be set for it during import. However, the database and schemas do not need to be prepared before the export in order for their instantiation SCNs to be set during import.

Full Database Export and Full Database Import

A full database export and full database import sets the following instantiation SCNs at the import database:

- The database, or global, instantiation SCN
- The schema instantiation SCN for each imported user
- The table instantiation SCNs for each prepared table that is imported

Full Database or User Export and User Import

A full database or user export and user import sets the following instantiation SCNs at the import database:

- The schema instantiation SCN for each imported user
- The table instantiation SCN for each prepared table that is imported

Full Database, User, or Table Export and Table Import

Any export that includes one or more tables and a table import sets the table instantiation SCN for each prepared table that is imported at the import database.

Note:

- If a non-`NULL` instantiation SCN already exists for a database object at a destination database that performs an import, then the import updates the instantiation SCN for that database object.
- During an export for a Streams instantiation, make sure no DDL changes are made to objects being exported.
- Any table supplemental logging specifications for the tables exported from the export database are retained when the tables are imported at the import database.

See Also:

- "Oracle Data Pump and Streams Instantiation" on page 2-7, "Original Export/Import and Streams Instantiation" on page 2-12, and *Oracle Database Utilities* for information about using export/import
- Part II, "Configuring Streams Replication" for more information about performing export/import operations to set instantiation SCNs when configuring a Streams environment
- "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package

You can set an instantiation SCN at a destination database for a specified table, a specified schema, or an entire database using one of the following procedures in the `DBMS_APPLY_ADM` package:

- `SET_TABLE_INSTANTIATION_SCN`
- `SET_SCHEMA_INSTANTIATION_SCN`
- `SET_GLOBAL_INSTANTIATION_SCN`

If you set the instantiation SCN for a schema using `SET_SCHEMA_INSTANTIATION_SCN`, then you can set the `recursive` parameter to `true` when you run this procedure to set the instantiation SCN for each table in the schema. Similarly, if you set the instantiation SCN for a database using `SET_GLOBAL_INSTANTIATION_SCN`, then you can set the `recursive` parameter to `true` when you run this procedure to set the instantiation SCN for the schemas in the database and for each table owned by these schemas.

Note:

- If you set the `recursive` parameter to `true` in the `SET_SCHEMA_INSTANTIATION_SCN` procedure or the `SET_GLOBAL_INSTANTIATION_SCN` procedure, then a database link from the destination database to the source database is required. This database link must have the same name as the global name of the source database and must be accessible to the user who executes the procedure.
- If a relevant instantiation SCN is not present, then an error is raised during apply.

Table 10–1 lists each procedure and the types of statements for which they set an instantiation SCN.

Table 10–1 Set Instantiation SCN Procedures and the Statements They Cover

Procedure	Sets Instantiation SCN for	Examples

`SET_TABLE_INSTANTIATION_SCN`	DML and DDL statements on tables, except `CREATE TABLE`
	DDL statements on table indexes and table triggers
		`UPDATE`
		`ALTER TABLE`
		`DROP TABLE`
		`CREATE`, `ALTER`, or `DROP INDEX` on a table
		`CREATE`, `ALTER`, or `DROP TRIGGER` on a table

`SET_SCHEMA_INSTANTIATION_SCN`	DDL statements on users, except `CREATE USER`
	DDL statements on all database objects that have a non-`PUBLIC` owner, except for those DDL statements handled by a table-level instantiation SCN
		`CREATE TABLE`
		`ALTER USER`
		`DROP USER`
		`CREATE PROCEDURE`

`SET_GLOBAL_INSTANTIATION_SCN`	DDL statements on database objects other than users with no owner DDL statements on database objects owned by public `CREATE USER` statements	`CREATE USER` `CREATE TABLESPACE`

Setting the Instantiation SCN While Connected to the Source Database

The user who runs the examples in this section must have access to a database link from the source database to the destination database. In these example, the database link is `hrdb2.net`. The following example sets the instantiation SCN for the `hr.departments` table at the `hrdb2.net` database to the current SCN by running the following procedure at the source database `hrdb1.net`:

```
DECLARE
  iscn  NUMBER;         -- Variable to hold instantiation SCN value
BEGIN
  iscn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();
  DBMS_APPLY_ADM.SET_TABLE_INSTANTIATION_SCN@HRDB2.NET(
    source_object_name    => 'hr.departments',
    source_database_name  => 'hrdb1.net',
    instantiation_scn     => iscn);
END;
/
```

The following example sets the instantiation SCN for the `oe` schema and all of its objects at the `hrdb2.net` database to the current source database SCN by running the following procedure at the source database `hrdb1.net`:

```
DECLARE
  iscn  NUMBER;         -- Variable to hold instantiation SCN value
BEGIN
  iscn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();
  DBMS_APPLY_ADM.SET_SCHEMA_INSTANTIATION_SCN@HRDB2.NET(
    source_schema_name    => 'oe',
    source_database_name  => 'hrdb1.net',
    instantiation_scn     => iscn,
    recursive             => true);
END;
/
```

Because the `recursive` parameter is set to `true`, running this procedure sets the instantiation SCN for each database object in the `oe` schema.

> **Note:** When you set the `recursive` parameter to `true`, a database link from the destination database to the source database is required, even if you run the procedure while you are connected to the source database. This database link must have the same name as the global name of the source database and must be accessible to the current user.

Setting the Instantiation SCN While Connected to the Destination Database

The user who runs the examples in this section must have access to a database link from the destination database to the source database. In these example, the database link is `hrdb1.net`. The following example sets the instantiation SCN for the `hr.departments` table at the `hrdb2.net` database to the current source database SCN at `hrdb1.net` by running the following procedure at the destination database `hrdb2.net`:

```
DECLARE
  iscn  NUMBER;         -- Variable to hold instantiation SCN value
BEGIN
  iscn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER@HRDB1.NET;
  DBMS_APPLY_ADM.SET_TABLE_INSTANTIATION_SCN(
    source_object_name    => 'hr.departments',
    source_database_name  => 'hrdb1.net',
    instantiation_scn     => iscn);
END;
/
```

The following example sets the instantiation SCN for the oe schema and all of its objects at the hrdb2.net database to the current source database SCN at hrdb1.net by running the following procedure at the destination database hrdb2.net:

```
DECLARE
  iscn  NUMBER;         -- Variable to hold instantiation SCN value
BEGIN
  iscn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER@HRDB1.NET;
  DBMS_APPLY_ADM.SET_SCHEMA_INSTANTIATION_SCN(
    source_schema_name    => 'oe',
    source_database_name  => 'hrdb1.net',
    instantiation_scn     => iscn,
    recursive             => true);
END;
/
```

Because the recursive parameter is set to true, running this procedure sets the instantiation SCN for each database object in the oe schema.

Note: If an apply process applies changes to a remote non-Oracle database, then set the apply_database_link parameter to the database link used for remote apply when you set the instantiation SCN.

See Also:

- Part II, "Configuring Streams Replication" for more information when to set instantiation SCNs when you are configuring a Streams environment
- Chapter 15, "Single-Source Heterogeneous Replication Example" and Chapter 16, "Multiple-Source Replication Example" for detailed examples that uses the SET_TABLE_INSTANTIATION_SCN procedure
- The information about the DBMS_APPLY_ADM package in the *Oracle Database PL/SQL Packages and Types Reference* for more information about which instantiation SCN can be used for a DDL LCR

11

Managing Logical Change Records (LCRs)

This chapter contains instructions for managing logical change records (LCRs) in a Streams replication environment.

This chapter contains these topics:

- Requirements for Managing LCRs
- Constructing and Enqueuing LCRs
- Executing LCRs
- Managing LCRs Containing LOB Columns
- Managing LCRs Containing LONG or LONG RAW Columns

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* and *Oracle Streams Concepts and Administration* for more information about LCRs

Requirements for Managing LCRs

This section describes requirements for creating or modifying LCRs. You can create an LCR using a constructor for an LCR type, and then enqueue the LCR into an `ANYDATA` queue. Such an LCR is a user-enqueued LCR.

Also, you can modify an LCR using an apply handler or a rule-based transformation. You can modify both LCRs captured by a capture process and LCRs constructed and enqueued by a user or application.

Make sure you meet the following requirements when you manage an LCR:

- If you create or modify a row LCR, then make sure the `command_type` attribute is consistent with the presence or absence of old column values and the presence or absence of new column values.
- If you create or modify a DDL LCR, then make sure the `ddl_text` is consistent with the `base_table_name`, `base_table_owner`, `object_type`, `object_owner`, `object_name`, and `command_type` attributes.
- The following datatypes are allowed for columns in a user-constructed row LCR:
  - `CHAR`
  - `VARCHAR2`
  - `NCHAR`
  - `NVARCHAR2`
  - `NUMBER`

- DATE
- BINARY_FLOAT
- BINARY_DOUBLE
- RAW
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

These datatypes are the only datatypes allowed for columns in a user-constructed row LCR. However, you can use certain techniques to construct LCRs that contain LOB information. Also, LCRs captured by a capture process support more datatypes.

See Also:

- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers
- "Managing LCRs Containing LOB Columns" on page 11-12
- *Oracle Streams Concepts and Administration* for information about the datatypes captured by a capture process and for information about rule-based transformations

Constructing and Enqueuing LCRs

Use the following LCR constructors to create LCRs:

- To create a row LCR that contains a change to a row that resulted from a data manipulation language (DML) statement, use the `SYS.LCR$_ROW_RECORD` constructor.
- To create a DDL LCR that contains a data definition language change, use the `SYS.LCR$_DDL_RECORD` constructor. Make sure the DDL text specified in the `ddl_text` attribute of each DDL LCR conforms to Oracle SQL syntax.

The following example creates a queue in an Oracle database and an apply process associated with the queue. Next, it creates a PL/SQL procedure that constructs a row LCR based on information passed to it and enqueues the row LCR into the queue. This example assumes that you have configured a Streams administrator named `strmadmin` and granted this administrator `DBA` role.

Complete the following steps:

1. While connected as an administrative user, grant the Streams administrator `EXECUTE` privilege on the `DBMS_STREAMS_MESSAGING` package. For example:

   ```
   GRANT EXECUTE ON DBMS_STREAMS_MESSAGING TO strmadmin;
   ```

   Explicit `EXECUTE` privilege on the package is required because a procedure in the package is called within a PL/SQL procedure in Step 7. In this case, granting the privilege through a role is not sufficient.

2. Create an ANYDATA queue in an Oracle database. This example assumes that the Streams administrator is strmadmin user.

```
CONNECT strmadmin/strmadminpw

BEGIN
  DBMS_STREAMS_ADM.SET_UP_QUEUE(
    queue_table          =>  'strm04_queue_table',
    storage_clause       =>  NULL,
    queue_name           =>  'strm04_queue');
END;
/
```

3. Create an apply process at the Oracle database to receive messages in the queue. Make sure the apply_captured parameter is set to false when you create the apply process, because the apply process will be applying user-enqueued LCRs, not LCRs captured by a capture process. Also, make sure the apply_user parameter is set to hr, because changes will be applied in to the hr.regions table, and the apply user must have privileges to make DML changes to this table.

```
BEGIN
  DBMS_APPLY_ADM.CREATE_APPLY(
     queue_name      => 'strm04_queue',
     apply_name      => 'strm04_apply',
     apply_captured  => false,
     apply_user      => 'hr');
END;
/
```

4. Create a positive rule set for the apply process and add a rule that applies DML changes to the hr.regions table made at the dbs1.net source database.

```
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
    table_name          =>  'hr.regions',
    streams_type        =>  'apply',
    streams_name        =>  'strm04_apply',
    queue_name          =>  'strm04_queue',
    include_dml         =>  true,
    include_ddl         =>  false,
    include_tagged_lcr  =>  false,
    source_database     =>  'dbs1.net',
    inclusion_rule      =>  true);
END;
/
```

5. Set the disable_on_error parameter for the apply process to n.

```
BEGIN
  DBMS_APPLY_ADM.SET_PARAMETER(
    apply_name  => 'strm04_apply',
    parameter   => 'disable_on_error',
    value       => 'n');
END;
/
```

6. Start the apply process.

```
EXEC DBMS_APPLY_ADM.START_APPLY('strm04_apply');
```

7. Create a procedure called `construct_row_lcr` that constructs a row LCR and enqueues it into the queue created in Step 2.

```
CREATE OR REPLACE PROCEDURE construct_row_lcr(
                 source_dbname  VARCHAR2,
                 cmd_type       VARCHAR2,
                 obj_owner      VARCHAR2,
                 obj_name       VARCHAR2,
                 old_vals       SYS.LCR$_ROW_LIST,
                 new_vals       SYS.LCR$_ROW_LIST) AS
  row_lcr        SYS.LCR$_ROW_RECORD;
BEGIN
  -- Construct the LCR based on information passed to procedure
  row_lcr := SYS.LCR$_ROW_RECORD.CONSTRUCT(
    source_database_name  =>  source_dbname,
    command_type          =>  cmd_type,
    object_owner          =>  obj_owner,
    object_name           =>  obj_name,
    old_values            =>  old_vals,
    new_values            =>  new_vals);
  -- Enqueue the created row LCR
  DBMS_STREAMS_MESSAGING.ENQUEUE(
    queue_name         =>  'strm04_queue',
    payload            =>  ANYDATA.ConvertObject(row_lcr));
END construct_row_lcr;
/
```

Note: The application does not need to specify a transaction identifier or SCN when it creates an LCR because the apply process generates these values and stores them in memory. If a transaction identifier or SCN is specified in the LCR, then the apply process ignores it and assigns a new value.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about LCR constructors

8. Create and enqueue LCRs using the `construct_row_lcr` procedure created in Step 3.

   a. Create a row LCR that inserts a row into the `hr.regions` table.

```
CONNECT strmadmin/strmadminpw

DECLARE
  newunit1  SYS.LCR$_ROW_UNIT;
  newunit2  SYS.LCR$_ROW_UNIT;
  newvals   SYS.LCR$_ROW_LIST;
BEGIN
  newunit1 := SYS.LCR$_ROW_UNIT(
    'region_id',
    ANYDATA.ConvertNumber(5),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
```

```
  newunit2 := SYS.LCR$_ROW_UNIT(
    'region_name',
    ANYDATA.ConvertVarchar2('Moon'),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  newvals := SYS.LCR$_ROW_LIST(newunit1,newunit2);
construct_row_lcr(
  source_dbname  =>  'dbs1.net',
  cmd_type       =>  'INSERT',
  obj_owner      =>  'hr',
  obj_name       =>  'regions',
  old_vals       =>  NULL,
  new_vals       =>  newvals);
END;
/
COMMIT;
```

b. Connect as the `hr` user and query the `hr.regions` table to view the applied row change. The row with a `region_id` of `5` should have `Moon` for the `region_name`.

```
CONNECT hr/hr

SELECT * FROM hr.regions;
```

c. Create a row LCR that updates a row in the `hr.regions` table.

```
CONNECT strmadmin/strmadminpw

DECLARE
  oldunit1  SYS.LCR$_ROW_UNIT;
  oldunit2  SYS.LCR$_ROW_UNIT;
  oldvals   SYS.LCR$_ROW_LIST;
  newunit1  SYS.LCR$_ROW_UNIT;
  newvals   SYS.LCR$_ROW_LIST;
BEGIN
  oldunit1 := SYS.LCR$_ROW_UNIT(
    'region_id',
    ANYDATA.ConvertNumber(5),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldunit2 := SYS.LCR$_ROW_UNIT(
    'region_name',
    ANYDATA.ConvertVarchar2('Moon'),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldvals := SYS.LCR$_ROW_LIST(oldunit1,oldunit2);
  newunit1 := SYS.LCR$_ROW_UNIT(
    'region_name',
    ANYDATA.ConvertVarchar2('Mars'),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  newvals := SYS.LCR$_ROW_LIST(newunit1);
```

```
construct_row_lcr(
  source_dbname  =>  'dbs1.net',
  cmd_type       =>  'UPDATE',
  obj_owner      =>  'hr',
  obj_name       =>  'regions',
  old_vals       =>  oldvals,
  new_vals       =>  newvals);
END;
/
COMMIT;
```

d. Connect as the `hr` user and query the `hr.regions` table to view the applied row change. The row with a `region_id` of `5` should have `Mars` for the `region_name`.

```
CONNECT hr/hr

SELECT * FROM hr.regions;
```

e. Create a row LCR that deletes a row from the `hr.regions` table.

```
CONNECT strmadmin/strmadminpw

DECLARE
  oldunit1  SYS.LCR$_ROW_UNIT;
  oldunit2  SYS.LCR$_ROW_UNIT;
  oldvals   SYS.LCR$_ROW_LIST;
BEGIN
  oldunit1 := SYS.LCR$_ROW_UNIT(
    'region_id',
    ANYDATA.ConvertNumber(5),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldunit2 := SYS.LCR$_ROW_UNIT(
    'region_name',
    ANYDATA.ConvertVarchar2('Mars'),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldvals := SYS.LCR$_ROW_LIST(oldunit1,oldunit2);
construct_row_lcr(
  source_dbname  =>  'dbs1.net',
  cmd_type       =>  'DELETE',
  obj_owner      =>  'hr',
  obj_name       =>  'regions',
  old_vals       =>  oldvals,
  new_vals       =>  NULL);
END;
/
COMMIT;
```

f. Connect as the `hr` user and query the `hr.regions` table to view the applied row change. The row with a `region_id` of `5` should have been deleted.

```
CONNECT hr/hr

SELECT * FROM hr.regions;
```

Executing LCRs

There are separate EXECUTE member procedures for row LCRs and DDL LCRs. These member procedures execute an LCR under the security domain of the current user. When an LCR is executed successfully, the change recorded in the LCR is made to the local database. The following sections describe executing row LCRs and DDL LCRs:

- Executing Row LCRs
- Executing DDL LCRs

Executing Row LCRs

The EXECUTE member procedure for row LCRs is a subprogram of the LCR$_ROW_RECORD type. When the EXECUTE member procedure is run on a row LCR, the row LCR is executed. If the row LCR is executed by an apply process, then any apply process handlers that would be run for the LCR are not run.

The EXECUTE member procedure can be run on a row LCR under any of the following conditions:

- The LCR is being processed by an apply handler.
- The LCR is in a queue and was last enqueued by an apply process, an application, or a user.
- The LCR has been constructed using the LCR$_ROW_RECORD constructor function but has not been enqueued.
- The LCR is in the error queue.

When you run the EXECUTE member procedure on a row LCR, the conflict_resolution parameter controls whether conflict resolution is performed. Specifically, if the conflict_resolution parameter is set to true, then any conflict resolution defined for the table being changed is used to resolve conflicts resulting from the execution of the LCR. If the conflict_resolution parameter is set to false, then conflict resolution is not used. If the conflict_resolution parameter is not set or is set to NULL, then an error is raised.

Note: A custom rule-based transformation should not run the EXECUTE member procedure on a row LCR. Doing so could execute the row LCR outside of its transactional context.

See Also:

- "Apply Processing Options for LCRs" on page 1-12
- "Managing a DML Handler" on page 9-12 for an example of a DML handler that runs the EXECUTE member procedure on row LCRs
- *Oracle Database PL/SQL Packages and Types Reference* for more information about row LCRs and the LCR$_ROW_RECORD type

Example of Constructing and Executing Row LCRs

The example in this section creates PL/SQL procedures to insert, update, and delete rows in the `hr.jobs` table by constructing and executing row LCRs. The row LCRs are executed without being enqueued into a queue or processed by an apply process. This example assumes that you have configured a Streams administrator named `strmadmin` and granted this administrator `DBA` role.

Complete the following steps:

1. Create a PL/SQL procedure named `execute_row_lcr` that executes a row LCR:

```
CONNECT strmadmin/strmadminpw

CREATE OR REPLACE PROCEDURE execute_row_lcr(
                 source_dbname  VARCHAR2,
                 cmd_type       VARCHAR2,
                 obj_owner      VARCHAR2,
                 obj_name       VARCHAR2,
                 old_vals       SYS.LCR$_ROW_LIST,
                 new_vals       SYS.LCR$_ROW_LIST) as
  xrow_lcr  SYS.LCR$_ROW_RECORD;
BEGIN
  -- Construct the row LCR based on information passed to procedure
  xrow_lcr := SYS.LCR$_ROW_RECORD.CONSTRUCT(
    source_database_name => source_dbname,
    command_type         => cmd_type,
    object_owner         => obj_owner,
    object_name          => obj_name,
    old_values           => old_vals,
    new_values           => new_vals);
  -- Execute the row LCR
  xrow_lcr.EXECUTE(FALSE);
END execute_row_lcr;
/
```

2. Create a PL/SQL procedure named `insert_job_lcr` that executes a row LCR that inserts a row into the `hr.jobs` table:

```
CREATE OR REPLACE PROCEDURE insert_job_lcr(
                 j_id     VARCHAR2,
                 j_title  VARCHAR2,
                 min_sal  NUMBER,
                 max_sal  NUMBER) AS
  xrow_lcr   SYS.LCR$_ROW_RECORD;
  col1_unit  SYS.LCR$_ROW_UNIT;
  col2_unit  SYS.LCR$_ROW_UNIT;
  col3_unit  SYS.LCR$_ROW_UNIT;
  col4_unit  SYS.LCR$_ROW_UNIT;
  newvals    SYS.LCR$_ROW_LIST;
BEGIN
  col1_unit := SYS.LCR$_ROW_UNIT(
    'job_id',
    ANYDATA.ConvertVarchar2(j_id),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
```

```
  col2_unit := SYS.LCR$_ROW_UNIT(
    'job_title',
    ANYDATA.ConvertVarchar2(j_title),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  col3_unit := SYS.LCR$_ROW_UNIT(
    'min_salary',
    ANYDATA.ConvertNumber(min_sal),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  col4_unit := SYS.LCR$_ROW_UNIT(
    'max_salary',
    ANYDATA.ConvertNumber(max_sal),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  newvals := SYS.LCR$_ROW_LIST(col1_unit,col2_unit,col3_unit,col4_unit);
  -- Execute the row LCR
  execute_row_lcr(
    source_dbname => 'DB1.NET',
    cmd_type      => 'INSERT',
    obj_owner     => 'HR',
    obj_name      => 'JOBS',
    old_vals      => NULL,
    new_vals      => newvals);
END insert_job_lcr;
/
```

3. Create a PL/SQL procedure named `update_max_salary_lcr` that executes a row LCR that updates the `max_salary` value for a row in the `hr.jobs` table:

```
CREATE OR REPLACE PROCEDURE update_max_salary_lcr(
                 j_id         VARCHAR2,
                 old_max_sal NUMBER,
                 new_max_sal NUMBER) AS
  xrow_lcr      SYS.LCR$_ROW_RECORD;
  oldcol1_unit  SYS.LCR$_ROW_UNIT;
  oldcol2_unit  SYS.LCR$_ROW_UNIT;
  newcol1_unit  SYS.LCR$_ROW_UNIT;
  oldvals       SYS.LCR$_ROW_LIST;
  newvals       SYS.LCR$_ROW_LIST;
BEGIN
  oldcol1_unit := SYS.LCR$_ROW_UNIT(
    'job_id',
    ANYDATA.ConvertVarchar2(j_id),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldcol2_unit := SYS.LCR$_ROW_UNIT(
    'max_salary',
    ANYDATA.ConvertNumber(old_max_sal),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldvals := SYS.LCR$_ROW_LIST(oldcol1_unit,oldcol2_unit);
```

```
  newcol1_unit := SYS.LCR$_ROW_UNIT(
    'max_salary',
    ANYDATA.ConvertNumber(new_max_sal),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  newvals := SYS.LCR$_ROW_LIST(newcol1_unit);
  -- Execute the row LCR
  execute_row_lcr(
    source_dbname => 'DB1.NET',
    cmd_type      => 'UPDATE',
    obj_owner     => 'HR',
    obj_name      => 'JOBS',
    old_vals      => oldvals,
    new_vals      => newvals);
END update_max_salary_lcr;
/
```

4. Create a PL/SQL procedure named `delete_job_lcr` that executes a row LCR that deletes a row from the `hr.jobs` table:

```
CREATE OR REPLACE PROCEDURE delete_job_lcr(j_id VARCHAR2) AS
  xrow_lcr   SYS.LCR$_ROW_RECORD;
  col1_unit  SYS.LCR$_ROW_UNIT;
  oldvals    SYS.LCR$_ROW_LIST;
BEGIN
  col1_unit := SYS.LCR$_ROW_UNIT(
    'job_id',
    ANYDATA.ConvertVarchar2(j_id),
    DBMS_LCR.NOT_A_LOB,
    NULL,
    NULL);
  oldvals := SYS.LCR$_ROW_LIST(col1_unit);
  -- Execute the row LCR
  execute_row_lcr(
    source_dbname => 'DB1.NET',
    cmd_type      => 'DELETE',
    obj_owner     => 'HR',
    obj_name      => 'JOBS',
    old_vals      => oldvals,
    new_vals      => NULL);
END delete_job_lcr;
/
```

5. Insert a row into the `hr.jobs` table using the `insert_job_lcr` procedure:

```
EXEC insert_job_lcr('BN_CNTR','BEAN COUNTER',5000,10000);
```

6. Select the inserted row in the `hr.jobs` table:

```
SELECT * FROM hr.jobs WHERE job_id = 'BN_CNTR';

JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
BN_CNTR    BEAN COUNTER                             5000      10000
```

7. Update the `max_salary` value for the row inserted into the `hr.jobs` table in Step 5 using the `update_max_salary_lcr` procedure:

```
EXEC update_max_salary_lcr('BN_CNTR',10000,12000);
```

8. Select the updated row in the `hr.jobs` table:

```
SELECT * FROM hr.jobs WHERE job_id = 'BN_CNTR';

JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
BN_CNTR    BEAN COUNTER                             5000      12000
```

9. Delete the row inserted into the `hr.jobs` table in Step 5 using the `delete_job_lcr` procedure:

```
EXEC delete_job_lcr('BN_CNTR');
```

10. Select the deleted row in the `hr.jobs` table:

```
SELECT * FROM hr.jobs WHERE job_id = 'BN_CNTR';

no rows selected
```

Executing DDL LCRs

The `EXECUTE` member procedure for DDL LCRs is a subprogram of the `LCR$_DDL_RECORD` type. When the `EXECUTE` member procedure is run on a DDL LCR, the LCR is executed, and any apply process handlers that would be run for the LCR are not run. The `EXECUTE` member procedure for DDL LCRs can be invoked only in an apply handler for an apply process.

All applied DDL LCRs commit automatically. Therefore, if a DDL handler calls the `EXECUTE` member procedure of a DDL LCR, then a commit is performed automatically.

See Also:

- "Apply Processing Options for LCRs" on page 1-12
- "Managing a DDL Handler" on page 9-16 for an example of a DDL handler that runs the `EXECUTE` member procedure on DDL LCRs
- *Oracle Database PL/SQL Packages and Types Reference* for more information about DDL LCRs and the `LCR$_DDL_RECORD` type

Managing LCRs Containing LOB Columns

LOB datatypes can be present in row LCRs captured by a capture process, but these datatypes are represented by other datatypes. Certain LOB datatypes cannot be present in user-constructed LCRs. Table 11–1 shows the LCR representation for these datatypes and whether these datatypes can be present in a user-constructed LCR.

Table 11–1 LOB Datatype Representations in Row LCRs

Datatype	Row LCR Representation	Can Be Present in a Captured LCR?	Can Be Present in a User-Constructed LCR?
Fixed-width `CLOB`	`VARCHAR2`	Yes	Yes
Variable-width `CLOB`	`RAW` in AL16UTF16 character set	Yes	No
`NCLOB`	`RAW` in AL16UTF16 character set	Yes	No
`BLOB`	`RAW`	Yes	Yes

The following are general considerations for row changes involving LOB datatypes in a Streams environment:

- A row change involving a LOB column can be captured, propagated, and applied as several row LCRs.
- Rules used to evaluate these row LCRs must be deterministic, so that either all of the row LCRs corresponding to the row change cause a rule in a rule set to evaluate to `TRUE`, or none of them do.

The following sections contain information about the requirements you must meet when constructing or processing LOB columns, about apply process behavior for LCRs containing LOB columns, and about LOB assembly. There is also an example that constructs and enqueues LCRs containing LOB columns.

> **See Also:** *Oracle Database Application Developer's Guide - Large Objects* for more information about LOBs

Apply Process Behavior for Direct Apply of LCRs Containing LOBs

An apply process behaves in the following ways when it applies an LCR that contains a LOB column directly (without the use of an apply handler):

- If an LCR whose command type is `INSERT` or `UPDATE` has a new LOB that contains data, and the `lob_information` is not `DBMS_LCR.LOB_CHUNK` or `DBMS_LCR.LAST_LOB_CHUNK`, then the data is applied.
- If an LCR whose command type is `INSERT` or `UPDATE` has a new LOB that contains no data, and the `lob_information` is `DBMS_LCR.EMPTY_LOB`, then it is applied as an empty LOB.
- If an LCR whose command type is `INSERT` or `UPDATE` has a new LOB that contains no data, and the `lob_information` is `DBMS_LCR.NULL_LOB` or `DBMS_LCR.INLINE_LOB`, then it is applied as a `NULL`.

- If an LCR whose command type is INSERT or UPDATE has a new LOB and the lob_information is DBMS_LCR.LOB_CHUNK or DBMS_LCR.LAST_LOB_CHUNK, then any LOB value is ignored. If the command type is INSERT, then an empty LOB is inserted into the column under the assumption that LOB chunks will follow. If the command type is UPDATE, then the column value is ignored under the assumption that LOB chunks will follow.
- If all of the new columns in an LCR whose command type is UPDATE are LOBs whose lob_information is DBMS_LCR.LOB_CHUNK or DBMS_LCR.LAST_LOB_CHUNK, then the update is skipped under the assumption that LOB chunks will follow.
- For any LCR whose command type is UPDATE or DELETE, old LOB values are ignored.

LOB Assembly and Custom Apply of LCRs Containing LOB Columns

A change to a row in a table that does not include any LOB columns results in a single row LCR, but a change to a row that includes one or more LOB columns can result in multiple row LCRs. An apply process that does not send row LCRs that contain LOB columns to an apply handler can apply these row LCRs directly. However, prior to Oracle Database 10*g* Release 2, custom processing of row LCRs that contain LOB columns was complicated because apply handlers had to be configured to process multiple LCRs correctly for a single row change.

In Oracle Database 10*g* Release 2, **LOB assembly** simplifies custom processing of captured row LCRs with LOB columns. LOB assembly automatically combines multiple captured row LCRs resulting from a change to a row with LOB columns into one row LCR. An apply process passes this single row LCR to a DML handler or error handler when LOB assembly is enabled. Also, after LOB assembly, the LOB column values are represented by LOB locators, not by VARCHAR2 or RAW datatype values. To enable LOB assembly for a DML or error handler, set the assemble_lobs parameter to true in the DBMS_APPLY_ADM.SET_DML_HANDLER procedure.

If the assemble_lobs parameter is set to false for a DML or error handler, then LOB assembly is disabled and multiple row LCRs are passed to the handler for a change to a single row with LOB columns. Table 11–2 shows Streams behavior when LOB assembly is disabled. Specifically, the table shows the LCRs passed to a DML handler or error handler resulting from a change to a single row with LOB columns.

Table 11–2 Streams Behavior with LOB Assembly Disabled

Original Row Change	First Set of LCRs	Second Set of LCRs	Third Set of LCRs	Final LCR
INSERT	One INSERT LCR	One or more LOB WRITE LCRs	One or more LOB TRIM LCRs	UPATE
UPDATE	One UPDATE LCR	One or more LOB WRITE LCRs	One or more LOB TRIM LCRs	UPATE
DELETE	One DELETE LCR	N/A	N/A	N/A
DBMS_LOB.WRITE	One or more LOB WRITE LCRs	N/A	N/A	N/A
DBMS_LOB.TRIM	One LOB TRIM LCR	N/A	N/A	N/A
DBMS_LOB.ERASE	One LOB ERASE LCR	N/A	N/A	N/A

Table 11–3 shows Streams behavior when LOB assembly is enabled. Specifically, the table shows the row LCR passed to a DML handler or error handler resulting from a change to a single row with LOB columns.

Table 11–3 Streams Behavior with LOB Assembly Enabled

Original Row Change	Single LCR
INSERT	INSERT
UPDATE	UPDATE
DELETE	DELETE
DBMS_LOB.WRITE	LOB WRITE
DBMS_LOB.TRIM	LOB TRIM
DBMS_LOB.ERASE	LOB ERASE

When LOB assembly is enabled, a DML or error handler can modify LOB columns in a row LCR. Within the PL/SQL procedure specified as a DML or error handler, the preferred way to perform operations on a LOB is to use a subprogram in the DBMS_LOB package. If a row LCR contains a LOB column that is NULL, then a new LOB locator must replace the NULL. If a row LCR will be applied with the EXECUTE member procedure, then use the ADD_COLUMN, SET_VALUE, and SET_VALUES member procedures for row LCRs to make changes to a LOB.

When LOB assembly is enabled, LOB assembly converts non-NULL LOB columns in user-enqueued LCRs into LOB locators. However, LOB assembly does not combine multiple user-enqueued row LCRs into a single row LCR. For example, for user-enqueued row LCRs, LOB assembly does not combine multiple LOB WRITE row LCRs following an INSERT row LCR into a single INSERT row LCR.

See Also:

- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers
- *Oracle Database Application Developer's Guide - Large Objects* and *Oracle Database PL/SQL Packages and Types Reference* for more information about using the DBMS_LOB package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the ADD_COLUMN, SET_VALUE, and SET_VALUES member procedures for row LCRs

LOB Assembly Considerations

The following are issues to consider when you use LOB assembly:

- To use a DML or error handler to process assembled LOBs at multiple destination databases, LOB assembly must assemble the LOBs separately on each destination database.
- Row LCRs captured on a database running a release of Oracle prior to Oracle Database 10*g* Release 2 cannot be assembled by LOB assembly.
- Row LCRs captured on a database running Oracle Database 10*g* Release 2 with a compatibility level lower than 10.2.0 cannot be assembled by LOB assembly.
- The compatibility level of the database running an apply handler must be 10.2.0 or higher to specify LOB assembly for the apply handler.

- Row LCRs from a table containing any LONG or LONG RAW columns cannot be assembled by LOB assembly.
- The SET_ENQUEUE_DESTINATION and the SET_EXECUTE procedures in the DBMS_APPLY_ADM package always operate on original, nonassembled row LCRs. Therefore, for row LCRs that contain LOB columns, the original, nonassembled row LCRs are enqueued or executed, even if these row LCRs are assembled separately for an apply handler at the destination database.
- If rule-based transformations were performed on row LCRs that contain LOB columns during capture, propagation, or apply, then an apply handler operates on the transformed row LCRs. If there are LONG or LONG RAW columns at a source database, and a rule-based transformation uses the CONVERT_LONG_TO_LOB_CHUNK member function for row LCRs to convert them to LOBs, then LOB assembly can be enabled for apply handlers that operate on these row LCRs.

See Also:

- *Oracle Database Reference* and *Oracle Database Upgrade Guide* for more information database compatibility
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the subprograms in the DBMS_APPLY_ADM package

LOB Assembly Example

This section contains an example that uses LOB assembly with a DML handler. The example scenario involves a company that shares the oe.production_information table at several databases, but only some of these databases are used for the company's online World Wide Web catalog. The company wants to store a photograph of each product in the catalog databases, but, to save space, it does not want to store these photographs at the non catalog databases.

To accomplish this goal, a DML handler at a catalog destination database can add a column named photo of datatype BLOB to each INSERT and UPDATE made to the product_information table at a source database. The source database does not include the photo column in the table. The DML handler is configured to use an existing photograph at the destination for updates and inserts.

The company also wants to add a product_long_desc to the oe.product_information table at all databases. This table already has a product_description column that contains short descriptions. The product_long_desc column is of CLOB datatype and contains detailed descriptions. The detailed descriptions are in English, but one of the company databases is used to display the company catalog in Spanish. Therefore, the DML handler updates the product_long_desc column so that the long description is in the correct language.

The following steps configure a DML handler that uses LOB assembly to accomplish the goals described previously:

Step 1 Add the photo Column to the product_information Table

The following statement adds the photo column to the product_information table at the destination database:

```
ALTER TABLE oe.product_information ADD(photo BLOB);
```

Step 2 Add the product_long_desc Column to the product_information Table

The following statement adds the `product_long_desc` column to the `product_information` table at all of the databases in the environment:

```
ALTER TABLE oe.product_information ADD(product_long_desc CLOB);
```

Step 3 Create the PL/SQL Procedure for the DML Handler

This example creates the `convert_product_information` procedure. This procedure will be used for the DML handler. This procedure assumes that the following user-created PL/SQL subprograms exist:

- The `get_photo` procedure obtains a photo in `BLOB` format from a URL or table based on the `product_id` and updates the `BLOB` locator that has been passed in as an argument.
- The `get_product_long_desc` procedure has an `IN` argument of `product_id` and an `IN OUT` argument of `product_long_desc` and translates the `product_long_desc` into Spanish or obtains the Spanish replacement description and updates `product_long_desc`.

The following code creates the `convert_product_information` procedure:

```
CREATE OR REPLACE PROCEDURE convert_product_information(in_any IN ANYDATA)
IS
  lcr                     SYS.LCR$_ROW_RECORD;
  rc                      PLS_INTEGER;
  product_id_anydata      ANYDATA;
  photo_anydata           ANYDATA;
  long_desc_anydata       ANYDATA;
  tmp_photo               BLOB;
  tmp_product_id          NUMBER;
  tmp_prod_long_desc      CLOB;
  tmp_prod_long_desc_src  CLOB;
  tmp_prod_long_desc_dest CLOB;
  t                       PLS_INTEGER;
BEGIN
  -- Access LCR
  rc := in_any.GETOBJECT(lcr);
  product_id_anydata := lcr.GET_VALUE('OLD', 'PRODUCT_ID');
  t := product_id_anydata.GETNUMBER(tmp_product_id);
  IF ((lcr.GET_COMMAND_TYPE = 'INSERT') or (lcr.GET_COMMAND_TYPE = 'UPDATE')) THEN
    -- If there is no photo column in the lcr then it must be added
    photo_anydata := lcr.GET_VALUE('NEW', 'PHOTO');
    -- Check if photo has been sent and if so whether it is NULL
    IF (photo_anydata is NULL) THEN
      tmp_photo := NULL;
      ELSE
      t := photo_anydata.GETBLOB(tmp_photo);
    END IF;
    -- If tmp_photo is NULL then a new temporary LOB must be created and
    -- updated with the photo if it exists
    IF (tmp_photo is NULL) THEN
      DBMS_LOB.CREATETEMPORARY(tmp_photo, true);
      get_photo(tmp_product_id, tmp_photo);
    END IF;
```

```
    -- If photo column did not exist then it must be added
    IF (photo_anydata is NULL) THEN
      lcr.ADD_COLUMN('NEW', 'PHOTO', ANYDATA.CONVERTBLOB(tmp_photo));
      -- Else the existing photo column must be set to the new photo
      ELSE
        lcr.SET_VALUE('NEW', 'PHOTO', ANYDATA.CONVERTBLOB(tmp_photo));
    END IF;
    long_desc_anydata := lcr.GET_VALUE('NEW', 'PRODUCT_LONG_DESC');
    IF (long_desc_anydata is NULL) THEN
      tmp_prod_long_desc_src := NULL;
      ELSE
      t := long_desc_anydata.GETCLOB(tmp_prod_long_desc_src);
    END IF;
    IF (tmp_prod_long_desc_src IS NOT NULL) THEN
      get_product_long_desc(tmp_product_id, tmp_prod_long_desc);
    END IF;
    -- If tmp_prod_long_desc IS NOT NULL, then use it to update the LCR
    IF (tmp_prod_long_desc IS NOT NULL) THEN
      lcr.SET_VALUE('NEW', 'PRODUCT_LONG_DESC',
                    ANYDATA.CONVERTCLOB(tmp_prod_long_desc_dest));
    END IF;
  END IF;
  -- DBMS_LOB operations also are executed
  -- Inserts and updates invoke all changes
  lcr.EXECUTE(true);
END;
/
```

Step 4 Set the DML Handler for the Apply Process

This step sets the `convert_product_information` procedure as the DML handler at the destination database for `INSERT`, `UPDATE`, and `LOB_UPDATE` operations. Notice that the `assemble_lobs` parameter is set to `true` each time the `SET_DML_HANDLER` procedure is run.

```
BEGIN
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name         => 'oe.product_information',
    object_type         => 'TABLE',
    operation_name      => 'INSERT',
    error_handler       => false,
    user_procedure      => 'strmadmin.convert_product_information',
    apply_database_link => NULL,
    assemble_lobs       => true);
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name         => 'oe.product_information',
    object_type         => 'TABLE',
    operation_name      => 'UPDATE',
    error_handler       => false,
    user_procedure      => 'strmadmin.convert_product_information',
    apply_database_link => NULL,
    assemble_lobs       => true);
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name         => 'oe.product_information',
    object_type         => 'TABLE',
    operation_name      => 'LOB_UPDATE',
    error_handler       => false,
    user_procedure      => 'strmadmin.convert_product_information',
    apply_database_link => NULL,
    assemble_lobs       => true);
END;
```

```
/
```

Step 5 Query the DBA_APPLY_DML_HANDLERS View

To ensure that the DML handler is set properly for the `oe.product_information` table, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Table|Owner' FORMAT A5
COLUMN OBJECT_NAME HEADING 'Table Name' FORMAT A20
COLUMN OPERATION_NAME HEADING 'Operation' FORMAT A10
COLUMN USER_PROCEDURE HEADING 'Handler Procedure' FORMAT A25
COLUMN ASSEMBLE_LOBS HEADING 'LOB Assembly?' FORMAT A15

SELECT OBJECT_OWNER,
       OBJECT_NAME,
       OPERATION_NAME,
       USER_PROCEDURE,
       ASSEMBLE_LOBS
  FROM DBA_APPLY_DML_HANDLERS;
```

Your output looks similar to the following:

```
Table
Owner Table Name           Operation  Handler Procedure         LOB Assembly?
----- -------------------- ---------- ------------------------- ---------------
OE    PRODUCT_INFORMATION  INSERT     "STRMADMIN"."CONVERT_PROD Y
                                      UCT_INFORMATION"

OE    PRODUCT_INFORMATION  UPDATE     "STRMADMIN"."CONVERT_PROD Y
                                      UCT_INFORMATION"

OE    PRODUCT_INFORMATION  LOB_UPDATE "STRMADMIN"."CONVERT_PROD Y
                                      UCT_INFORMATION"
```

Notice that the correct procedure, `convert_product_information`, is used for each operation on the table. Also, notice that each handler uses LOB assembly.

Requirements for Constructing and Processing LCRs Containing LOB Columns

If your environment produces row LCRs that contain LOB columns, then you must meet the requirements in the following sections when you construct or process these LCRs:

- Requirements for Constructing and Processing LCRs Without LOB Assembly
- Requirements for Apply Handler Processing of LCRs with LOB Assembly
- Requirements for Rule-Based Transformation Processing of LCRs with LOBs

Requirements for Constructing and Processing LCRs Without LOB Assembly

The following requirements must be met when you are constructing LCRs with LOB columns and when you are processing LOB columns with a DML or error handler that has LOB assembly disabled:

- Do not modify LOB column data in a row LCR with a DML handler or error handler that has LOB assembly disabled. However, you can modify non-LOB columns in row LCRs with a DML or error handler.
- Do not allow LCRs from a table that contains LOB columns to be processed by an apply handler that is invoked only for specific operations. For example, an apply handler that is invoked only for `INSERT` operations should not process LCRs from a table with one or more LOB columns.
- The data portion of the LCR LOB column must be of type `VARCHAR2` or `RAW`. A `VARCHAR2` is interpreted as a `CLOB`, and a `RAW` is interpreted as a `BLOB`.
- A LOB column in a user-constructed row LCR must be either a `BLOB` or a fixed-width `CLOB`. You cannot construct a row LCR with the following types of LOB columns: `NCLOB` or variable-width `CLOB`.
- `LOB WRITE`, `LOB ERASE`, and `LOB TRIM` are the only valid command types for out-of-line LOBs.
- For `LOB WRITE`, `LOB ERASE`, and `LOB TRIM` LCRs, the `old_values` collection should be empty or `NULL`, and `new_values` should not be empty.
- The `lob_offset` should be a valid value for `LOB WRITE` and `LOB ERASE` LCRs. For all other command types, `lob_offset` should be `NULL`, under the assumption that LOB chunks for that column will follow.
- The `lob_operation_size` should be a valid value for `LOB ERASE` and `LOB TRIM` LCRs. For all other command types, `lob_operation_size` should be `NULL`.
- `LOB TRIM` and `LOB ERASE` are valid command types only for an LCR containing a LOB column with `lob_information` set to `LAST_LOB_CHUNK`.
- `LOB WRITE` is a valid command type only for an LCR containing a LOB column with `lob_information` set to `LAST_LOB_CHUNK` or `LOB_CHUNK`.
- For LOBs with `lob_information` set to `NULL_LOB`, the data portion of the column should be a `NULL` of `VARCHAR2` type (for a `CLOB`) or a `NULL` of `RAW` type (for a `BLOB`). Otherwise, it is interpreted as a non-`NULL` inline LOB column.
- Only one LOB column reference with one new chunk is allowed for each `LOB WRITE`, `LOB ERASE`, and `LOB TRIM` LCR.
- The new LOB chunk for a `LOB ERASE` and a `LOB TRIM` LCR should be a `NULL` value encapsulated in an `ANYDATA`.

An apply process performs all validation of these requirements. If these requirements are not met, then a row LCR containing LOB columns cannot be applied by an apply process nor processed by an apply handler. In this case, the LCR is moved to the error queue with the rest of the LCRs in the same transaction.

See Also:

- "Constructing and Enqueuing LCRs" on page 11-2
- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers

Requirements for Apply Handler Processing of LCRs with LOB Assembly

The following requirements must be met when you are processing LOB columns with a DML or error handler that has LOB assembly enabled:

- Do not use the following row LCR member procedures on LOB columns in row LCRs that contain assembled LOBs:
  - SET_LOB_INFORMATION
  - SET_LOB_OFFSET
  - SET_LOB_OPERATION_SIZE

  An error is raised if one of these procedures is used on a LOB column in a row LCR.
- Row LCRs constructed by LOB assembly cannot be enqueued by a DML handler or error handler. However, even when LOB assembly is enabled for one or more handlers at a destination database, the original, nonassembled row LCRs with LOB columns can be enqueued using the SET_ENQUEUE_DESTINATION procedure in the DBMS_APPLY_ADM package.

An apply process performs all validation of these requirements. If these requirements are not met, then a row LCR containing LOB columns cannot be applied by an apply process nor processed by an apply handler. In this case, the LCR is moved to the error queue with the rest of the LCRs in the same transaction. For row LCRs with LOB columns, the original, nonassembled row LCRs are placed in the error queue.

See Also:

- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers
- *Oracle Database PL/SQL Packages and Types Reference* for more information about member procedures for row LCRs and for information about the SET_ENQUEUE_DESTINATION procedure

Requirements for Rule-Based Transformation Processing of LCRs with LOBs

The following requirements must be met when you are processing row LCRs that contain LOB columns with a rule-based transformation:

- Do not modify LOB column data in a row LCR with a custom rule-based transformation. However, a custom rule-based transformation can modify non-LOB columns in row LCRs that contain LOB columns.
- You cannot use the following row LCR member procedures on a LOB column when you are processing a row LCR with a custom rule-based transformation:
  - ADD_COLUMN
  - SET_LOB_INFORMATION
  - SET_LOB_OFFSET
  - SET_LOB_OPERATION_SIZE
  - SET_VALUE
  - SET_VALUES
- A declarative rule-based transformation created by the ADD_COLUMN procedure in the DBMS_STREAMS_ADM package cannot add a LOB column to a row LCR.

- Rule-based transformation functions that are run on row LCRs with LOB columns must be deterministic, so that all row LCRs corresponding to the row change are transformed in the same way.
- Do not allow LCRs from a table that contains LOB columns to be processed by an a custom rule-based transformation that is invoked only for specific operations. For example, a custom rule-based transformation that is invoked only for `INSERT` operations should not process LCRs from a table with one or more LOB columns.

Note: If row LCRs contain LOB columns, then rule-based transformations always operate on the original, nonassembled row LCRs.

See Also:

- "Constructing and Enqueuing LCRs" on page 11-2
- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers
- *Oracle Streams Concepts and Administration* for information about rule-based transformations
- *Oracle Database PL/SQL Packages and Types Reference* for more information about member procedures for row LCRs
- *Oracle Database SQL Reference* for more information about deterministic functions

Example Script for Constructing and Enqueuing LCRs Containing LOBs

The example in this section illustrates creating a PL/SQL procedure for constructing and enqueuing LCRs containing LOBs. This example assumes that you have prepared your database for Streams by completing the necessary actions described in *Oracle Streams Concepts and Administration*.

Note: The extended example is not included in the PDF version of this chapter, but it is included in the HTML version of the chapter.

Managing LCRs Containing LONG or LONG RAW Columns

`LONG` and `LONG RAW` datatypes all can be present in row LCRs captured by a capture process, but these datatypes are represented by the following datatypes in row LCRs.

- `LONG` datatype is represented as `VARCHAR2` datatype in row LCRs.
- `LONG RAW` datatype is represented as `RAW` datatype in row LCRs.

A row change involving a `LONG` or `LONG RAW` column can be captured, propagated, and applied as several LCRs. If your environment uses LCRs that contain `LONG` or `LONG RAW` columns, then the data portion of the LCR `LONG` or `LONG RAW` column must be of type `VARCHAR2` or `RAW`. A `VARCHAR2` is interpreted as a `LONG`, and a `RAW` is interpreted as a `LONG RAW`.

You must meet the following requirements when you are processing row LCRs that contain LONG or LONG RAW column data in Streams:

- Do not modify LONG or LONG RAW column data in an LCR using a custom rule-based transformation. However, you can use a rule-based transformation to modify non LONG and non LONG RAW columns in row LCRs that contain LONG or LONG RAW column data.
- Do not use the SET_VALUE or SET_VALUES row LCR member procedures in a custom rule-based transformation that is processing a row LCR that contains LONG or LONG RAW data. Doing so raises the ORA-26679 error.
- Rule-based transformation functions that are run on LCRs that contain LONG or LONG RAW columns must be deterministic, so that all LCRs corresponding to the row change are transformed in the same way.
- A declarative rule-based transformation created by the ADD_COLUMN procedure in the DBMS_STREAMS_ADM package cannot add a LONG or LONG RAW column to a row LCR.
- You cannot use a DML handler or error handler to process row LCRs that contain LONG or LONG RAW column data.
- Rules used to evaluate LCRs that contain LONG or LONG RAW columns must be deterministic, so that either all of the LCRs corresponding to the row change cause a rule in a rule set to evaluate to TRUE, or none of them do.
- You cannot use an apply process to enqueue LCRs that contain LONG or LONG RAW column data into a destination queue. The SET_DESTINATION_QUEUE procedure in the DBMS_APPLY_ADM package sets the destination queue for LCRs that satisfy a specified apply process rule.

Note: LONG and LONG RAW datatypes cannot be present in user-constructed LCRs.

See Also:

- "Apply Processing Options for LCRs" on page 1-12 for more information about apply handlers
- *Oracle Streams Concepts and Administration* for information about rule-based transformations
- *Oracle Database SQL Reference* for more information about deterministic functions

12

Monitoring Streams Replication

This chapter provides information about the static data dictionary views and dynamic performance views related to Streams replication. You can use these views to monitor your Streams replication environment. This chapter also illustrates example queries that you can use to monitor your Streams replication environment.

This chapter contains these topics:

- Monitoring Supplemental Logging
- Monitoring an Apply Process in a Streams Replication Environment
- Monitoring Streams Tags
- Monitoring Instantiation
- Running Flashback Queries in a Streams Replication Environment

---

Note:

- The Streams tool in the Oracle Enterprise Manager Console is also an excellent way to monitor a Streams environment. See the online help for the Streams tool for more information.
- To collect elapsed time statistics in the dynamic performance views discussed in this chapter, set the `TIMED_STATISTICS` initialization parameter to `true`.

See Also:

- *Oracle Streams Concepts and Administration* for more information about monitoring a Streams environment
- *Oracle Database Reference* for information about the data dictionary views described in this chapter

Monitoring Supplemental Logging

The following sections contain queries that you can run to monitor supplemental logging at a source database:

- Displaying Supplemental Log Groups at a Source Database
- Displaying Database Supplemental Logging Specifications
- Displaying Supplemental Logging Specified During Preparation for Instantiation

The total supplemental logging at a database is determined by the results shown in all three of the queries in these sections combined. For example, supplemental logging can be enabled for columns in a table even if no results for the table are returned by the query in the "Displaying Supplemental Log Groups at a Source Database" section. That is, supplemental logging can be enabled for the table if database supplemental logging is enabled or if the table is in a schema for which supplemental logging was enabled during preparation for instantiation.

Supplemental logging places additional column data into a redo log when an operation is performed. The capture process captures this additional information and places it in LCRs. An apply process that applies captured LCRs might need this additional information to schedule or apply changes correctly.

See Also:

- "Supplemental Logging for Streams Replication" on page 1-8
- "Managing Supplemental Logging in a Streams Replication Environment" on page 9-3

Displaying Supplemental Log Groups at a Source Database

To check whether one or more log groups are specified for the table at the source database, run the following query:

```
COLUMN LOG_GROUP_NAME HEADING 'Log Group' FORMAT A20
COLUMN TABLE_NAME HEADING 'Table' FORMAT A15
COLUMN ALWAYS HEADING 'Conditional or|Unconditional' FORMAT A14
COLUMN LOG_GROUP_TYPE HEADING 'Type of Log Group' FORMAT A20

SELECT
    LOG_GROUP_NAME,
    TABLE_NAME,
    DECODE(ALWAYS,
             'ALWAYS', 'Unconditional',
             'CONDITIONAL', 'Conditional') ALWAYS,
    LOG_GROUP_TYPE
  FROM DBA_LOG_GROUPS;
```

Your output looks similar to the following:

```
                                Conditional or
Log Group           Table       Unconditional  Type of Log Group
------------------- ----------- -------------- --------------------
LOG_GROUP_DEP_PK    DEPARTMENTS Unconditional  USER LOG GROUP
SYS_C002105         REGIONS     Unconditional  PRIMARY KEY LOGGING
SYS_C002106         REGIONS     Conditional    FOREIGN KEY LOGGING
SYS_C002110         LOCATIONS   Unonditional   ALL COLUMN LOGGING
SYS_C002111         COUNTRIES   Conditional    ALL COLUMN LOGGING
LOG_GROUP_JOBS_CR   JOBS        Conditional    USER LOG GROUP
```

If the output for the type of log group shows how the log group was created:

- If the output is USER LOG GROUP, then the log group was created using the ADD SUPPLEMENTAL LOG GROUP clause of the ALTER TABLE statement.
- Otherwise, the log group was created using the ADD SUPPLEMENTAL LOG DATA clause of the ALTER TABLE statement.

If the type of log group is USER LOG GROUP, then you can list the columns in the log group by querying the DBA_LOG_GROUP_COLUMNS data dictionary view.

> **Attention:** If the type of log group is not USER LOG GROUP, then the DBA_LOG_GROUP_COLUMNS data dictionary view does not contain information about the columns in the log group. Instead, Oracle supplementally logs the correct columns when an operation is performed on the table. For example, if the type of log group is PRIMARY KEY LOGGING, then Oracle logs the current primary key column(s) when a change is performed on the table.

Displaying Database Supplemental Logging Specifications

To display the database supplemental logging specifications, query the V$DATABASE dynamic performance view, as in the following example:

```
COLUMN log_min HEADING 'Minimum|Supplemental|Logging?' FORMAT A12
COLUMN log_pk HEADING 'Primary Key|Supplemental|Logging?' FORMAT A12
COLUMN log_fk HEADING 'Foreign Key|Supplemental|Logging?' FORMAT A12
COLUMN log_ui HEADING 'Unique|Supplemental|Logging?' FORMAT A12
COLUMN log_all HEADING 'All Columns|Supplemental|Logging?' FORMAT A12

SELECT SUPPLEMENTAL_LOG_DATA_MIN log_min,
       SUPPLEMENTAL_LOG_DATA_PK log_pk,
       SUPPLEMENTAL_LOG_DATA_FK log_fk,
       SUPPLEMENTAL_LOG_DATA_UI log_ui,
       SUPPLEMENTAL_LOG_DATA_ALL log_all
  FROM V$DATABASE;
```

Your output looks similar to the following:

```
Minimum      Primary Key  Foreign Key  Unique        All Columns
Supplemental Supplemental Supplemental Supplemental  Supplemental
Logging?     Logging?     Logging?     Logging?      Logging?
------------ ------------ ------------ ------------- ------------
YES          YES          YES          YES           NO
```

These results show that minimum, primary key, foreign key, and unique key columns are being supplementally logged for all of the tables in the database. Because unique key columns are supplementally logged, bitmap index columns also are supplementally logged. However, all columns are not being supplementally logged.

Displaying Supplemental Logging Specified During Preparation for Instantiation

Supplemental logging can be enabled when database objects are prepared for instantiation using one of the three procedures in the DBMS_CAPTURE_ADM package. A data dictionary view displays the supplemental logging enabled by each of these procedures: PREPARE_TABLE_INSTANTIATION, PREPARE_SCHEMA_INSTANTIATION, and PREPARE_GLOBAL_INSTANTIATION.

- The DBA_CAPTURE_PREPARED_TABLES view displays the supplemental logging enabled by the PREPARE_TABLE_INSTANTIATION procedure.
- The DBA_CAPTURE_PREPARED_SCHEMAS view displays the supplemental logging enabled by the PREPARE_SCHEMA_INSTANTIATION procedure.
- The DBA_CAPTURE_PREPARED_DATABASE view displays the supplemental logging enabled by the PREPARE_GLOBAL_INSTANTIATION procedure.

Each of these views has the following columns:

- SUPPLEMENTAL_LOG_DATA_PK shows whether primary key supplemental logging was enabled by a procedure.
- SUPPLEMENTAL_LOG_DATA_UI shows whether unique key and bitmap index supplemental logging was enabled by a procedure.
- SUPPLEMENTAL_LOG_DATA_FK shows whether foreign key supplemental logging was enabled by a procedure.
- SUPPLEMENTAL_LOG_DATA_ALL shows whether supplemental logging for all columns was enabled by a procedure.

Each of these columns can display one of the following values:

- IMPLICIT means that the relevant procedure enabled supplemental logging for the columns.
- EXPLICIT means that supplemental logging was enabled for the columns manually using an ALTER TABLE or ALTER DATABASE statement with an ADD SUPPLEMENTAL LOG DATA clause.
- NO means that supplemental logging was not enabled for the columns using a prepare procedure or an ALTER TABLE or ALTER DATABASE statement with an ADD SUPPLEMENTAL LOG DATA clause. Supplemental logging might not be enabled for the columns. However, supplemental logging might be enabled for the columns at another level (table, schema, or database), or it might have been enabled using an ALTER TABLE statement with an ADD SUPPLEMENTAL LOG GROUP clause.

The following sections contain queries that display the supplemental logging enabled by these procedures:

- Displaying Supplemental Logging Enabled by PREPARE_TABLE_INSTANTIATION
- Displaying Supplemental Logging Enabled by PREPARE_SCHEMA_INSTANTIATION
- Displaying Supplemental Logging Enabled by PREPARE_GLOBAL_INSTANTIATION

Displaying Supplemental Logging Enabled by PREPARE_TABLE_INSTANTIATION

The following query displays the supplemental logging enabled by the PREPARE_TABLE_INSTANTIATION procedure for the tables in the hr schema:

```
COLUMN TABLE_NAME HEADING 'Table Name' FORMAT A15
COLUMN log_pk HEADING 'Primary Key|Supplemental|Logging' FORMAT A12
COLUMN log_fk HEADING 'Foreign Key|Supplemental|Logging' FORMAT A12
COLUMN log_ui HEADING 'Unique|Supplemental|Logging' FORMAT A12
COLUMN log_all HEADING 'All Columns|Supplemental|Logging' FORMAT A12

SELECT TABLE_NAME,
       SUPPLEMENTAL_LOG_DATA_PK log_pk,
       SUPPLEMENTAL_LOG_DATA_FK log_fk,
       SUPPLEMENTAL_LOG_DATA_UI log_ui,
       SUPPLEMENTAL_LOG_DATA_ALL log_all
  FROM DBA_CAPTURE_PREPARED_TABLES
  WHERE TABLE_OWNER = 'HR';
```

Your output looks similar to the following:

```
                Primary Key  Foreign Key  Unique         All Columns
                Supplemental Supplemental Supplemental   Supplemental
Table Name      Logging      Logging      Logging        Logging
--------------- ------------ ------------ -------------- ------------
COUNTRIES       NO           NO           NO             NO
REGIONS         IMPLICIT     IMPLICIT     IMPLICIT       NO
DEPARTMENTS     IMPLICIT     IMPLICIT     IMPLICIT       NO
LOCATIONS       EXPLICIT     NO           NO             NO
EMPLOYEES       NO           NO           NO             IMPLICIT
JOB_HISTORY     NO           NO           NO             NO
JOBS            NO           NO           NO             NO
```

These results show the following:

- The PREPARE_TABLE_INSTANTIATION procedure enabled supplemental logging for the primary key, unique key, bitmap index, and foreign key columns in the hr.regions and hr.departments tables.
- The PREPARE_TABLE_INSTANTIATION procedure enabled supplemental logging for all columns in the hr.employees table.
- An ALTER TABLE statement with an ADD SUPPLEMENTAL LOG DATA clause enabled primary key supplemental logging for the hr.locations table.

> **Note:** Omit the WHERE clause in the query to list the information for all of the tables in the database.

Displaying Supplemental Logging Enabled by PREPARE_SCHEMA_INSTANTIATION

The following query displays the supplemental logging enabled by the PREPARE_SCHEMA_INSTANTIATION procedure:

```
COLUMN SCHEMA_NAME HEADING 'Schema Name' FORMAT A20
COLUMN log_pk HEADING 'Primary Key|Supplemental|Logging' FORMAT A12
COLUMN log_fk HEADING 'Foreign Key|Supplemental|Logging' FORMAT A12
COLUMN log_ui HEADING 'Unique|Supplemental|Logging' FORMAT A12
COLUMN log_all HEADING 'All Columns|Supplemental|Logging' FORMAT A12
```

```
SELECT SCHEMA_NAME,
       SUPPLEMENTAL_LOG_DATA_PK log_pk,
       SUPPLEMENTAL_LOG_DATA_FK log_fk,
       SUPPLEMENTAL_LOG_DATA_UI log_ui,
       SUPPLEMENTAL_LOG_DATA_ALL log_all
  FROM DBA_CAPTURE_PREPARED_SCHEMAS;
```

Your output looks similar to the following:

```
                    Primary Key  Foreign Key  Unique         All Columns
                    Supplemental Supplemental Supplemental   Supplemental
Schema Name         Logging      Logging      Logging        Logging
------------------- ------------ ------------ -------------- ------------
OUTLN               NO           NO           NO             NO
DIP                 NO           NO           NO             NO
TSMSYS              NO           NO           NO             NO
DBSNMP              NO           NO           NO             NO
WMSYS               NO           NO           NO             NO
CTXSYS              NO           NO           NO             NO
SCOTT               NO           NO           NO             NO
ADAMS               NO           NO           NO             NO
JONES               NO           NO           NO             NO
CLARK               NO           NO           NO             NO
BLAKE               NO           NO           NO             NO
HR                  NO           NO           NO             IMPLICIT
OE                  IMPLICIT     IMPLICIT     IMPLICIT       NO
IX                  NO           NO           NO             NO
ORDSYS              NO           NO           NO             NO
ORDPLUGINS          NO           NO           NO             NO
SI_INFORMTN_SCHEMA  NO           NO           NO             NO
MDSYS               NO           NO           NO             NO
PM                  NO           NO           NO             NO
SH                  NO           NO           NO             NO
```

These results show the following:

- The `PREPARE_SCHEMA_INSTANTIATION` procedure enabled supplemental logging for all columns in tables in the `hr` schema.
- The `PREPARE_SCHEMA_INSTANTIATION` procedure enabled supplemental logging for the primary key, unique key, bitmap index, and foreign key columns in the tables in the `oe` schema.

Displaying Supplemental Logging Enabled by PREPARE_GLOBAL_INSTANTIATION

The following query displays the supplemental logging enabled by the `PREPARE_GLOBAL_INSTANTIATION` procedure:

```
COLUMN log_pk HEADING 'Primary Key|Supplemental|Logging' FORMAT A12
COLUMN log_fk HEADING 'Foreign Key|Supplemental|Logging' FORMAT A12
COLUMN log_ui HEADING 'Unique|Supplemental|Logging' FORMAT A12
COLUMN log_all HEADING 'All Columns|Supplemental|Logging' FORMAT A12

SELECT SUPPLEMENTAL_LOG_DATA_PK log_pk,
       SUPPLEMENTAL_LOG_DATA_FK log_fk,
       SUPPLEMENTAL_LOG_DATA_UI log_ui,
       SUPPLEMENTAL_LOG_DATA_ALL log_all
  FROM DBA_CAPTURE_PREPARED_DATABASE;
```

Your output looks similar to the following:

```
Primary Key  Foreign Key  Unique         All Columns
Supplemental Supplemental Supplemental   Supplemental
Logging      Logging      Logging        Logging
------------ ------------ -------------- ------------
IMPLICIT     IMPLICIT     IMPLICIT       NO
```

These results show that the `PREPARE_GLOBAL_INSTANTIATION` procedure enabled supplemental logging for the primary key, unique key, bitmap index, and foreign key columns in all of the tables in the database.

Monitoring an Apply Process in a Streams Replication Environment

The following sections contain queries that you can run to monitor an apply process in a Stream replication environment:

- Displaying the Substitute Key Columns Specified at a Destination Database
- Displaying Information About DML and DDL Handlers
- Monitoring Virtual Dependency Definitions
- Displaying Information About Conflict Detection
- Displaying Information About Update Conflict Handlers

See Also:

- "Apply and Streams Replication" on page 1-12
- "Managing Apply for Streams Replication" on page 9-9

Displaying the Substitute Key Columns Specified at a Destination Database

You can designate a substitute key at a destination database, which is a column or set of columns that Oracle can use to identify rows in the table during apply. Substitute key columns can be used to specify key columns for a table that has no primary key, or they can be used instead of a table's primary key when the table is processed by any apply process at a destination database.

To display all of the substitute key columns specified at a destination database, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Table Owner' FORMAT A20
COLUMN OBJECT_NAME HEADING 'Table Name' FORMAT A20
COLUMN COLUMN_NAME HEADING 'Substitute Key Name' FORMAT A20
COLUMN APPLY_DATABASE_LINK HEADING 'Database Link|for Remote|Apply' FORMAT A15

SELECT OBJECT_OWNER, OBJECT_NAME, COLUMN_NAME, APPLY_DATABASE_LINK
  FROM DBA_APPLY_KEY_COLUMNS
  ORDER BY APPLY_DATABASE_LINK, OBJECT_OWNER, OBJECT_NAME;
```

Your output looks similar to the following:

```
                                                          Database Link
                                                          for Remote
Table Owner          Table Name          Substitute Key Name  Apply
-------------------- ------------------- -------------------- ---------------
HR                   DEPARTMENTS         DEPARTMENT_NAME
HR                   DEPARTMENTS         LOCATION_ID
HR                   EMPLOYEES           FIRST_NAME
HR                   EMPLOYEES           LAST_NAME
HR                   EMPLOYEES           HIRE_DATE
```

Note: This query shows the database link in the last column if the substitute key columns are for a remote non-Oracle database. The last column is NULL if a substitute key column is specified for the local destination database.

See Also:

- "Substitute Key Columns" on page 1-19
- "Managing the Substitute Key Columns for a Table" on page 9-11
- *Oracle Streams Concepts and Administration* for information about managing apply errors

Displaying Information About DML and DDL Handlers

This section contains queries that display information about apply process DML handlers and DDL handlers.

See Also: *Oracle Streams Concepts and Administration* for more information about DML and DDL handlers

Displaying All of the DML Handlers for Local Apply

When you specify a local DML handler using the SET_DML_HANDLER procedure in the DBMS_APPLY_ADM package at a destination database, you can either specify that the handler runs for a specific apply process or that the handler is a general handler that runs for all apply processes in the database that apply changes locally, when appropriate. A specific DML handler takes precedence over a generic DML handler. A DML is run for a specified operation on a specific table.

To display the DML handler for each apply process that applies changes locally in a database, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Table|Owner' FORMAT A11
COLUMN OBJECT_NAME HEADING 'Table Name' FORMAT A10
COLUMN OPERATION_NAME HEADING 'Operation' FORMAT A9
COLUMN USER_PROCEDURE HEADING 'Handler Procedure' FORMAT A25
COLUMN APPLY_NAME HEADING 'Apply Process|Name' FORMAT A15

SELECT OBJECT_OWNER,
       OBJECT_NAME,
       OPERATION_NAME,
       USER_PROCEDURE,
       APPLY_NAME
  FROM DBA_APPLY_DML_HANDLERS
  WHERE ERROR_HANDLER = 'N' AND
        APPLY_DATABASE_LINK IS NULL
  ORDER BY OBJECT_OWNER, OBJECT_NAME;
```

Your output looks similar to the following:

```
Table                                                        Apply Process
Owner       Table Name Operation Handler Procedure           Name
----------- ---------- --------- ------------------------- --------------
HR          LOCATIONS  UPDATE    "STRMADMIN"."HISTORY_DML"
```

Because Apply Process Name is NULL for the strmadmin.history_dml DML handler, this handler is a general handler that runs for all of the local apply processes.

Note: You can also specify DML handlers to process changes for remote non-Oracle databases. This query does not display such DML handlers because it lists a DML handler only if the APPLY_DATABASE_LINK column is NULL.

See Also: "Managing a DML Handler" on page 9-12

Displaying the DDL Handler for Each Apply Process

To display the DDL handler for each apply process in a database, run the following query:

```
COLUMN APPLY_NAME HEADING 'Apply Process Name' FORMAT A20
COLUMN DDL_HANDLER HEADING 'DDL Handler' FORMAT A40

SELECT APPLY_NAME, DDL_HANDLER FROM DBA_APPLY;
```

Your output looks similar to the following:

```
Apply Process Name   DDL Handler
-------------------- ----------------------------------------
STREP01_APPLY        "STRMADMIN"."HISTORY_DDL"
```

See Also: "Managing a DDL Handler" on page 9-16

Monitoring Virtual Dependency Definitions

The following sections contain queries that display information about virtual dependency definitions in a database:

- Displaying Value Dependencies
- Displaying Object Dependencies

See Also: "Apply Processes and Dependencies" on page 1-14 for more information about virtual dependency definitions

Displaying Value Dependencies

To display the value dependencies in a database, run the following query:

```
COLUMN DEPENDENCY_NAME HEADING 'Dependency Name' FORMAT A25
COLUMN OBJECT_OWNER HEADING 'Object Owner' FORMAT A15
COLUMN OBJECT_NAME HEADING 'Object Name' FORMAT A20
COLUMN COLUMN_NAME HEADING 'Column Name' FORMAT A15

SELECT DEPENDENCY_NAME,
       OBJECT_OWNER,
       OBJECT_NAME,
       COLUMN_NAME
  FROM DBA_APPLY_VALUE_DEPENDENCIES;
```

Your output should look similar to the following:

```
Dependency Name           Object Owner    Object Name          Column Name
------------------------- --------------- -------------------- ---------------
ORDER_ID_FOREIGN_KEY      OE              ORDERS               ORDER_ID
ORDER_ID_FOREIGN_KEY      OE              ORDER_ITEMS          ORDER_ID
KEY_53_FOREIGN_KEY        US_DESIGNS      ALL_DESIGNS_SUMMARY  KEY_53
KEY_53_FOREIGN_KEY        US_DESIGNS      DESIGN_53            KEY_53
```

This output shows the following value dependencies:

- The `order_id_foreign_key` value dependency describes a dependency between the `order_id` column in the `oe.orders` table and the `order_id` column in the `oe.order_items` table.
- The `key_53_foreign_key` value dependency describes a dependency between the `key_53` column in the `us_designs.all_designs_summary` table and the `key_53` column in the `us_designs.design_53` table.

Displaying Object Dependencies

To display the object dependencies in a database, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Object Owner' FORMAT A15
COLUMN OBJECT_NAME HEADING 'Object Name' FORMAT A15
COLUMN PARENT_OBJECT_OWNER HEADING 'Parent Object Owner' FORMAT A20
COLUMN PARENT_OBJECT_NAME HEADING 'Parent Object Name' FORMAT A20

SELECT OBJECT_OWNER,
       OBJECT_NAME,
       PARENT_OBJECT_OWNER,
       PARENT_OBJECT_NAME
  FROM DBA_APPLY_OBJECT_DEPENDENCIES;
```

Your output should look similar to the following:

```
Object Owner    Object Name     Parent Object Owner  Parent Object Name
--------------- --------------- -------------------- --------------------
ORD             CUSTOMERS       ORD                  SHIP_ORDERS
ORD             ORDERS          ORD                  SHIP_ORDERS
ORD             ORDER_ITEMS     ORD                  SHIP_ORDERS
```

This output shows an object dependency in which the `ord.ship_orders` table is a parent table to the following child tables:

- `ord.customers`
- `ord.orders`
- `ord.order_items`

Displaying Information About Conflict Detection

You can stop conflict detection for nonkey columns using the COMPARE_OLD_VALUES procedure in the DBMS_APPLY_ADM package. When you use this procedure, conflict detection is stopped for the specified columns for all apply processes at a destination database. To display each column for which conflict detection has been stopped, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Table Owner' FORMAT A15
COLUMN OBJECT_NAME HEADING 'Table Name' FORMAT A20
COLUMN COLUMN_NAME HEADING 'Column Name' FORMAT A20
COLUMN COMPARE_OLD_ON_DELETE HEADING 'Compare|Old On|Delete' FORMAT A7
COLUMN COMPARE_OLD_ON_UPDATE HEADING 'Compare|Old On|Update' FORMAT A7

SELECT OBJECT_OWNER,
       OBJECT_NAME,
       COLUMN_NAME,
       COMPARE_OLD_ON_DELETE,
       COMPARE_OLD_ON_UPDATE
  FROM DBA_APPLY_TABLE_COLUMNS
  WHERE APPLY_DATABASE_LINK IS NULL;
```

Your output should look similar to the following:

```
                                                    Compare Compare
                                                    Old On  Old On
Table Owner     Table Name          Column Name     Delete  Update
--------------- ------------------- ------------------- ------- -------
HR              EMPLOYEES           COMMISSION_PCT      NO      NO
HR              EMPLOYEES           EMAIL               NO      NO
HR              EMPLOYEES           FIRST_NAME          NO      NO
HR              EMPLOYEES           HIRE_DATE           NO      NO
HR              EMPLOYEES           JOB_ID              NO      NO
HR              EMPLOYEES           LAST_NAME           NO      NO
HR              EMPLOYEES           PHONE_NUMBER        NO      NO
HR              EMPLOYEES           SALARY              NO      NO
```

Note: You can also stop conflict detection for changes that are applied to remote non-Oracle databases. This query does not display such specifications because it lists a specification only if the APPLY_DATABASE_LINK column is NULL.

See Also:

- "Control Over Conflict Detection for Nonkey Columns" on page 3-4
- "Stopping Conflict Detection for Nonkey Columns" on page 9-25

Displaying Information About Update Conflict Handlers

When you specify an update conflict handler using the SET_UPDATE_CONFLICT_HANDLER procedure in the DBMS_APPLY_ADM package, the update conflict handler is run for all apply processes in the database, when a relevant conflict occurs.

The query in this section displays all of the columns for which conflict resolution has been specified using a prebuilt update conflict handler. That is, it shows the columns in all of the column lists specified in the database. This query also shows the type of prebuilt conflict handler specified and the resolution column specified for the column list.

To display information about all of the update conflict handlers in a database, run the following query:

```
COLUMN OBJECT_OWNER HEADING 'Table|Owner' FORMAT A5
COLUMN OBJECT_NAME HEADING 'Table Name' FORMAT A12
COLUMN METHOD_NAME HEADING 'Method' FORMAT A12
COLUMN RESOLUTION_COLUMN HEADING 'Resolution|Column' FORMAT A13
COLUMN COLUMN_NAME HEADING 'Column Name' FORMAT A30

SELECT OBJECT_OWNER,
       OBJECT_NAME,
       METHOD_NAME,
       RESOLUTION_COLUMN,
       COLUMN_NAME
  FROM DBA_APPLY_CONFLICT_COLUMNS
  ORDER BY OBJECT_OWNER, OBJECT_NAME, RESOLUTION_COLUMN;
```

Your output looks similar to the following:

```
Table                          Resolution
Owner Table Name   Method      Column       Column Name
----- ------------ ----------- ------------ ------------------------------
HR    COUNTRIES    MAXIMUM     TIME         COUNTRY_NAME
HR    COUNTRIES    MAXIMUM     TIME         REGION_ID
HR    COUNTRIES    MAXIMUM     TIME         TIME
HR    DEPARTMENTS  MAXIMUM     TIME         DEPARTMENT_NAME
HR    DEPARTMENTS  MAXIMUM     TIME         LOCATION_ID
HR    DEPARTMENTS  MAXIMUM     TIME         MANAGER_ID
HR    DEPARTMENTS  MAXIMUM     TIME         TIME
```

See Also:

- Chapter 3, "Streams Conflict Resolution"
- "Managing Streams Conflict Detection and Resolution" on page 9-22

Monitoring Streams Tags

The following sections contain queries that you can run to display the Streams tag for the current session and the default tag for each apply process:

- Displaying the Tag Value for the Current Session
- Displaying the Default Tag Value for Each Apply Process

See Also:

- Chapter 4, "Streams Tags"
- "Managing Streams Tags" on page 9-26
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_STREAMS package

Displaying the Tag Value for the Current Session

You can display the tag value generated in all redo entries for the current session by querying the DUAL view:

```
SELECT DBMS_STREAMS.GET_TAG FROM DUAL;
```

Your output looks similar to the following:

```
GET_TAG
--------------------------------------------------------------------------------
1D
```

You can also determine the tag for a session by calling the DBMS_STREAMS.GET_TAG function.

Displaying the Default Tag Value for Each Apply Process

You can get the default tag for all redo entries generated by each apply process by querying for the APPLY_TAG value in the DBA_APPLY data dictionary view. For example, to get the hexadecimal value of the default tag generated in the redo entries by each apply process, run the following query:

```
COLUMN APPLY_NAME HEADING 'Apply Process Name' FORMAT A30
COLUMN APPLY_TAG HEADING 'Tag Value' FORMAT A30

SELECT APPLY_NAME, APPLY_TAG FROM DBA_APPLY;
```

Your output looks similar to the following:

```
Apply Process Name             Tag Value
------------------------------ ------------------------------
APPLY_FROM_MULT2               00
APPLY_FROM_MULT3               00
```

A handler or custom rule-based transformation function associated with an apply process can get the tag by calling the DBMS_STREAMS.GET_TAG function.

Monitoring Instantiation

The following sections contain queries that you can run to determine which database objects are prepared for instantiation at a source database and the instantiation SCN for database objects at a destination database:

- Determining Which Database Objects Are Prepared for Instantiation
- Determining the Tables for Which an Instantiation SCN Has Been Set

See Also:

- "Overview of Instantiation and Streams Replication" on page 2-1
- Chapter 10, "Performing Instantiations"

Determining Which Database Objects Are Prepared for Instantiation

You prepare a database object for instantiation using one of the following procedures in the DBMS_CAPTURE_ADM package:

- PREPARE_TABLE_INSTANTIATION prepares a single table for instantiation.
- PREPARE_SCHEMA_INSTANTIATION prepares all of the database objects in a schema for instantiation.
- PREPARE_GLOBAL_INSTANTIATION prepares all of the database objects in a database for instantiation.

To determine which database objects have been prepared for instantiation, query the following corresponding data dictionary views:

- DBA_CAPTURE_PREPARED_TABLES
- DBA_CAPTURE_PREPARED_SCHEMAS
- DBA_CAPTURE_PREPARED_DATABASE

For example, to list all of the tables that have been prepared for instantiation, the SCN for the time when each table was prepared, and the time when each table was prepared, run the following query:

```
COLUMN TABLE_OWNER HEADING 'Table Owner' FORMAT A15
COLUMN TABLE_NAME HEADING 'Table Name' FORMAT A15
COLUMN SCN HEADING 'Prepare SCN' FORMAT 99999999999
COLUMN TIMESTAMP HEADING 'Time Ready for|Instantiation'

SELECT TABLE_OWNER,
       TABLE_NAME,
       SCN,
       TO_CHAR(TIMESTAMP, 'HH24:MI:SS MM/DD/YY') TIMESTAMP
  FROM DBA_CAPTURE_PREPARED_TABLES;
```

Your output looks similar to the following:

```
                                                  Time Ready for
Table Owner     Table Name            Prepare SCN Instantiation
--------------- --------------- ----------------- -----------------
HR              COUNTRIES                  196655 12:59:30 02/28/02
HR              DEPARTMENTS                196658 12:59:30 02/28/02
HR              EMPLOYEES                  196659 12:59:30 02/28/02
HR              JOBS                       196660 12:59:30 02/28/02
HR              JOB_HISTORY                196661 12:59:30 02/28/02
HR              LOCATIONS                  196662 12:59:30 02/28/02
HR              REGIONS                    196664 12:59:30 02/28/02
```

See Also: "Preparing Database Objects for Instantiation at a Source Database" on page 10-1

Determining the Tables for Which an Instantiation SCN Has Been Set

An instantiation SCN is set at a destination database. It controls which captured LCRs for a table are ignored by an apply process and which captured LCRs for a database object are applied by an apply process. If the commit SCN of an LCR for a table from a source database is less than or equal to the instantiation SCN for that table at a destination database, then the apply process at the destination database discards the LCR. Otherwise, the apply process applies the LCR.

You can set an instantiation SCN using one of the following procedures in the `DBMS_APPLY_ADM` package:

- `SET_TABLE_INSTANTIATION_SCN` sets the instantiation SCN for a single table.
- `SET_SCHEMA_INSTANTIATION_SCN` sets the instantiation SCN for a schema, and, optionally, for all of the database objects in the schema.
- `SET_GLOBAL_INSTANTIATION_SCN` sets the instantiation SCN for a database, and, optionally, for all of the database objects in the database.

To determine which database objects have a set instantiation SCN, query the following corresponding data dictionary views:

- `DBA_APPLY_INSTANTIATED_OBJECTS`
- `DBA_APPLY_INSTANTIATED_SCHEMAS`
- `DBA_APPLY_INSTANTIATED_GLOBAL`

The following query lists each table for which an instantiation SCN has been set at a destination database and the instantiation SCN for each table:

```
COLUMN SOURCE_DATABASE HEADING 'Source Database' FORMAT A15
COLUMN SOURCE_OBJECT_OWNER HEADING 'Object Owner' FORMAT A15
COLUMN SOURCE_OBJECT_NAME HEADING 'Object Name' FORMAT A15
COLUMN INSTANTIATION_SCN HEADING 'Instantiation SCN' FORMAT 99999999999

SELECT SOURCE_DATABASE,
       SOURCE_OBJECT_OWNER,
       SOURCE_OBJECT_NAME,
       INSTANTIATION_SCN
  FROM DBA_APPLY_INSTANTIATED_OBJECTS
  WHERE APPLY_DATABASE_LINK IS NULL;
```

Your output looks similar to the following:

```
Source Database Object Owner    Object Name     Instantiation SCN
--------------- --------------- --------------- -----------------
DBS1.NET        HR              REGIONS                    196660
DBS1.NET        HR              COUNTRIES                  196660
DBS1.NET        HR              LOCATIONS                  196660
```

Note: You can also display instantiation SCNs for changes that are applied to remote non-Oracle databases. This query does not display these instantiation SCNs because it lists an instantiation SCN only if the `APPLY_DATABASE_LINK` column is `NULL`.

See Also: "Setting Instantiation SCNs at a Destination Database" on page 10-27

Running Flashback Queries in a Streams Replication Environment

Oracle Flashback Query enables you to view and repair historical data. You can perform queries on a database as of a certain clock time or system change number (SCN). In a Streams single-source replication environment, you can use Flashback Query at the source database and a destination database at a past time when the replicated database objects should be identical.

Running the queries at corresponding SCNS at the source and destination databases can be used to determine whether all of the changes to the replicated objects performed at the source database have been applied at the destination database. If there are apply errors at the destination database, then such a Flashback Query can show how the replicated objects looked at the time when the error was raised. This information could be useful in determining the cause of the error and the best way to correct the error.

Running a Flashback Query at each database can also check whether tables have certain rows at the corresponding SCNs. If the table data does not match at the corresponding SCNs, then there is a problem with the replication environment.

To run queries, the Streams replication environment must have the following characteristics:

- The replication environment must be a single-source environment, where changes to replicated objects are captured at only one database.
- No modifications are made to the replicated objects in the Stream. That is, no transformations, subset rules (row migration), or apply handlers modify the LCRs for the replicated objects.
- No DML or DDL changes are made to the replicated objects at the destination database.
- Both the source database and the destination database must be configured to use Oracle Flashback, and the Streams administrator at both databases must be able to execute subprograms in the `DBMS_FLASHBACK` package.
- The information in the undo tablespace must go back far enough to perform the query at each database. Oracle Flashback features use the Automatic Undo Management system to obtain historical data and metadata for a transaction. The `UNDO_RETENTION` initialization parameter at each database must be set to a value that is large enough to perform the Flashback Query.

Because Streams replication is asynchronous, you cannot use a past time in the Flashback Query. However, you can use the GET_SCN_MAPPING procedure in the DBMS_STREAMS_ADM package to determine the SCN at the destination database that corresponds to an SCN at the source database.

These instructions assume that you know the SCN for the Flashback Query at the source database. Using this SCN, you can determine the corresponding SCN for the Flashback Query at the destination database. To run these queries, complete the following steps:

1. At the destination database, ensure that the archived redo log file for the approximate time of the Flashback Query is available to the database. The GET_SCN_MAPPING procedure requires that this redo log file be available.
2. While connected as the Streams administrator at the destination database, run the GET_SCN_MAPPING procedure. In this example, assume that the SCN for the source database is 52073983 and that the name of the apply process that applies changes from the source database is strm01_apply:

```
SET SERVEROUTPUT ON
DECLARE
  dest_scn   NUMBER;
  start_scn  NUMBER;
  dest_skip  DBMS_UTILITY.NAME_ARRAY;
BEGIN
  DBMS_STREAMS_ADM.GET_SCN_MAPPING(
    apply_name             => 'strm01_apply',
    src_pit_scn            => '52073983',
    dest_instantiation_scn => dest_scn,
    dest_start_scn         => start_scn,
    dest_skip_txn_ids      => dest_skip);
  IF dest_skip.count = 0 THEN
    DBMS_OUTPUT.PUT_LINE('No Skipped Transactions');
    DBMS_OUTPUT.PUT_LINE('Destination SCN: ' || dest_scn);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Destination SCN invalid for Flashback Query.');
    DBMS_OUTPUT.PUT_LINE('At least one transaction was skipped.');
  END IF;
END;
/
```

   If a valid destination SCN is returned, then proceed to Step 3.

   If the destination SCN was not valid for Flashback Query because one or more transactions were skipped by the apply process, then the apply process parameter commit_serialization was set to none, and nondependent transactions have been applied out of order. There is at least one transaction with a source commit SCN less than src_pit_scn that was committed at the destination database after the returned dest_instantiation_scn. Therefore, tables might not be the same at the source and destination databases for the specified source SCN. You can choose a different source SCN and restart at Step 1.
3. Run the Flashback Query at the source database using the source SCN.
4. Run the Flashback Query at the destination database using the SCN returned in Step 2.
5. Compare the results of the queries in Steps 3 and 4 and take any necessary action.

See Also:

- *Oracle Database Concepts* and *Oracle Database Application Developer's Guide - Fundamentals* for more information about Flashback Query
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `GET_SCN_MAPPING` procedure

13

Troubleshooting Streams Replication

This chapter contains information about identifying and resolving common problems in a Streams replication environment.

This chapter contains these topics:

- Recovering from Configuration Errors
- Troubleshooting an Apply Process in a Replication Environment

> **See Also:** *Oracle Streams Concepts and Administration* for more information about troubleshooting Streams environments

Recovering from Configuration Errors

The following procedures in the `DBMS_STREAMS_ADM` package configure a replication environment that is maintained by Streams:

- `MAINTAIN_GLOBAL`
- `MAINTAIN_SCHEMAS`
- `MAINTAIN_SIMPLE_TTS`
- `MAINTAIN_TABLES`
- `MAINTAIN_TTS`
- `PRE_INSTANTIATION_SETUP` and `POST_INSTANTIATION_SETUP`

When one of these procedures configures the replication environment directly (with the `perform_actions` parameter is set to `true`), information about the configuration actions is stored in the following data dictionary views when the procedure is running:

- `DBA_RECOVERABLE_SCRIPT`
- `DBA_RECOVERABLE_SCRIPT_PARAMS`
- `DBA_RECOVERABLE_SCRIPT_BLOCKS`
- `DBA_RECOVERABLE_SCRIPT_ERRORS`

When the procedure completes successfully, metadata about the configuration operation is purged from these views. However, when one of these procedures encounters an error and stops, metadata about the configuration operation remains in these views. Typically, these procedures encounter errors when one or more prerequisites for running them is not met.

When one of these procedures encounters an error, you can use the RECOVER_OPERATION procedure in the DBMS_STREAMS_ADM package to either roll the operation forward, roll the operation back, or purge the metadata about the operation. Specifically, the operation_mode parameter in the RECOVER_OPERATION procedure provides the following options:

- FORWARD: This option attempts to complete the configuration operation from the point at which it failed. Before specifying this option, correct the conditions that caused the errors reported in the DBA_RECOVERABLE_SCRIPT_ERRORS view.
- ROLLBACK: This option rolls back all of the actions performed by the configuration procedure. If the rollback is successful, then this options also purges the metadata about the operation in the data dictionary views described previously.
- PURGE: This option purges the metadata about the operation in the data dictionary views described previously without rolling the operation back.

Note:

- If the perform_actions parameter is set to false when one of the configuration procedures is run, and a script is used to configure the Streams replication environment, then the data dictionary views are not populated, and the RECOVER_OPERATION procedure cannot be used for the operation.
- To run the RECOVER_OPERATION procedure, both databases must be Oracle Database 10*g* Release 2 databases.

See Also:

- "Configuring Replication Using the DBMS_STREAMS_ADM Package" on page 6-5 for more information about configuring a Streams replication environment with these procedures
- "Tasks to Complete Before Configuring Streams Replication" on page 6-12 for information about prerequisites that must be met before running these procedures

Recovery Scenario

This section contains a scenario in which the MAINTAIN_SCHEMAS procedure stops because it encounters an error. Assume that the following procedure encountered an error when it was run at the capture database:

```
BEGIN
  DBMS_STREAMS_ADM.MAINTAIN_SCHEMAS(
    schema_names                 => 'hr',
    source_directory_object      => 'SOURCE_DIRECTORY',
    destination_directory_object => 'DEST_DIRECTORY',
    source_database              => 'inst1.net',
    destination_database         => 'inst2.net',
    perform_actions              => true,
    dump_file_name               => 'export_hr.dmp',
    capture_queue_table          => 'rep_capture_queue_table',
    capture_queue_name           => 'rep_capture_queue',
    capture_queue_user           => NULL,
    apply_queue_table            => 'rep_dest_queue_table',
    apply_queue_name             => 'rep_dest_queue',
    apply_queue_user             => NULL,
```

```
    capture_name                => 'capture_hr',
    propagation_name            => 'prop_hr',
    apply_name                  => 'apply_hr',
    log_file                    => 'export_hr.clg',
    bi_directional              => false,
    include_ddl                 => true,
    instantiation               => DBMS_STREAMS_ADM.INSTANTIATION_SCHEMA);
END;
/
```

Complete the following steps to diagnose the problem and recover the operation:

1. Query the DBA_RECOVERABLE_SCRIPT_ERRORS data dictionary view at the capture database to determine the error:

```
COLUMN SCRIPT_ID     HEADING 'Script ID'     FORMAT A35
COLUMN BLOCK_NUM     HEADING 'Block|Number' FORMAT 999999
COLUMN ERROR_MESSAGE HEADING 'Error Message' FORMAT A33

SELECT SCRIPT_ID, BLOCK_NUM, ERROR_MESSAGE FROM DBA_RECOVERABLE_SCRIPT_ERRORS;
```

   The query returns the following output:

```
                                    Block
Script ID                          Number Error Message
---------------------------------- ------- --------------------------------
F73ED2C9E96B27B0E030578CB10B2424         12 ORA-39001: invalid argument value
```

2. Query the DBA_RECOVERABLE_SCRIPT_BLOCKS data dictionary view for the script ID and block number returned in Step 1 for information about the block in which the error occurred. For example, if the script ID is F73ED2C9E96B27B0E030578CB10B2424 and the block number is 12, run the following query:

```
COLUMN FORWARD_BLOCK        HEADING 'Forward Block'               FORMAT A50
COLUMN FORWARD_BLOCK_DBLINK HEADING 'Forward Block|Database Link' FORMAT A13
COLUMN STATUS               HEADING 'Status'                      FORMAT A12

SET LONG 10000
SELECT FORWARD_BLOCK,
       FORWARD_BLOCK_DBLINK,
       STATUS
  FROM DBA_RECOVERABLE_SCRIPT_BLOCKS
  WHERE SCRIPT_ID = 'F73ED2C9E96B27B0E030578CB10B2424' AND
        BLOCK_NUM = 12;
```

   The output contains the following information:

   - The FORWARD_BLOCK column contains detailed information about the actions performed by the procedure in the specified block. If necessary, spool the output into a file. In this scenario, the FORWARD_BLOCK column for block 12 contains the code for the Data Pump export.
   - The FORWARD_BLOCK_DBLINK column shows the database where the block is executed. In this scenario, the FORWARD_BLOCK_DBLINK column for block 12 shows INST1.NET because the Data Pump export was being performed on INST1.NET when the error occurred.
   - The STATUS column shows the status of the block execution. In this scenario, the STATUS column for block 12 shows ERROR.

3. Interpret the output of the queries and diagnose the problem. The output returned in Step 1 provides the following information:

   - The unique identifier for the configuration operation is `F73ED2C9E96B27B0E030578CB10B2424`. This value is the `RAW` value returned in the `SCRIPT_ID` field.
   - Only one Streams configuration procedure is in the process of running because only one row was returned by the query. If more than one row was returned by the query, then query the `DBA_RECOVERABLE_SCRIPT` and `DBA_RECOVERABLE_SCRIPT_PARAMS` views to determine which script ID applies to the configuration operation.
   - The cause in *Oracle Database Error Messages* for the `ORA-39001` error is the following: The user specified API parameters were of the wrong type or value range. Subsequent messages supplied by `DBMS_DATAPUMP.GET_STATUS` will further describe the error.
   - The query on the `DBA_RECOVERABLE_SCRIPT_BLOCKS` view shows that the error occurred during Data Pump export.

   The output from the queries shows that the `MAINTAIN_SCHEMAS` procedure encountered a Data Pump error. Notice that the `instantiation` parameter in the `MAINTAIN_SCHEMAS` procedure was set to `DBMS_STREAMS_ADM.INSTANTIATION_SCHEMA`. This setting means that the `MAINTAIN_SCHEMAS` procedure performs the instantiation using a Data Pump export and import. A Data Pump export dump file is generated to complete the export/import.

   Data Pump errors usually are caused by one of the following conditions:

   - One or more of the directory objects used to store the export dump file do not exist.
   - The user running the procedure does not have access to specified directory objects.
   - An export dump file with the same name as the one generated by the procedure already exists in a directory specified in the `source_directory_object` or `destination_directory_object` parameter.

4. Query the `DBA_RECOVERABLE_SCRIPT_PARAMS` data dictionary view at the capture database to determine the names of the directory objects specified when the `MAINTAIN_SCHEMAS` procedure was run:

```
COLUMN PARAMETER HEADING 'Parameter' FORMAT A30
COLUMN VALUE     HEADING 'Value'     FORMAT A45

SELECT PARAMETER,
       VALUE
       FROM DBA_RECOVERABLE_SCRIPT_PARAMS
       WHERE SCRIPT_ID = 'F73ED2C9E96B27B0E030578CB10B2424';
```

The query returns the following output:

```
Parameter                      Value
------------------------------ -------------------------------------------
SOURCE_DIRECTORY_OBJECT        SOURCE_DIRECTORY
DESTINATION_DIRECTORY_OBJECT   DEST_DIRECTORY
SOURCE_DATABASE                INST1.NET
DESTINATION_DATABASE           INST2.NET
CAPTURE_QUEUE_TABLE            REP_CAPTURE_QUEUE_TABLE
CAPTURE_QUEUE_OWNER            STRMADMIN
CAPTURE_QUEUE_NAME             REP_CAPTURE_QUEUE
CAPTURE_QUEUE_USER
APPLY_QUEUE_TABLE              REP_DEST_QUEUE_TABLE
APPLY_QUEUE_OWNER              STRMADMIN
APPLY_QUEUE_NAME               REP_DEST_QUEUE
APPLY_QUEUE_USER
CAPTURE_NAME                   CAPTURE_HR
APPLY_NAME                     APPLY_HR
PROPAGATION_NAME               PROP_HR
INSTANTIATION                  INSTANTIATION_SCHEMA
BI_DIRECTIONAL                 TRUE
INCLUDE_DDL                    TRUE
LOG_FILE                       export_hr.clg
DUMP_FILE_NAME                 export_hr.dmp
SCHEMA_NAMES                   HR
```

5. Make sure the directory object specified for the `source_directory_object` parameter exists at the source database, and make sure the directory object specified for the `destination_directory_object` parameter exists at the destination database. Check for these directory objects by querying the `DBA_DIRECTORIES` data dictionary view.

   For this scenario, assume that the `SOURCE_DIRECTORY` directory object does not exist at the source database, and the `DEST_DIRECTORY` directory object does not exist at the destination database. The Data Pump error occurred because the directory objects used for the export dump file did not exist.

6. Create the required directory objects at the source and destination databases using the SQL statement `CREATE DIRECTORY`. See "Create the Required Directory Objects" on page 6-14 for instructions.

7. Run the `RECOVER_OPERATION` procedure at the capture database:

   ```
   BEGIN
     DBMS_STREAMS_ADM.RECOVER_OPERATION(
       script_id       => 'F73ED2C9E96B27B0E030578CB10B2424',
       operation_mode  => 'FORWARD');
   END;
   /
   ```

   Notice that the `script_id` parameter is set to the value determined in Step 1, and the `operation_mode` parameter is set to `FORWARD` to complete the configuration. Also, the `RECOVER_OPERATION` procedure must be run at the database where the configuration procedure was run.

Troubleshooting an Apply Process in a Replication Environment

The following sections provide information about troubleshooting apply process problems in a replication environment:

- Is the Apply Process Encountering Contention?
- Is the Apply Process Waiting for a Dependent Transaction?
- Is an Apply Server Performing Poorly for Certain Transactions?
- Are There Any Apply Errors in the Error Queue?

Is the Apply Process Encountering Contention?

An apply server is a component of an apply process. Apply servers apply DML and DDL changes to database objects at a destination database. An apply process can use one or more apply servers, and the `parallelism` apply process parameter specifies the number of apply servers that can concurrently apply transactions. For example, if `parallelism` is set to `5`, then an apply process uses a total of five apply servers.

An apply server encounters contention when the apply server must wait for a resource that is being used by another session. Contention can result from logical dependencies. For example, when an apply server tries to apply a change to a row that a user has locked, then the apply server must wait for the user. Contention can also result from physical dependencies. For example, interested transaction list (ITL) contention results when two transactions that are being applied, which might not be logically dependent, are trying to lock the same block on disk. In this case, one apply server locks rows in the block, and the other apply server must wait for access to the block, even though the second apply server is trying to lock different rows. See "Is the Apply Process Waiting for a Dependent Transaction?" on page 13-7 for detailed information about ITL contention.

When an apply server encounters contention that does not involve another apply server in the same apply process, it waits until the contention clears. When an apply server encounters contention that involves another apply server in the same apply process, one of the two apply servers is rolled back. An apply process that is using multiple apply servers might be applying multiple transactions at the same time. The apply process tracks the state of the apply server that is applying the transaction with the lowest commit SCN. If there is a dependency between two transactions, then an apply process always applies the transaction with the lowest commit SCN first. The transaction with the higher commit SCN waits for the other transaction to commit. Therefore, if the apply server with the lowest commit SCN transaction is encountering contention, then the contention results from something other than a dependent transaction. In this case, you can monitor the apply server with the lowest commit SCN transaction to determine the cause of the contention.

The following four wait states are possible for an apply server:

- **Not waiting:** The apply server is not encountering contention and is not waiting. No action is necessary in this case.
- **Waiting for an event that is not related to another session:** An example of an event that is not related to another session is a `log file sync` event, where redo data must be flushed because of a commit or rollback. In these cases, nothing is written to the log initially because such waits are common and are usually transient. If the apply server is waiting for the same event after a certain interval of time, then the apply server writes a message to the alert log and apply process trace file. For example, an apply server `a001` might write a message similar to the following:

```
A001: warning -- apply server 1, sid 26 waiting for event:
A001: [log file sync] ...
```

This output is written to the alert log at intervals until the problem is rectified.

- **Waiting for an event that is related to a non apply server session:** The apply server writes a message to the alert log and apply process trace file immediately. For example, an apply server `a001` might write a message similar to the following:

```
A001: warning -- apply server 1, sid 10 waiting on user sid 36 for event:
A001: [enq: TM - contention] name|mode=544d0003, object #=a078,
      table/partition=0
```

This output is written to the alert log at intervals until the problem is rectified.

- **Waiting for another apply server session:** This state can be caused by interested transaction list (ITL) contention, but it can also be caused by more serious issues, such as an apply handler that obtains conflicting locks. In this case, the apply server that is blocked by another apply server prints only once to the alert log and the trace file for the apply process, and the blocked apply server issues a rollback to the blocking apply server. When the blocking apply server rolls back, another message indicating that the apply server has been rolled back is printed to the log files, and the rolled back transaction is reassigned by the coordinator process for the apply process.

  For example, if apply server 1 of apply process `a001` is blocked by apply server 2 of the same apply process (`a001`), then the apply process writes the following messages to the log files:

```
A001: apply server 1 blocked on server 2
A001: [enq: TX - row lock contention] name|mode=54580006, usn<<16 |
      slot=1000e, sequence=1853
A001: apply server 2 rolled back
```

  You can determine the total number of times an apply server was rolled back since the apply process last started by querying the `TOTAL_ROLLBACKS` column in the `V$STREAMS_APPLY_COORDINATOR` dynamic performance view.

  See Also:

  - *Oracle Database Performance Tuning Guide* for more information about contention and about resolving different types of contention
  - *Oracle Streams Concepts and Administration* for more information about trace files and the alert log

Is the Apply Process Waiting for a Dependent Transaction?

If you set the `parallelism` parameter for an apply process to a value greater than `1`, and you set the `commit_serialization` parameter of the apply process to `full`, then the apply process can detect interested transaction list (ITL) contention if there is a transaction that is dependent on another transaction with a higher SCN. ITL contention occurs if the session that created the transaction waited for an ITL slot in a block. This happens when the session wants to lock a row in the block, but one or more other sessions have rows locked in the same block, and there is no free ITL slot in the block.

ITL contention also is possible if the session is waiting due to a shared bitmap index fragment. Bitmap indexes index key values and a range of rowids. Each entry in a bitmap index can cover many rows in the actual table. If two sessions want to update

rows covered by the same bitmap index fragment, then the second session waits for the first transaction to either COMMIT or ROLLBACK.

When an apply process detects such a dependency, it resolves the ITL contention automatically and records information about it in the alert log and apply process trace file for the database. ITL contention can negatively affect the performance of an apply process because there might not be any progress while it is detecting the deadlock.

To avoid the problem in the future, perform one of the following actions:

- Increase the number of ITLs available. You can do so by changing the INITRANS setting for the table using the ALTER TABLE statement.
- Set the commit_serialization parameter to none for the apply process.
- Set the parallelism apply process parameter to 1 for the apply process.

See Also:

- *Oracle Streams Concepts and Administration* for more information about apply process parameters and about checking the trace files and alert log for problems
- *Oracle Database Administrator's Guide* and *Oracle Database SQL Reference* for more information about INITRANS

Is an Apply Server Performing Poorly for Certain Transactions?

If an apply process is not performing well, then the reason might be that one or more apply servers used by the apply process are taking an inordinate amount of time to apply certain transactions. The following query displays information about the transactions being applied by each apply server used by an apply process named strm01_apply:

```
COLUMN SERVER_ID HEADING 'Apply Server ID' FORMAT 99999999
COLUMN STATE HEADING 'Apply Server State' FORMAT A20
COLUMN APPLIED_MESSAGE_NUMBER HEADING 'Applied Message|Number' FORMAT 99999999
COLUMN MESSAGE_SEQUENCE HEADING 'Message Sequence|Number' FORMAT 99999999

SELECT SERVER_ID, STATE, APPLIED_MESSAGE_NUMBER, MESSAGE_SEQUENCE
  FROM V$STREAMS_APPLY_SERVER
  WHERE APPLY_NAME = 'STRM01_APPLY'
  ORDER BY SERVER_ID;
```

If you run this query repeatedly, then over time the apply server state, applied message number, and message sequence number should continue to change for each apply server as it applies transactions. If these values do not change for one or more apply servers, then the apply server might not be performing well. In this case, you should make sure that, for each table to which the apply process applies changes, every key column has an index.

If you have many such tables, then you might need to determine the specific table and DML or DDL operation that is causing an apply server to perform poorly. To do so, run the following query when an apply server is taking an inordinately long time to apply a transaction. In this example, assume that the name of the apply process is strm01_apply and that apply server number two is performing poorly:

```
COLUMN OPERATION HEADING 'Operation' FORMAT A20
COLUMN OPTIONS HEADING 'Options' FORMAT A20
COLUMN OBJECT_OWNER HEADING 'Object|Owner' FORMAT A10
COLUMN OBJECT_NAME HEADING 'Object|Name' FORMAT A10
COLUMN COST HEADING 'Cost' FORMAT 99999999
```

```
SELECT p.OPERATION, p.OPTIONS, p.OBJECT_OWNER, p.OBJECT_NAME, p.COST
  FROM V$SQL_PLAN p, V$SESSION s, V$STREAMS_APPLY_SERVER a
  WHERE a.APPLY_NAME = 'STRM01_APPLY' AND a.SERVER_ID = 2
    AND s.SID = a.SID
    AND p.HASH_VALUE = s.SQL_HASH_VALUE;
```

This query returns the operation being performed currently by the specified apply server. The query also returns the owner and name of the table on which the operation is being performed and the cost of the operation. Make sure each key column in this table has an index. If the results show `FULL` for the `COST` column, then the operation is causing full table scans, and indexing the table's key columns might solve the problem.

In addition, you can run the following query to determine the specific DML or DDL SQL statement that is causing an apply server to perform poorly, assuming that the name of the apply process is `strm01_apply` and that apply server number two is performing poorly:

```
SELECT t.SQL_TEXT
  FROM V$SESSION s, V$SQLTEXT t, V$STREAMS_APPLY_SERVER a
  WHERE a.APPLY_NAME = 'STRM01_APPLY' AND a.SERVER_ID = 2
    AND s.SID = a.SID
    AND s.SQL_ADDRESS = t.ADDRESS
    AND s.SQL_HASH_VALUE = t.HASH_VALUE
    ORDER BY PIECE;
```

This query returns the SQL statement being run currently by the specified apply server. The statement includes the name of the table to which the transaction is being applied. Make sure each key column in this table has an index.

If the SQL statement returned by the previous query is less than one thousand characters long, then you can run the following simplified query instead:

```
SELECT t.SQL_TEXT
  FROM V$SESSION s, V$SQLAREA t, V$STREAMS_APPLY_SERVER a
  WHERE a.APPLY_NAME = 'STRM01_APPLY' AND a.SERVER_ID = 2
    AND s.SID = a.SID
    AND s.SQL_ADDRESS = t.ADDRESS
    AND s.SQL_HASH_VALUE = t.HASH_VALUE;
```

> **See Also:** *Oracle Database Performance Tuning Guide* and *Oracle Database Reference* for more information about the `V$SQL_PLAN` dynamic performance view

Are There Any Apply Errors in the Error Queue?

When an apply process cannot apply a message, it moves the message and all of the other messages in the same transaction into the error queue. You should check for apply errors periodically to see if there are any transactions that could not be applied. You can check for apply errors by querying the `DBA_APPLY_ERROR` data dictionary view.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about checking for apply errors and about managing apply errors

Using a DML Handler to Correct Error Transactions

When an apply process moves a transaction to the error queue, you can examine the transaction to analyze the feasibility reexecuting the transaction successfully. If an abnormality is found in the transaction, then you might be able to configure a DML handler to correct the problem. In this case, configure the DML handler to run when you reexecute the error transaction.

When a DML handler is used to correct a problem in an error transaction, the apply process that uses the DML handler should be stopped to prevent the DML handler from acting on LCRs that are not involved with the error transaction. After successful reexecution, if the DML handler is no longer needed, then remove it. Also, correct the problem that caused the transaction to moved to the error queue to prevent future error transactions.

See Also: "Creating a DML Handler" on page 9-12

Troubleshooting Specific Apply Errors

You might encounter the following types of apply process errors for LCRs:

- ORA-01031 Insufficient Privileges
- ORA-01403 No Data Found
- ORA-23605 Invalid Value for Streams Parameter*
- ORA-23607 Invalid Column*
- ORA-24031 Invalid Value, parameter_name Should Be Non-NULL*
- ORA-26687 Instantiation SCN Not Set
- ORA-26688 Missing Key in LCR
- ORA-26689 Column Type Mismatch*

The errors marked with an asterisk (*) in the previous list often result from a problem with an apply handler or a rule-based transformation.

ORA-01031 Insufficient Privileges An `ORA-01031` error occurs when the user designated as the apply user does not have the necessary privileges to perform SQL operations on the replicated objects. The apply user privileges must be granted by an explicit grant of each privilege. Granting these privileges through a role is not sufficient for the Streams apply user.

Specifically, the following privileges are required:

- For table level DML changes, the `INSERT`, `UPDATE`, `DELETE`, and `SELECT` privileges must be granted.
- For table level DDL changes, the `ALTER TABLE` privilege must be granted.
- For schema level changes, the `CREATE ANY TABLE`, `CREATE ANY INDEX`, `CREATE ANY PROCEDURE`, `ALTER ANY TABLE`, and `ALTER ANY PROCEDURE` privileges must be granted.
- For global level changes, `ALL PRIVILEGES` must be granted to the apply user.

To correct this error, complete the following steps:

1. Connect as the apply user on the destination database.
2. Query the `SESSION_PRIVS` data dictionary view to determine which required privileges are not granted to the apply user.

3. Connect as an administrative user who can grant privileges.
4. Grant the necessary privileges to the apply user.
5. Reexecute the error transactions in the error queue for the apply process.

See Also:

- "Apply and Streams Replication" on page 1-12 for more information about apply users
- *Oracle Streams Concepts and Administration* for information about reexecuting error transactions

ORA-01403 No Data Found Typically, an `ORA-01403` error occurs when an apply process tries to update an existing row and the `OLD_VALUES` in the row LCR do not match the current values at the destination database.

Typically, one of the following conditions causes this error:

- Supplemental logging is not specified for columns that require supplemental logging at the source database. In this case, LCRs from the source database might not contain values for key columns. You can use a DML handler to modify the LCR so that it contains the necessary supplemental data. See "Using a DML Handler to Correct Error Transactions" on page 13-10. Also, specify the necessary supplemental logging at the source database to prevent future errors.
- There is a problem with the primary key in the table for which an LCR is applying a change. In this case, make sure the primary key is enabled by querying the `DBA_CONSTRAINTS` data dictionary view. If no primary key exists for the table, or if the target table has a different primary key than the source table, then specify substitute key columns using the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package. You also might encounter error `ORA-23416` if a table being applied does not have a primary key. After you make these changes, you can reexecute the error transaction.
- The transaction being applied depends on another transaction which has not yet executed. For example, if a transaction tries to update an employee with an `employee_id` of `300`, but the row for this employee has not yet been inserted into the `employees` table, then the update fails. In this case, execute the transaction on which the error transaction depends. Then, reexecute the error transaction.
- There is a data mismatch between a row LCR and the table for which the LCR is applying a change. Make sure row data in the table at the destination database matches the row data in the LCR. When you are checking for differences in the data, if there are any `DATE` columns in the shared table, then make sure your query shows the hours, minutes, and seconds. If there is a mismatch, then you can use a DML handler to modify an LCR so that it matches the table. See "Using a DML Handler to Correct Error Transactions" on page 13-10.

Alternatively, you can update the current values in the row so that the row LCR can be applied successfully. If changes to the row are captured by a capture process at the destination database, then you probably do not want to replicate this manual change to destination databases. In this case, complete the following steps:

1. Set a tag in the session that corrects the row. Make sure you set the tag to a value that prevents the manual change from being replicated. For example, the tag can prevent the change from being captured by a capture process.

   ```
   EXEC DBMS_STREAMS.SET_TAG(tag => HEXTORAW('17'));
   ```

   In some environments, you might need to set the tag to a different value.

2. Update the row in the table so that the data matches the old values in the LCR.

3. Reexecute the error or reexecute all errors. To reexecute an error, run the `EXECUTE_ERROR` procedure in the `DBMS_APPLY_ADM` package, and specify the transaction identifier for the transaction that caused the error. For example:

   ```
   EXEC DBMS_APPLY_ADM.EXECUTE_ERROR(local_transaction_id => '5.4.312');
   ```

   Or, execute all errors for the apply process by running the `EXECUTE_ALL_ERRORS` procedure:

   ```
   EXEC DBMS_APPLY_ADM.EXECUTE_ALL_ERRORS(apply_name => 'APPLY');
   ```

4. If you are going to make other changes in the current session that you want to replicate destination databases, then reset the tag for the session to an appropriate value, as in the following example:

   ```
   EXEC DBMS_STREAMS.SET_TAG(tag => NULL);
   ```

   In some environments, you might need to set the tag to a value other than `NULL`.

   See Also:

   - "Supplemental Logging for Streams Replication" on page 1-8 and "Monitoring Supplemental Logging" on page 12-2
   - "Considerations for Applying DML Changes to Tables" on page 1-18 for information about possible causes of apply errors
   - Chapter 4, "Streams Tags"
   - *Oracle Streams Concepts and Administration* for more information about managing apply errors and for instructions that enable you to display detailed information about apply errors

ORA-23605 Invalid Value for Streams Parameter When calling row LCR (`SYS.LCR$_ROW_RECORD` type) member subprograms, an `ORA-23605` error might be raised if the values of the parameters passed by the member subprogram do not match the row LCR. For example, an error results if a member subprogram tries to add an old column value to an insert row LCR, or if a member subprogram tries to set the value of a LOB column to a number.

Row LCRs should contain the following old and new values, depending on the operation:

- A row LCR for an INSERT operation should contain new values but no old values.
- A row LCR for an UPDATE operation can contain both new values and old values.
- A row LCR for a DELETE operation should contain old values but no new values.

Verify that the correct parameter type (OLD, or NEW, or both) is specified for the row LCR operation (INSERT, UPDATE, or DELETE). For example, if a DML handler or custom rule-based transformation changes an UPDATE row LCR into an INSERT row LCR, then the handler or transformation should remove the old values in the row LCR.

If an apply handler caused the error, then correct the apply handler and reexecute the error transaction. If a custom rule-based transformation caused the error, then you might be able to create a DML handler to correct the problem. See "Using a DML Handler to Correct Error Transactions" on page 13-10. Also, correct the rule-based transformation to avoid future errors.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about rule-based transformations

ORA-23607 Invalid Column An ORA-23607 error is raised by a row LCR (SYS.LCR$_ROW_RECORD type) member subprogram, when the value of the column_name parameter in the member subprogram does not match the name of any of the columns in the row LCR. Check the column names in the row LCR.

If an apply handler caused the error, then correct the apply handler and reexecute the error transaction. If a custom rule-based transformation caused the error, then you might be able to create a DML handler to correct the problem. See "Using a DML Handler to Correct Error Transactions" on page 13-10. Also, correct the rule-based transformation to avoid future errors.

An apply handler or custom rule-based transformation can cause this error by using one of the following row LCR member procedures:

- DELETE_COLUMN, if this procedure tries to delete a column from a row LCR that does not exist in the row LCR
- RENAME_COLUMN, if this procedure tries to rename a column that does not exist in the row LCR

In this case, to avoid similar errors in the future, perform one of the following actions:

- Instead of using an apply handler or custom rule-based transformation to delete or rename a column in row LCRs, use a declarative rule-based transformation. If a declarative rule-based transformation tries to delete or rename a column that does not exist, then the declarative rule-based transformation does not raise an error. You can specify a declarative rule-based transformation that deletes a column using the DBMS_STREAMS_ADM.DELETE_COLUMN procedure, and you can specify a declarative rule-based transformation that renames a column using the DBMS_STREAMS_ADM.RENAME_COLUMN procedure. You can use a declarative rule-based transformation in combination with apply handlers and custom rule-based transformations.

- If you want to continue to use an apply handler or custom rule-based transformation to delete or rename a column in row LCRs, then modify the handler or transformation to prevent future errors. For example, modify the handler or transformation to verify that a column exists before trying to rename or delete the column.

  See Also:

  - *Oracle Streams Concepts and Administration* for more information about rule-based transformations
  - *Oracle Database PL/SQL Packages and Types Reference* for more information about the DELETE_COLUMN and RENAME_COLUMN member procedures for row LCRs

ORA-24031 Invalid Value, *parameter_name* Should Be Non-NULL An ORA-24031 error can occur when an apply handler or a custom rule-based transformation passes a NULL value to an LCR member subprogram instead of an ANYDATA value that contains a NULL.

For example, the following call to the ADD_COLUMN member procedure for row LCRs can result in this error:

```
new_lcr.ADD_COLUMN('OLD','LANGUAGE',NULL);
```

The following example shows the correct way to call the ADD_COLUMN member procedure for row LCRs:

```
new_lcr.ADD_COLUMN('OLD','LANGUAGE',ANYDATA.ConvertVarchar2(NULL));
```

If an apply handler caused the error, then correct the apply handler and reexecute the error transaction. If a custom rule-based transformation caused the error, then you might be able to create a DML handler to correct the problem. See "Using a DML Handler to Correct Error Transactions" on page 13-10. Also, correct the rule-based transformation to avoid future errors.

> **See Also:** *Oracle Streams Concepts and Administration* for more information about rule-based transformations

ORA-26687 Instantiation SCN Not Set Typically, an ORA-26687 error occurs because the instantiation SCN is not set on an object for which an apply process is attempting to apply changes. You can query the DBA_APPLY_INSTANTIATED_OBJECTS data dictionary view to list the objects that have an instantiation SCN.

You can set an instantiation SCN for one or more objects by exporting the objects at the source database, and then importing them at the destination database. You can use either Data Pump export/import or original export/import. If you do not want to use export/import, then you can run one or more of the following procedures in the DBMS_APPLY_ADM package:

- SET_TABLE_INSTANTIATION_SCN
- SET_SCHEMA_INSTANTIATION_SCN
- SET_GLOBAL_INSTANTIATION_SCN

Some of the common reasons why an instantiation SCN is not set for an object at a destination database include the following:

- You used export/import for instantiation, and you exported the objects from the source database before preparing the objects for instantiation. You can prepare objects for instantiation either by creating Streams rules for the objects with the DBMS_STREAMS_ADM package or by running a procedure in the DBMS_CAPTURE_ADM package. If the objects were not prepared for instantiation before the export, then the instantiation SCN information will not be available in the export file, and the instantiation SCNs will not be set.

  In this case, prepare the database objects for instantiation at the source database by following the instructions in "Preparing Database Objects for Instantiation at a Source Database" on page 10-1. Next, set the instantiation SCN for the database objects at the destination database.

- You used original export/import for instantiation, and you performed the import without specifying y for the STREAMS_INSTANTIATION import parameter. If this parameter is not set to y for the import, then the instantiation SCN will not be set.

  In this case, repeat the original export/import operation, and set the STREAMS_INSTANTIATION parameter to y during import. Follow the instructions in "Instantiating Objects in a Tablespace Using Transportable Tablespace or RMAN" on page 10-7.

  Alternatively, use Data Pump export/import. An instantiation SCN is set for each imported prepared object automatically when you use Data Pump import.

- Instead of using export/import for instantiation, you set the instantiation SCN explicitly with the appropriate procedure in the DBMS_APPLY_ADM package. When the instantiation SCN is set explicitly by the database administrator, responsibility for the correctness of the data is assumed by the administrator.

  In this case, set the instantiation SCN for the database objects explicitly by following the instructions in "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29. Alternatively, you can choose to perform a metadata-only export/import to set the instantiation SCNs by following the instructions in "Setting Instantiation SCNs at a Destination Database" on page 10-27.

- You want to apply DDL changes, but you did not set the instantiation SCN at the schema or global level.

  In this case, set the instantiation SCN for the appropriate schemas by running the SET_SCHEMA_INSTANTIATION_SCN procedure, or set the instantiation SCN for the source database by running the SET_GLOBAL_INSTANTIATION_SCN procedure. Both of these procedures are in the DBMS_APPLY_ADM package. Follow the instructions in "Setting Instantiation SCNs Using the DBMS_APPLY_ADM Package" on page 10-29.

After you correct the condition that caused the error, whether you should reexecute the error transaction or delete it depends on whether the changes included in the transaction were executed at the destination database when you corrected the error condition. Follow these guidelines when you decide whether you should reexecute the transaction in the error queue or delete it:

- If you performed a new export/import, and the new export includes the transaction in the error queue, then delete the transaction in the error queue.
- If you set instantiation SCNs explicitly or reimported an existing export dump file, then reexecute the transaction in the error queue.

See Also:

- "Overview of Instantiation and Streams Replication" on page 2-1
- "Setting Instantiation SCNs at a Destination Database" on page 10-27
- *Oracle Streams Concepts and Administration* for information about reexecuting and deleting error transactions

ORA-26688 Missing Key in LCR Typically, an `ORA-26688` error occurs because of one of the following conditions:

- At least one LCR in a transaction does not contain enough information for the apply process to apply it. For dependency computation, an apply process always needs values for the defined primary key column(s) at the destination database. Also, if the parallelism of any apply process that will apply the changes is greater than 1, then the apply process needs values for any indexed column at a destination database, which includes unique or non unique index columns, foreign key columns, and bitmap index columns.

  If an apply process needs values for a column, and the column exists at the source database, then this error results when supplemental logging is not specified for one or more of these columns at the source database. In this case, specify the necessary supplemental logging at the source database to prevent apply errors.

  However, the definition of the source database table might be different than the definition of the corresponding destination database table. If an apply process needs values for a column, and the column exists at the destination database but *does not exist* at the source database, then you can configure a rule-based transformation to add the required values to the LCRs from the source database to prevent apply errors.

  To correct a transaction placed in the error queue because of this error, you can use a DML handler to modify the LCRs so that they contain the necessary supplemental data. See "Using a DML Handler to Correct Error Transactions" on page 13-10.

- There is a problem with the primary key in the table for which an LCR is applying a change. In this case, make sure the primary key is enabled by querying the `DBA_CONSTRAINTS` data dictionary view. If no primary key exists for the table, or if the destination table has a different primary key than the source table, then specify substitute key columns using the `SET_KEY_COLUMNS` procedure in the `DBMS_APPLY_ADM` package. You can also encounter error `ORA-23416` if a table does not have a primary key. After you make these changes, you can reexecute the error transaction.

  See Also:

  - "Supplemental Logging for Streams Replication" on page 1-8
  - "Substitute Key Columns" on page 1-19
  - *Oracle Streams Concepts and Administration* for more information about rule-based transformations

ORA-26689 Column Type Mismatch Typically, an `ORA-26689` error occurs because one or more columns at a table in the source database do not match the corresponding columns at the destination database. The LCRs from the source database might contain more columns than the table at the destination database, or there might be a column name or column type mismatch for one or more columns. If the columns differ at the databases, then you can use rule-based transformations to avoid future errors.

If you use an apply handler or a custom rule-based transformation, then make sure any `ANYDATA` conversion functions match the datatype in the LCR that is being converted. For example, if the column is specified as `VARCHAR2`, then use `ANYDATA.CONVERTVARCHAR2` function to convert the data from type `ANY` to `VARCHAR2`.

Also, make sure you use the correct character case in rule conditions, apply handlers, and rule-based transformations. For example, if a column name has all uppercase characters in the data dictionary, then you should specify the column name with all uppercase characters in rule conditions, apply handlers, and rule-based transformations

This error can also occur because supplemental logging is not specified where it is required for nonkey columns at the source database. In this case, LCRs from the source database might not contain needed values for these nonkey columns.

You might be able to configure a DML handler to apply the error transaction. See "Using a DML Handler to Correct Error Transactions" on page 13-10.

See Also:

- "Considerations for Applying DML Changes to Tables" on page 1-18 for information about possible causes of apply errors
- "Supplemental Logging for Streams Replication" on page 1-8 and "Monitoring Supplemental Logging" on page 12-2
- *Oracle Streams Replication Administrator's Guide* for information about rule-based transformations

Part IV

Sample Replication Environments

This part includes the following detailed examples that configure and maintain Streams replication environments:

- Chapter 14, "Simple Single-Source Replication Example"
- Chapter 15, "Single-Source Heterogeneous Replication Example"
- Chapter 16, "Multiple-Source Replication Example"

14

Simple Single-Source Replication Example

This chapter illustrates an example of a simple single-source replication environment that can be constructed using Streams.

This chapter contains these topics:

- Overview of the Simple Single-Source Replication Example
- Prerequisites

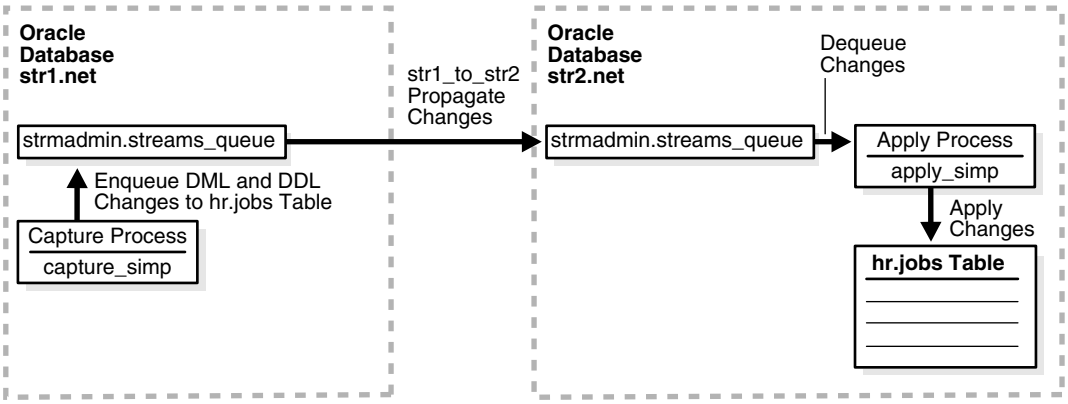
Note: The extended example is not included in the PDF version of this chapter, but it is included in the HTML version of the chapter.

Overview of the Simple Single-Source Replication Example

The example in this chapter illustrates using Streams to replicate data in one table between two databases. A capture process captures data manipulation language (DML) and data definition language (DDL) changes made to the `jobs` table in the `hr` schema at the `str1.net` Oracle database, and a propagation propagates these changes to the `str2.net` Oracle database. Next, an apply process applies these changes at the `str2.net` database. This example assumes that the `hr.jobs` table is read-only at the `str2.net` database.

Figure 14–1 provides an overview of the environment.

Figure 14–1 Simple Example that Shares Data from a Single-Source Database



Prerequisites

The following prerequisites must be completed before you begin the example in this chapter.

- Set the following initialization parameters to the values indicated:
  - GLOBAL_NAMES: This parameter must be set to true at each database that is participating in your Streams environment.
  - JOB_QUEUE_PROCESSES: This parameter must be set to at least 2 at each database that is propagating messages in your Streams environment. It should be set to the same value as the maximum number of jobs that can run simultaneously plus one. In this example, str1.net propagates messages. So, JOB_QUEUE_PROCESSES must be set to at least 2 at str1.net.
  - COMPATIBLE: This parameter must be set to 10.2.0 or higher at each database that is participating in your Streams environment.
  - STREAMS_POOL_SIZE: Optionally set this parameter to an appropriate value for each database in the environment. This parameter specifies the size of the Streams pool. The Streams pool stores messages in a buffered queue and is used for internal communications during parallel capture and apply. When the SGA_TARGET initialization parameter is set to a nonzero value, the Streams pool size is managed by Automatic Shared Memory Management.

    See Also: *Oracle Streams Concepts and Administration* for information about other initialization parameters that are important in a Streams environment

- Any database producing changes that will be captured must be running in ARCHIVELOG mode. In this example, changes are produced at str1.net, and so str1.net must be running in ARCHIVELOG mode.

  See Also: *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode

- Configure your network and Oracle Net so that the str1.net database can communicate with the str2.net database.

  See Also: *Oracle Database Net Services Administrator's Guide*

- This example creates a new user to function as the Streams administrator (strmadmin) at each database and prompts you for the tablespace you want to use for this user's data. Before you start this example, either create a new tablespace or identify an existing tablespace for the Streams administrator to use at each database. The Streams administrator should not use the SYSTEM tablespace.

15

Single-Source Heterogeneous Replication Example

This chapter illustrates an example of a single-source heterogeneous replication environment that can be constructed using Streams, as well as the tasks required to add new objects and databases to such an environment.

This chapter contains these topics:

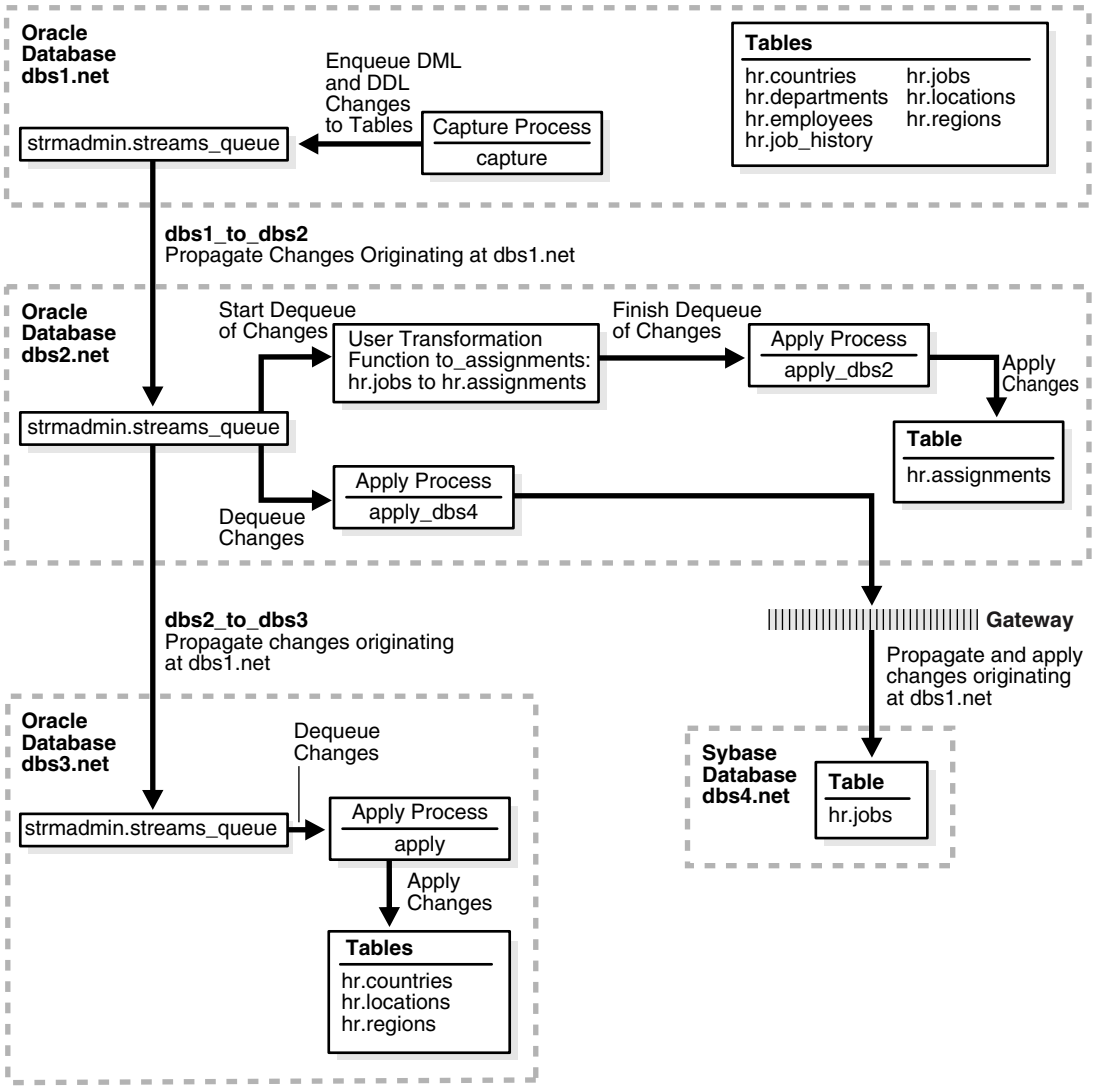
Note: The extended example is not included in the PDF version of this chapter, but it is included in the HTML version of the chapter.

Overview of the Single-Source Heterogeneous Replication Example

This example illustrates using Streams to replicate data between four databases. The environment is heterogeneous because three of the databases are Oracle databases and one is a Sybase database. DML and DDL changes made to tables in the `hr` schema at the `dbs1.net` Oracle database are captured and propagated to the other two Oracle databases. Only DML changes are captured and propagated to the `dbs4.net` database, because an apply process cannot apply DDL changes to a non-Oracle database. Changes to the `hr` schema occur only at `dbs1.net`. The `hr` schema is read-only at the other databases in the environment.

Figure 15–1 provides an overview of the environment.

Figure 15–1 Sample Environment that Shares Data from a Single Source Database



As illustrated in Figure 15–1, `dbs1.net` contains the following tables in the `hr` schema:

- `countries`
- `departments`
- `employees`
- `job_history`
- `jobs`
- `locations`
- `regions`

This example uses directed networks, which means that captured changes at a source database are propagated to another database through one or more intermediate databases. Here, the `dbs1.net` database propagates changes to the `dbs3.net` database through the intermediate database `dbs2.net`. This configuration is an

example of queue forwarding in a directed network. Also, the dbs1.net database propagates changes to the dbs2.net database, which applies the changes directly to the dbs4.net database through a gateway.

Some of the databases in the environment do not have certain tables. If the database is not an intermediate database for a table and the database does not contain the table, then changes to the table do not need to be propagated to that database. For example, the departments, employees, job_history, and jobs tables do not exist at dbs3.net. Therefore, dbs2.net does not propagate changes to these tables to dbs3.net.

In this example, Streams is used to perform the following series of actions:

1. The capture process captures DML and DDL changes for all of the tables in the hr schema and enqueues them into a queue at the dbs1.net database. In this example, changes to only four of the seven tables are propagated to destination databases, but in the example that illustrates "Add Objects to an Existing Streams Replication Environment" on page 15-5, the remaining tables in the hr schema are added to a destination database.
2. The dbs1.net database propagates these changes in the form of messages to a queue at dbs2.net.
3. At dbs2.net, DML changes to the jobs table are transformed into DML changes for the assignments table (which is a direct mapping of jobs) and then applied. Changes to other tables in the hr schema are not applied at dbs2.net.
4. Because the queue at dbs3.net receives changes from the queue at dbs2.net that originated in countries, locations, and regions tables at dbs1.net, these changes are propagated from dbs2.net to dbs3.net. This configuration is an example of directed networks.
5. The apply process at dbs3.net applies changes to the countries, locations, and regions tables.
6. Because dbs4.net, a Sybase database, receives changes from the queue at dbs2.net to the jobs table that originated at dbs1.net, these changes are applied remotely from dbs2.net using the dbs4.net database link through a gateway. This configuration is an example of heterogeneous support.

Prerequisites

The following prerequisites must be completed before you begin the example in this chapter.

- Set the following initialization parameters to the values indicated for all databases in the environment:
  - GLOBAL_NAMES: This parameter must be set to true at each database that is participating in your Streams environment.
  - JOB_QUEUE_PROCESSES: This parameter must be set to at least 2 at each database that is propagating messages in your Streams environment. It should be set to the same value as the maximum number of jobs that can run simultaneously plus one. In this example, dbs1.net and dbs2.net propagate messages. So, JOB_QUEUE_PROCESSES must be set to at least 2 at these databases.

  - COMPATIBLE: This parameter must be set to 10.2.0 or higher.
  - STREAMS_POOL_SIZE: Optionally set this parameter to an appropriate value for each database in the environment. This parameter specifies the size of the Streams pool. The Streams pool stores messages in a buffered queue and is used for internal communications during parallel capture and apply. When the SGA_TARGET initialization parameter is set to a nonzero value, the Streams pool size is managed by Automatic Shared Memory Management.

    See Also: *Oracle Streams Concepts and Administration* for information about other initialization parameters that are important in a Streams environment

- Any database producing changes that will be captured must be running in ARCHIVELOG mode. In this example, changes are produced at dbs1.net, and so dbs1.net must be running in ARCHIVELOG mode.

  See Also: *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode

- Configure an Oracle gateway on dbs2.net to communicate with the Sybase database dbs4.net.

  See Also: *Oracle Database Heterogeneous Connectivity Administrator's Guide*

- At the Sybase database dbs4.net, set up the hr user.

  See Also: Your Sybase documentation for information about creating users and tables in your Sybase database

- Instantiate the hr.jobs table from the dbs1.net Oracle database at the dbs4.net Sybase database.

  See Also: "Instantiation in an Oracle to Non-Oracle Environment" on page 5-5

- Configure your network and Oracle Net so that the following databases can communicate with each other:
  - dbs1.net and dbs2.net
  - dbs2.net and dbs3.net
  - dbs2.net and dbs4.net
  - dbs3.net and dbs1.net (for optional Data Pump network instantiation)

    See Also: *Oracle Database Net Services Administrator's Guide*

- This examples creates a new user to function as the Streams administrator (strmadmin) at each database and prompts you for the tablespace you want to use for this user's data. Before you start this example, either create a new tablespace or identify an existing tablespace for the Streams administrator to use at each database. The Streams administrator should not use the SYSTEM tablespace.

Add Objects to an Existing Streams Replication Environment

This example extends the Streams environment configured in the previous sections by adding replicated objects to an existing database. To complete this example, you must have completed the tasks in one of the previous examples in this chapter.

This example will add the following tables to the `hr` schema in the `dbs3.net` database:

- `departments`
- `employees`
- `job_history`
- `jobs`

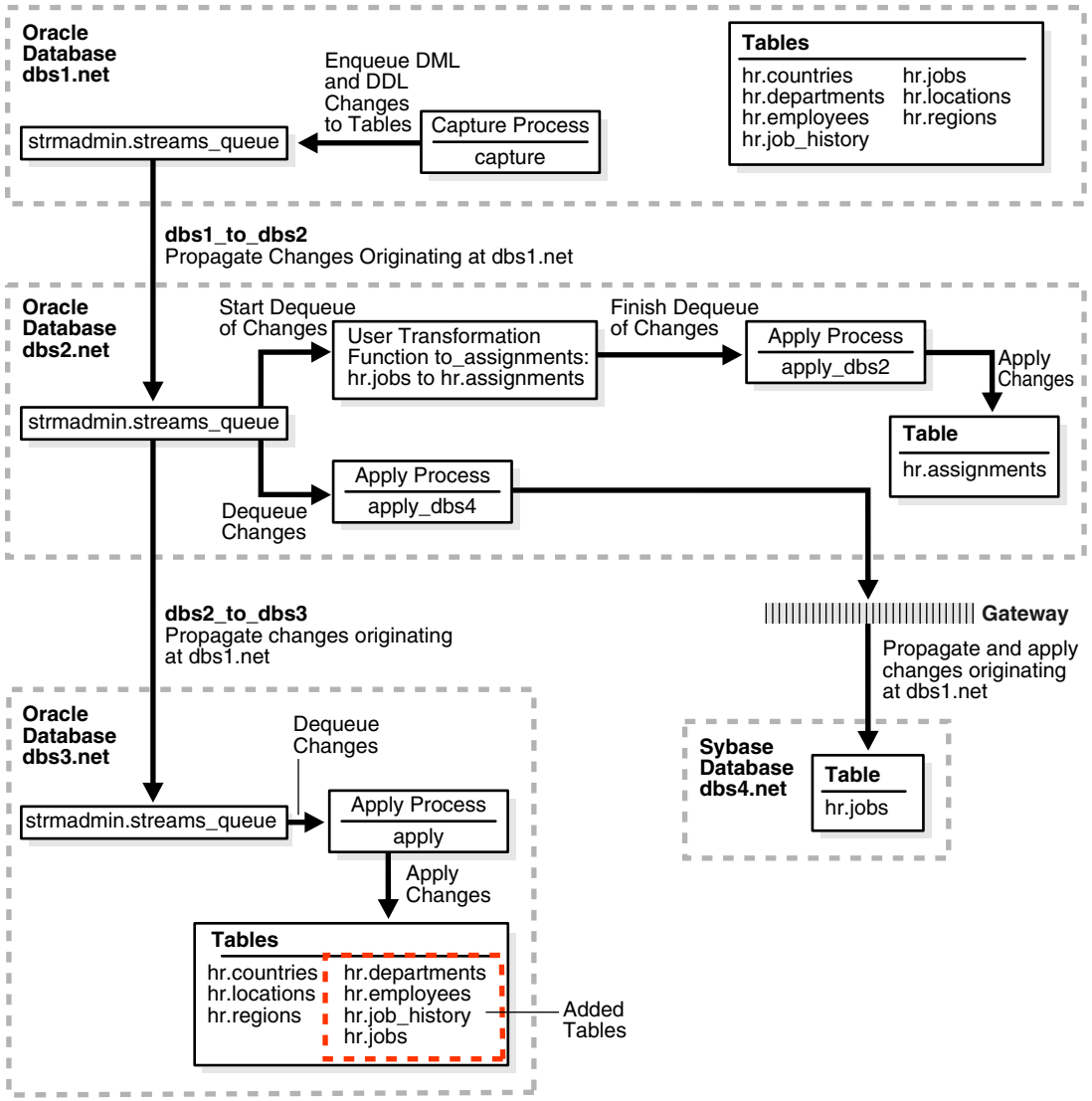
When you complete this example, Streams processes changes to these tables with the following series of actions:

1. The capture process captures changes at `dbs1.net` and enqueues them at `dbs1.net`.
2. A propagation propagates changes from the queue at `dbs1.net` to the queue at `dbs2.net`.
3. A propagation propagates changes from the queue at `dbs2.net` to the queue at `dbs3.net`.
4. The apply process at `dbs3.net` applies the changes at `dbs3.net`.

When you complete this example, the `hr` schema at the `dbs3.net` database will have all of its original tables, because the `countries`, `locations`, and `regions` tables were instantiated at `dbs3.net` in the previous section.

Figure 15–2 provides an overview of the environment with the added tables.

Figure 15–2 Adding Objects to dbs3.net in the Environment

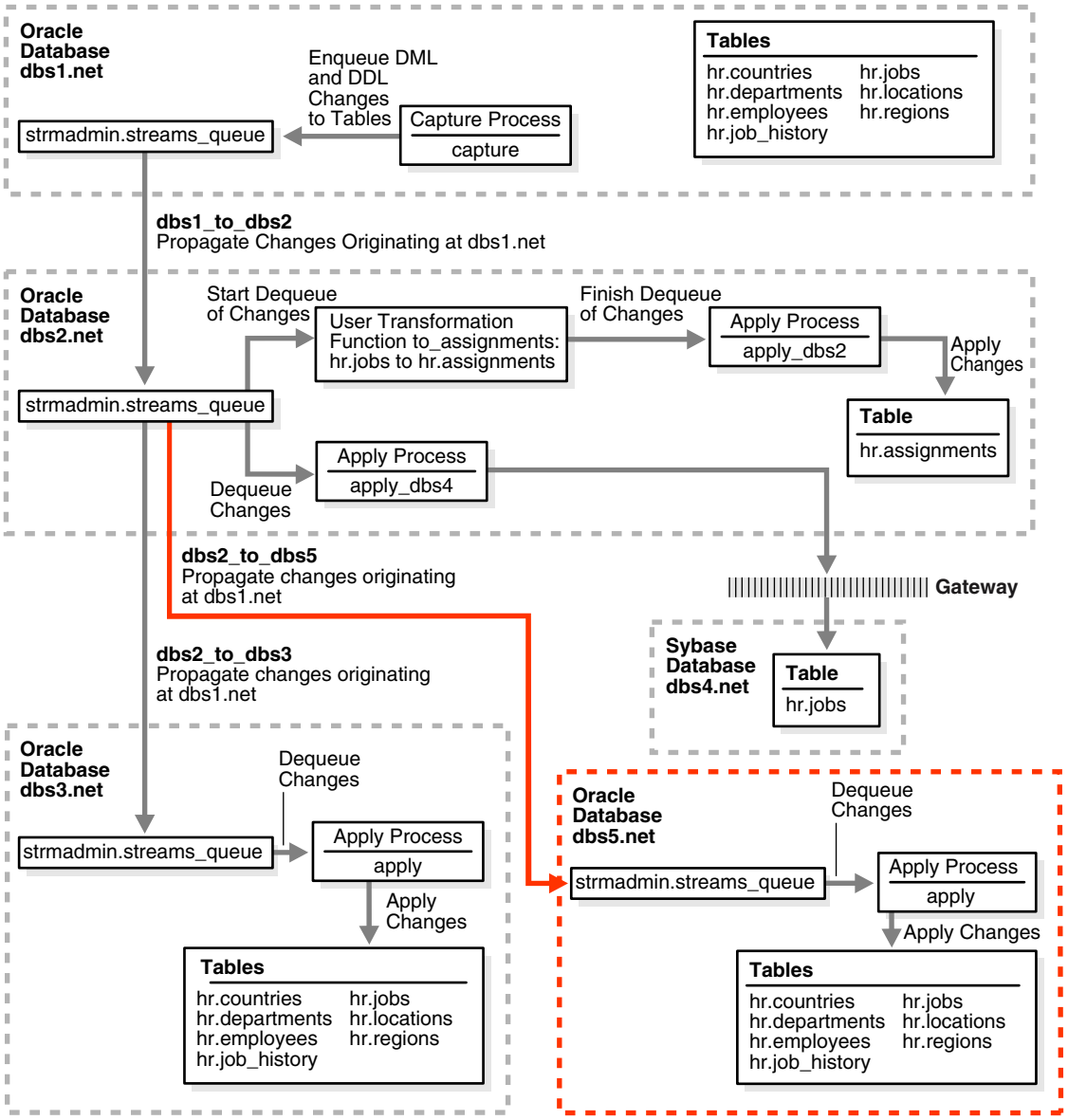


Add a Database to an Existing Streams Replication Environment

This example extends the Streams environment configured in the previous sections by adding an additional database to the existing configuration. In this example, an existing Oracle database named `dbs5.net` is added to receive changes to the entire `hr` schema from the queue at `dbs2.net`.

Figure 15–3 provides an overview of the environment with the added database.

Figure 15–3 Adding the dbs5.net Oracle Database to the Environment



To complete this example, you must meet the following prerequisites:

- The `dbs5.net` database must exist.
- The `dbs2.net` and `dbs5.net` databases must be able to communicate with each other through Oracle Net.
- The `dbs5.net` and `dbs1.net` databases must be able to communicate with each other through Oracle Net (for optional Data Pump network instantiation).
- You must have completed the tasks in the previous examples in this chapter.
- The "Prerequisites" on page 15-3 must be met if you want the entire Streams environment to work properly.
- This examples creates a new user to function as the Streams administrator (`strmadmin`) at the `dbs5.net` database and prompts you for the tablespace you want to use for this user's data. Before you start this example, either create a new tablespace or identify an existing tablespace for the Streams administrator to use at the `dbs5.net` database. The Streams administrator should not use the `SYSTEM` tablespace.

16

Multiple-Source Replication Example

This chapter illustrates an example of a multiple-source replication environment that can be constructed using Streams.

This chapter contains these topics:

- Overview of the Multiple Source Databases Example
- Prerequisites

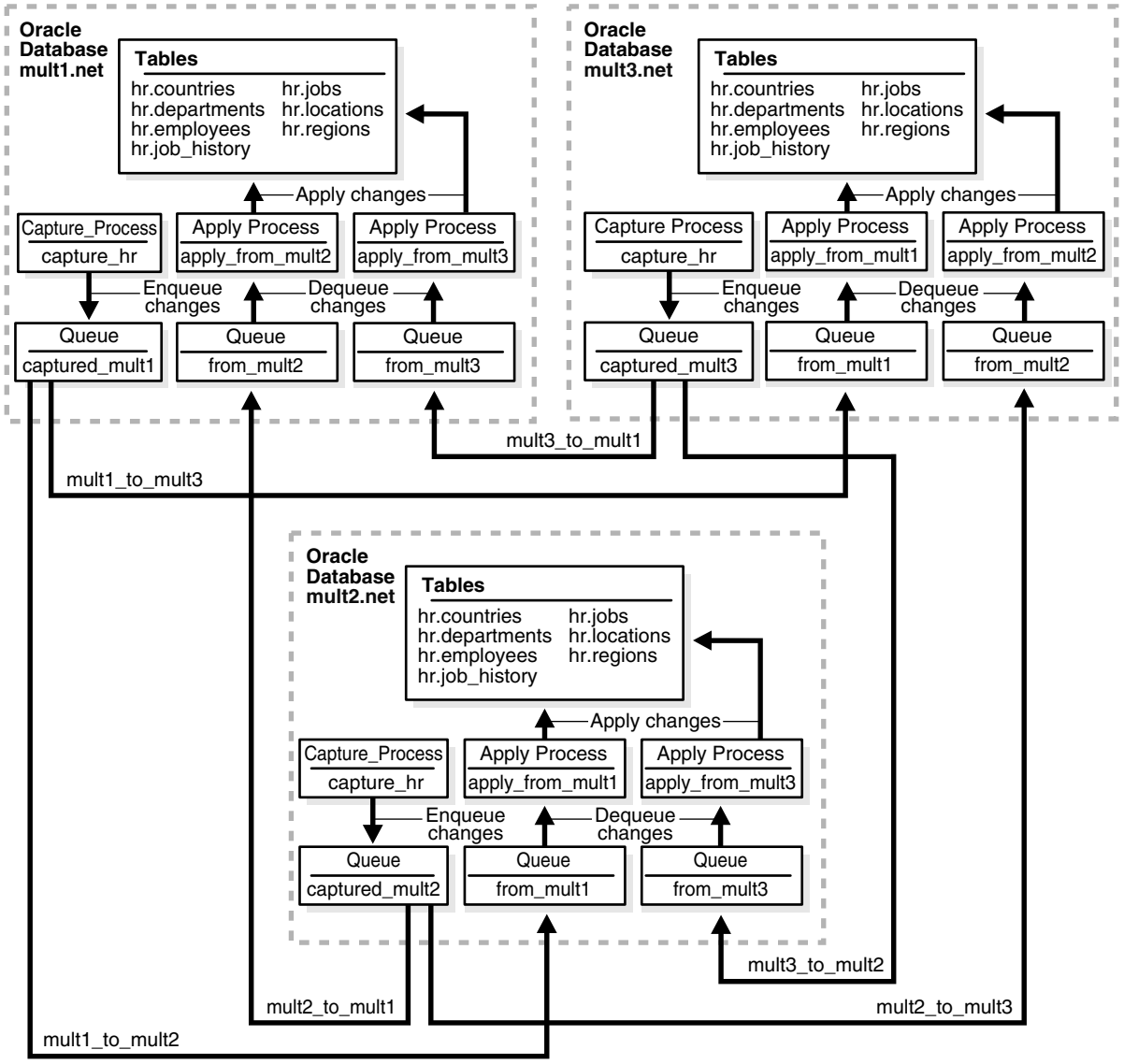
> **Note:** The extended example is not included in the PDF version of this chapter, but it is included in the HTML version of the chapter.

Overview of the Multiple Source Databases Example

This example illustrates using Streams to replicate data for a schema among three Oracle databases. DML and DDL changes made to tables in the `hr` schema are captured at all databases in the environment and propagated to each of the other databases in the environment.

Figure 16–1 provides an overview of the environment.

Figure 16–1 Sample Environment that Shares Data from Multiple Databases



As illustrated in Figure 16–1, all of the databases will contain the hr schema when the example is complete. However, at the beginning of the example, the hr schema exists only at mult1.net. During the example, you instantiate the hr schema at mult2.net and mult3.net.

In this example, Streams is used to perform the following series of actions:

1. After instantiation, the capture process at each database captures DML and DDL changes for all of the tables in the hr schema and enqueues them into a local queue.
2. Each database propagates these changes to all of the other databases in the environment.
3. The apply process at each database applies changes in the hr schema received from the other databases in the environment.

This example uses only one queue for each database, but you can use multiple queues for each database if you want to separate changes from different source databases. In addition, this example avoids sending changes back to their source database by using the default apply tag for the apply processes. When you create an apply process, the changes applied by the apply process have redo entries with a tag of '00' (double zero) by default. These changes are not recaptured because, by default, rules created by the DBMS_STREAMS_ADM package have an is_null_tag()='Y' condition by default, and this condition ensures that each capture process captures a change in a redo entry only if the tag for the redo entry is NULL.

> **See Also:** Chapter 4, "Streams Tags" for more information about tags

Prerequisites

The following prerequisites must be completed before you begin the example in this chapter.

- Set the following initialization parameters to the values indicated at each database in the Streams environment:
  - GLOBAL_NAMES: This parameter must be set to true. Make sure the global names of the databases are mult1.net, mult2.net, and mult3.net.
  - JOB_QUEUE_PROCESSES: This parameter must be set to at least 2 because each database propagates messages. It should be set to the same value as the maximum number of jobs that can run simultaneously plus one.
  - COMPATIBLE: This parameter must be set to 10.2.0 or higher.
  - Make sure the PROCESSES and SESSIONS initialization parameters are set high enough for all of the Streams clients used in this example. This example configures one capture process, two propagations, and two apply processes at each database.
  - STREAMS_POOL_SIZE: Optionally set this parameter to an appropriate value for each database in the environment. This parameter specifies the size of the Streams pool. The Streams pool stores messages in a buffered queue and is used for internal communications during parallel capture and apply. When the SGA_TARGET initialization parameter is set to a nonzero value, the Streams pool size is managed by Automatic Shared Memory Management.

Attention: You might need to modify other initialization parameter settings for this example to run properly.

See Also: *Oracle Streams Concepts and Administration* for information about other initialization parameters that are important in a Streams environment

- Any database producing changes that will be captured must be running in `ARCHIVELOG` mode. In this example, all databases are capturing changes, and so all databases must be running in `ARCHIVELOG` mode.

  See Also: *Oracle Database Administrator's Guide* for information about running a database in `ARCHIVELOG` mode

- Configure your network and Oracle Net so that all three databases can communicate with each other.

  See Also: *Oracle Database Net Services Administrator's Guide*

- This examples creates a new user to function as the Streams administrator (`strmadmin`) at each database and prompts you for the tablespace you want to use for this user's data. Before you start this example, either create a new tablespace or identify an existing tablespace for the Streams administrator to use at each database. The Streams administrator should not use the `SYSTEM` tablespace.

Part V

Appendixes

This part includes the following appendix:

- Appendix A, "Migrating Advanced Replication to Streams"

A

Migrating Advanced Replication to Streams

Database administrators who have been using Advanced Replication to maintain replicated database objects at different sites can migrate their Advanced Replication environment to a Streams environment. This chapter provides a conceptual overview of the steps in this process and documents each step with procedures and examples.

This chapter contains these topics:

- Overview of the Migration Process
- Preparing to Generate the Migration Script
- Generating and Modifying the Migration Script
- Performing the Migration for Advanced Replication to Streams
- Recreating Master Sites to Retain Materialized View Groups

Note: The example of a generated migration script is not included in the PDF version of this chapter, but it is included in the HTML version of the chapter.

See Also: *Oracle Database Advanced Replication* and *Oracle Database Advanced Replication Management API Reference* for more information about Advanced Replication

Overview of the Migration Process

The following sections provide a conceptual overview of the migration process:

- Migration Script Generation and Use
- Modification of the Migration Script
- Actions Performed by the Generated Script
- Migration Script Errors
- Manual Migration of Updatable Materialized Views
- Advanced Replication Elements that Cannot Be Migrated to Streams

Migration Script Generation and Use

You can use the procedure DBMS_REPCAT.STREAMS_MIGRATION to generate a SQL*Plus script that migrates an existing Advanced Replication environment to a Streams environment. When you run the DBMS_REPCAT.STREAMS_MIGRATION procedure at a master definition site in a multimaster replication environment, it generates a SQL*Plus script in a file at a location that you specify. Once the script is generated, you run it at each master site in your Advanced Replication environment to set up a Streams environment for each master site. To successfully generate the Streams environment for your replication groups, the replication groups for which you run the script must have exactly the same master sites. If replication groups have different master sites, then you can generate multiple scripts to migrate each replication group to Streams.

At times, you must stop, or quiesce, all replication activity for a replication group so that you can perform certain administrative tasks. You do not need to quiesce the replication groups when you run the DBMS_REPCAT.STREAMS_MIGRATION procedure. However, you must quiesce the replication groups being migrated to Streams when you run the generated script at the master sites. Because you have queisced the replication groups to run the script at the master sites, you do not have to stop any existing capture processes, propagation jobs, or apply processes at these sites.

Modification of the Migration Script

The generated migration script uses comments to indicate Advanced Replication elements that cannot be converted to Streams. It also provides suggestions for modifying the script to convert these elements to Streams. You can use these suggestions to edit the script before you run it. You can also customize the migration script in other ways to meet your needs.

The script sets all parameters when it runs PL/SQL procedures and functions. When you generate the script, it sets default values for parameters that typically do not need to be changed. However, you can change these default parameters by editing the script if necessary. The parameters with default settings include the following:

- include_dml
- include_ddl
- include_tagged_lcr

The beginning of the script has a list of variables for names that are used by the procedures and functions in the script. When you generate the script, it sets these variables to default values that you should not need to change. However, you can change the default settings for these variables if necessary. The variables specify names of queues, capture processes, propagations, and apply processes.

Actions Performed by the Generated Script

The migration script performs the following actions:

- Prints warnings in comments if the replication groups contain features that cannot be converted to Streams.
- Creates ANYDATA queues, if needed, using the DBMS_STREAMS_ADM.SET_UP_QUEUE procedure.
- Configures propagation between all master sites using the DBMS_STREAMS_ADMIN.ADD_TABLE_PROPAGATION_RULES procedure for each table.

- Configures capture at each master site using the `DBMS_STREAMS_ADMIN.ADD_TABLE_RULES` procedure for each table.
- Configures apply for changes from all the other master sites using the `DBMS_STREAMS_ADMIN.ADD_TABLE_RULES` procedure for each table.
- Sets the instantiation SCN for each replicated object at each site where changes to the object are applied.
- Creates the necessary supplemental log groups at source databases.
- Sets key columns, if any.
- Configures conflict resolution if it was configured for the Advanced Replication environment being migrated.

Migration Script Errors

If Oracle encounters an error while running the migration script, then the migration script exits immediately. If this happens, then you must modify the script to run any commands that have not already been executed successfully.

Manual Migration of Updatable Materialized Views

You cannot migrate updatable materialized views using the migration script. You must migrate updatable materialized views from an Advanced Replication environment to a Streams environment manually.

> **See Also:** "Recreating Master Sites to Retain Materialized View Groups" on page A-12

Advanced Replication Elements that Cannot Be Migrated to Streams

Streams does not support the following:

- Replication of changes to tables with columns of the following datatypes: `BFILE`, `ROWID`, and user-defined types (including object types, `REF`s, varrays, and nested tables)
- Synchronous replication

If your current Advanced Replication environment uses these features, then these elements of the environment cannot be migrated to Streams. In this case, you might decide not to migrate the environment to Streams at this time, or you might decide to modify the environment so that it can be migrated to Streams.

Preparing to Generate the Migration Script

Before generating the migration script, make sure all the following conditions are met:

- All the replication groups must have the same master site(s).
- The master site that generates the migration script must be running Oracle Database 10*g*.
- The other master sites that run the script, but do not generate the script, must be running Oracle9*i* Database Release 2 (9.2) or later.

Generating and Modifying the Migration Script

To generate the migration script, use the procedure `DBMS_REPCAT.STREAMS_MIGRATION` in the `DBMS_REPCAT` package. The syntax for this procedure is as follows:

```
DBMS_REPCAT.STREAMS_MIGRATION (
     gnames              IN   DBMS_UTILITY.NAME_ARRAY,
     file_location       IN   VARCHAR2,
     filename            IN   VARCHAR2);
```

Parameters for the `DBMS_REPCAT.STREAMS_MIGRATION` procedure include the following:

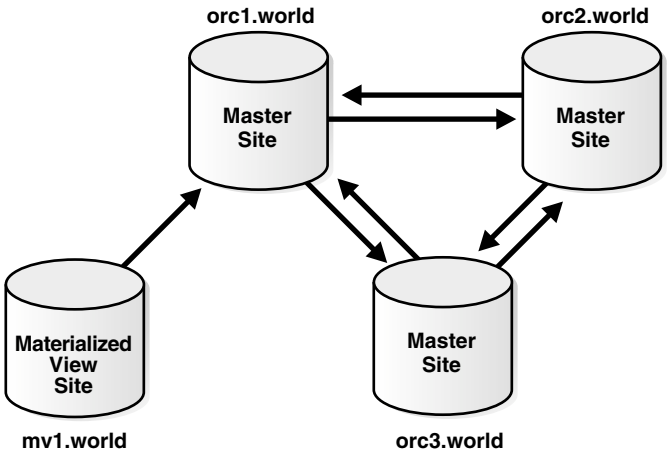
- `gnames`: List of replication groups to migrate to Streams. The replication groups listed must all contain exactly the same master sites. An error is raised if the replication groups have different masters.
- `file_location`: Directory location of the migration script.
- `filename`: Name of the migration script.

This procedure generates a script for setting up a Streams environment for the given replication groups. The script can be customized and run at each master site.

Example Advanced Replication Environment to be Migrated to Streams

Figure A–1 shows the Advanced Replication environment that will be migrated to Streams in this example.

Figure A–1 Advanced Replication Environment to be Migrated to Streams



This Advanced Replication environment has the following characteristics:

- The `orc1.world` database is the master definition site for a three-way master configuration that also includes `orc2.world` and `orc3.world`.
- The `orc1.world` database is the master site for the `mv1.world` materialized view site.
- The environment replicates changes to the database objects in the `hr` schema between the three master sites and between the master site and the materialized view site. A single replication group named `hr_repg` contains the replicated objects.

- Conflict resolution is configured for the `hr.countries` table in the multimaster environment. The latest timestamp conflict resolution method resolves conflicts on this table.
- The materialized views at the `mv1.world` site are updatable.

You can configure this Advanced Replication environment by completing the tasks described in the following sections of the *Oracle Database Advanced Replication Management API Reference*:

- Set up the three master sites.
- Set up the materialized view sites (to set up `mv1.world` only).
- Create the `hr_repg` master group for the three master sites with `orc1.world` as the master definition site.
- Configure timestamp conflict resolution for the `hr.countries` table.
- Create the materialized view group at `mv1.world` based on the `hr_repg` master group at `orc1.world`.

To generate the migration script for this Advanced Replication environment, complete the following steps:

1. Create the Streams administrator at all master sites.
2. Make a directory location accessible.
3. Generate the migration script.
4. Verify the generated migration script creation and modify script.

Step 1 Create the Streams administrator at all master sites.

Complete the following steps to create the Streams administrator at each master site for the replication groups being migrated to Streams. For the sample environment described in "Example Advanced Replication Environment to be Migrated to Streams" on page A-4, complete these steps at `orc1.world`, `orc2.world`, and `orc3.world`:

1. Connect as an administrative user who can create users, grant privileges, and create tablespaces.
2. Either create a tablespace for the Streams administrator or use an existing tablespace. For example, the following statement creates a new tablespace for the Streams administrator:

```
CREATE TABLESPACE streams_tbs DATAFILE '/usr/oracle/dbs/streams_tbs.dbf'
  SIZE 25 M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

3. Create a new user to act as the Streams administrator or use an existing user. For example, to create a new user named `strmadmin` and specify that this user uses the `streams_tbs` tablespace, run the following statement:

```
CREATE USER strmadmin
 IDENTIFIED BY strmadminpw
   DEFAULT TABLESPACE streams_tbs
   QUOTA UNLIMITED ON streams_tbs;

GRANT DBA TO strmadmin;
```

Note:

- For security purposes, use a password other than `strmadminpw` for the Streams administrator.
- The migration script assumes that the username of the Streams administrator is `strmadmin`. If your Streams administrator has a different username, then edit the migration script to replace all instances of `strmadmin` with the username of your Streams administrator.
- Make sure you grant `DBA` role to the Streams administrator.

4. Grant any additional privileges required by the Streams administrator at each master site. The necessary privileges depend on your specific Streams environment.

   See Also: *Oracle Streams Concepts and Administration* for information about addition privileges that might be required for a Streams administrator

Step 2 Make a directory location accessible.

The directory specified by the `file_location` parameter in the `DBMS_REPCAT.STREAMS_MIGRATION` procedure must be accessible to PL/SQL. If you do not have directory object that is accessible to the Streams administrator at the master definition site currently, then connect as the Streams administrator, and create a directory object using the SQL statement `CREATE DIRECTORY`.

A directory object is similar to an alias for the directory. For example, to create a directory object called `MIG2STR_DIR` for the `/usr/scripts` directory on your computer system, run the following procedure:

```
CONNECT strmadmin/strmadminpw@orc1.world

CREATE DIRECTORY MIG2STR_DIR AS '/usr/scripts';
```

See Also: *Oracle Database SQL Reference* for more information about the `CREATE DIRECTORY` statement

Step 3 Generate the migration script.

To generate the migration script, run the `DBMS_REPCAT.STREAMS_MIGRATION` procedure at the master definition site and specify the appropriate parameters. For example, the following procedure generates a script that migrates an Advanced Replication environment with one replication group named `hr_repg`. The script name is `rep2streams.sql`, and it is generated into the `/usr/scripts` directory on the local computer system. This directory is represented by the directory object `MIG2STR_DIR`.

```
CONNECT strmadmin/strmadminpw@orc1.world
```

```
DECLARE
  rep_groups DBMS_UTILITY.NAME_ARRAY;
  BEGIN
    rep_groups(1) := 'HR_REPG';
    DBMS_REPCAT.STREAMS_MIGRATION(
      gnames         =>  rep_groups,
      file_location  =>  'MIG2STR_DIR',
      filename       =>  'rep2streams.sql');
END;
/
```

Step 4 Verify the generated migration script creation and modify script.

After generating the migration script, verify that the script was created viewing the script in the specified directory. If necessary, you can modify it to support the following:

- If your environment requires conflict resolution that used the additive, average, priority group, or site priority Advanced Replication conflict resolution methods, then configure user-defined conflict resolution methods to resolve conflicts. Streams does not provide prebuilt conflict resolution methods that are equivalent to these methods.

  However, the migration script supports the following conflict resolution methods automatically: overwrite, discard, maximum, and minimum. The script converts an earliest timestamp method to a minimum method automatically, and it converts a latest timestamp method to a maximum method automatically. If you use a timestamp conflict resolution method, then the script assumes that any triggers necessary to populate the timestamp column in a table already exist.
- Unique conflict resolution.
- Delete conflict resolution.
- Multiple conflict resolution methods to be executed in a specified order when a conflict occurs. Streams allows only one conflict resolution method to be specified for each column list.
- Queue-to-queue propagations. By default, the script creates queue-to-dblink propagations.
- Procedural replication.
- Replication of data definition language (DDL) changes for nontable objects, including the following:
  - Functions
  - Indexes
  - Indextypes
  - Operators
  - Packages
  - Package bodies
  - Procedures
  - Synonyms
  - Triggers
  - Types

- Type bodies
- Views

Because changes to these objects were being replicated by Advanced Replication at all sites, the migration script does not need to take any action to migrate these objects. You can add DDL rules to the Streams environment to support the future modification and creation of these types of objects.

For example, to specify that a capture process named `streams_capture` at the `orc1.world` database captures DDL changes to all of the database objects in the `hr` schema, add the following to the script:

```
BEGIN
  DBMS_STREAMS_ADM.ADD_SCHEMA_RULES(
    schema_name        => 'hr',
    streams_type       => 'capture',
    streams_name       => 'streams_capture',
    queue_name         => 'strmadmin.streams_queue',
    include_dml        => false,
    include_ddl        => true,
    include_tagged_lcr => false,
    source_database    => 'orc1.world');
END;
/
```

Notice that the `include_ddl` parameter is set to `true`. By setting this parameter to `true`, this procedure adds a schema rule for DDL changes to the `hr` schema to the rule set for the capture process. This rule instructs the capture process to capture DDL changes to the `hr` schema and its objects. For the DDL changes to be replicated, you must add similar rules to the appropriate propagations and apply processes.

See Also:

- Chapter 3, "Streams Conflict Resolution"
- ***Oracle Streams Concepts and Administration*** for information about queue-to-queue propagations

Performing the Migration for Advanced Replication to Streams

This section explains how to perform the migration from an Advanced Replication environment to a Streams environment.

This section contains the following topics:

- Before Executing the Migration Script
- Executing the Migration Script
- After Executing the Script

Before Executing the Migration Script

Complete the following steps before executing the migration script:

1. Set initialization parameters that are relevant to Streams.
2. Enable archive logging at all sites.
3. Create up database links.
4. Quiesce each replication group that you are migrating to Streams.

Step 1 Set initialization parameters that are relevant to Streams.

At each replication database, set initialization parameters that are relevant to Streams and restart the database if necessary.

> **See Also:** *Oracle Streams Concepts and Administration* for information about initialization parameters that are important to Streams

Step 2 Enable archive logging at all sites.

Make sure each master site is running in `ARCHIVELOG` mode, because a capture process requires `ARCHIVELOG` mode. In the sample environment, `orc1.world`, `orc2.world`, and `orc3.world` must be running in `ARCHIVELOG` mode. You can check the log mode for a database by querying the `LOG_MODE` column in the `V$DATABASE` dynamic performance view.

> **See Also:** *Oracle Database Administrator's Guide* for information about running a database in `ARCHIVELOG` mode

Step 3 Create up database links.

Create a database link from the Streams administrator at each master site to the Streams administrator at the other master sites. For the sample environment described in "Example Advanced Replication Environment to be Migrated to Streams" on page A-4, create the following database links:

```
CONNECT strmadmin/strmadminpw@orc1.world

CREATE DATABASE LINK orc2.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc2.world';

CREATE DATABASE LINK orc3.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc3.world';


CONNECT strmadmin/strmadminpw@orc2.world

CREATE DATABASE LINK orc1.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc1.world';

CREATE DATABASE LINK orc3.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc3.world';


CONNECT strmadmin/strmadminpw@orc3.world

CREATE DATABASE LINK orc1.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc1.world';

CREATE DATABASE LINK orc2.world CONNECT TO strmadmin
   IDENTIFIED BY strmadminpw USING 'orc2.world';
```

Step 4 Quiesce each replication group that you are migrating to Streams.

Run the `DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY` procedure at the master definition site for each replication group that you are migrating to Streams.

In the sample environment, `orc1.world` is the master definition site, and `hr_repg` is the replication group being migrated to Streams. So, connect to `orc1.world` as the replication administrator and run the `SUSPEND_MASTER_ACTIVITY` procedure:

```
CONNECT repadmin/repadmin@orc1.world

BEGIN
   DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY (
      gname => 'hr_repg');
END;
/
```

Do not proceed until the master group is quiesced. You can check the status of a master group by querying the `STATUS` column in the `DBA_REPGROUP` data dictionary view.

Executing the Migration Script

Perform the following steps to migrate:

1. Connect as the Streams administrator and run the script at each site.
2. Verify that Streams configuration completed successfully at all sites.

Step 1 Connect as the Streams administrator and run the script at each site.

In the sample environment, connect in SQL*Plus as the Streams administrator `strmadmin` in SQL*Plus at `orc1.world`, `orc2.world`, and `orc3.world` and execute the migration script `rep2streams.sql`:

```
CONNECT strmadmin/strmadminpw@orc1.world
SET ECHO ON
SPOOL rep2streams.out
@rep2streams.sql

CONNECT strmadmin/strmadminpw@orc2.world
SET ECHO ON
SPOOL rep2streams.out
@rep2streams.sql

CONNECT strmadmin/strmadminpw@orc3.world
SET ECHO ON
SPOOL rep2streams.out
@rep2streams.sql
```

Step 2 Verify that Streams configuration completed successfully at all sites.

Check the spool file at each site to make sure there are no errors. If there are errors, then you should modify the script to execute the steps that were not completed successfully, and then rerun the script. In the sample environment, the spool file is `rep2streams.out` at each master site.

After Executing the Script

Perform the following steps to complete the migration process:

1. Drop replication groups you migrated at each site.
2. Start the apply processes at each site.
3. Start the capture process at each site.

Step 1 Drop replication groups you migrated at each site.

To drop a replication group that you successfully migrated to Streams, connect as the replication administrator to the master definition site, and run the DBMS_REPCAT.DROP_MASTER_REPGROUP procedure.

> **Attention:** Make sure the drop_contents parameter is set to false in the DROP_MASTER_REPGROUP procedure. If it is set to true, then the replicated database objects are dropped.

```
CONNECT repadmin/repadmin@orc1.world

BEGIN
   DBMS_REPCAT.DROP_MASTER_REPGROUP (
     gname         => 'hr_repg',
     drop_contents => false,
     all_sites     => true);
END;
/
```

To make sure the migrated replication groups are dropped at each database, query the GNAME column in the DBA_REPGROUP data dictionary view. The migrated replication groups should not appear in the query output at any database.

If you no longer need the replication administrator, then you can drop this user also.

> **Caution:** Do not resume any Advanced Replication activity once Streams is set up.

Step 2 Start the apply processes at each site.

You can view the names of the apply processes at each site by running the following query while connected as the Streams administrator:

```
SELECT APPLY_NAME FROM DBA_APPLY;
```

When you know the names of the apply processes, you can start each one by running the START_APPLY procedure in the DBMS_APPLY_ADM package while connected as the Streams administrator. For example, the following procedure starts an apply process named apply_from_orc2 at orc1.world:

```
CONNECT strmadmin/strmadminpw@orc1.world

BEGIN
  DBMS_APPLY_ADM.START_APPLY(
    apply_name => 'apply_from_orc2');
END;
/
```

Make sure you start each apply process at every database in the new Streams environment.

Step 3 Start the capture process at each site.

You can view the name of the capture process at each site by running the following query while connected as the Streams administrator:

```
SELECT CAPTURE_NAME FROM DBA_CAPTURE;
```

When you know the name of the capture process, you can start each one by running the START_CAPTURE procedure in the DBMS_CAPTURE_ADM package while connected as the Streams administrator. For example, the following procedure starts a capture process named streams_capture at orc1.world:

```
CONNECT strmadmin/strmadminpw@orc1.world

BEGIN
  DBMS_CAPTURE_ADM.START_CAPTURE(
    capture_name => 'streams_capture');
END;
/
```

Make sure you start each capture process at every database in the new Streams environment.

Recreating Master Sites to Retain Materialized View Groups

If one or more materialized view groups used a master group that you migrated to Streams, then you must re-create the master group to retain these materialized view groups. Therefore, each database acting as the master site for a materialized view group must become the master definition site for a one-master configuration of a replication group that contains the tables used by the materialized views in the materialized view group.

Use the replication management APIs to create a replication group similar to the original replication group that was migrated to Streams. That is, the new replication group should have the same replication group name, objects, conflict resolution methods, and key columns. To retain the existing materialized view groups, you must re-create each master group at each master site that contained a master group for a materialized view group, re-create the master replication objects in the master group, regenerate replication support for the master group, and resume replication activity for the master group.

For example, consider the following Advanced Replication environment:

- Two master sites, mdb1.net and mdb2.net, have the replication group rg1. The mdb1.net database is the master definition site, and the objects in the rg1 replication group are replicated between mdb1.net and mdb2.net.
- The rg1 replication group at mdb1.net is the master group to the mvg1 materialized view group at mv1.net.
- The rg1 replication group at mdb2.net is the master group to the mvg2 materialized view group at mv2.net.

If the rg1 replication group is migrated to Streams at both mdb1.net and mdb2.net, and you want to retain the materialized view groups mvg1 at mv1.net and mvg2 at mv2.net, then you need to re-create the rg1 replication group at mdb1.net and mdb2.net after the migration to Streams. You configure both mdb1.net and mdb2.net to be the master definition site for the rg1 replication group in a one-master environment.

It is not necessary to drop or re-create materialized view groups at the materialized view sites. As long as a new master replication group resembles the original replication group, the materialized view groups are not affected. Do not refresh these materialized view groups until generation of replication support for each master object is complete (Step 3 in the task in this section). Similarly, do not push the deferred transaction queue at any materialized view site with updatable materialized views until generation of replication support for each master object is complete.

For the sample environment described in "Example Advanced Replication Environment to be Migrated to Streams" on page A-4, only the `hr_repg` replication group at `orc1.world` was the master group to a materialized view group at `mv1.world`. To retain this materialized view group at `mv1.world`, complete the following steps while connected as the replication administrator:

1. Create the master group `hr_repg` at `orc1.world`.

```
CONNECT repadmin/repadmin@orc1.world

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPGROUP (
      gname => 'hr_repg');
END;
/
```

2. Add the tables in the hr schema to the `hr_repg` master group. These tables are master tables to the materialized views at `mv1.world`.

```
BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'countries',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'departments',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'employees',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'jobs',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
```

```
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'job_history',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'locations',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/

BEGIN
   DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
      gname               => 'hr_repg',
      type                => 'TABLE',
      oname               => 'regions',
      sname               => 'hr',
      use_existing_object => true,
      copy_rows           => false);
END;
/
```

3. Generate replication support for each object in the `hr_repg` master group.

```
BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'countries',
      type              => 'TABLE');
END;
/

BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'departments',
      type              => 'TABLE');
END;
/

BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'employees',
      type              => 'TABLE');
END;
/
```

```
BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'jobs',
      type              => 'TABLE');
END;
/

BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'job_history',
      type              => 'TABLE');
END;
/

BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'locations',
      type              => 'TABLE');
END;
/

BEGIN
    DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT (
      sname             => 'hr',
      oname             => 'regions',
      type              => 'TABLE');
END;
/
```

4. Resume master activity for the `hr_repg` master group.

```
BEGIN
   DBMS_REPCAT.RESUME_MASTER_ACTIVITY (
      gname => 'hr_repg');
END;
/
```

Note: A materialized view log should exist for each table you added to the `hr_repg` master group, unless you deleted these logs manually after you migrated the replication group to Streams. If these materialized view logs do not exist, then you must create them.

Index

A

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

V