Oracle® Database

XStream Guide

11*g* Release 2 (11.2)

E15874-01

September 2009

Oracle Database XStream Guide, 11*g* Release 2 (11.2)

E15874-01

Primary Author: Randy Urbano

Contributors: Alan Downing, Thuvan Hoang, Tianshu Li, Jing Liu, Edwina Lu, Pat McElroy, Valarie Moore, Ashish Ray, Jim Stamos, Lik Wong, Jingwei Wu, Jun Yuan

Contents

Preface

Oracle Database XStream Guide describes the features and functionality of XStream. This document contains conceptual information about XStream, along with information about configuring and managing an XStream environment. In addition, this document contains reference information related to XStream.

Audience

This guide is intended for database administrators who configure and manage XStream environments. To use this document, database administrators must be familiar with relational database concepts, SQL, distributed database administration, Oracle Streams concepts, PL/SQL, and the operating systems under which they run an XStream environment.

This guide is also intended for programmers who develop applications that use XStream. To use this document, programmers need knowledge of an application development language and relational database concepts.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see the following documents:

- *Oracle Streams Concepts and Administration*
- *Oracle Streams Replication Administrator's Guide*
- *Oracle Streams XStream Java API Reference*
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Streams Advanced Queuing User's Guide*

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

XStream Concepts and Use Cases

This part describes XStream concepts and use cases. This part contains the following chapters:

1

XStream Concepts

This chapter contains concepts related to XStream.

This chapter contains these topics:

- About XStream
- XStream Out
- XStream In
- Position Order in an LCR Stream
- XStream and SQL Generation

See Also:

- Chapter 2, "XStream Use Cases"
- Chapter 3, "Configuring XStream"
- Chapter 4, "Managing XStream"
- Chapter 5, "Monitoring XStream"
- Chapter 6, "Troubleshooting XStream"

About XStream

XStream provides application programming interfaces (APIs) that enable client applications to receive data changes from an Oracle database and to send data changes to an Oracle database. These data changes can be shared between Oracle databases and other systems. The other systems include non-Oracle databases, non-RDBMS Oracle products, file systems, third party software applications, and so on. The client application is designed by the user for specific purposes and use cases.

XStream consists of two major features: XStream Out and XStream In. XStream Out provides APIs that enable you to share data changes made to an Oracle database with other systems. XStream In provides APIs that enable you to share data changes made to other systems with Oracle databases.

XStream is built on the infrastructure of Oracle Streams. Therefore, XStream inherits Oracle Streams flexibility and functionality, including:

- Filtering of database changes at the database level, schema level, table level, and row/column level
- Rules and rule sets that control behavior, including inclusion and exclusion rules
- Rule-based transformation that modify captured data changes

- Support for the data types supported by Oracle Streams, including LOBs, `LONG`, `LONG RAW`, and `XMLType`
- Customized configurations, including multiple inbound streams to a single database instance, multiple outbound streams from a single database instance, multiple outbound streams from a single capture process, and so on
- Support for data manipulation language (DML) and data definition language (DDL) changes
- Full-featured apply for XStream In, including apply parallelism for optimal performance, SQL generation, conflict detection and resolution, error handling, and customized apply with apply handlers

You can configure XStream using the `DBMS_XSTREAM_ADM` package.

Licensing XStream

Using the XStream APIs requires purchasing a license for the Oracle GoldenGate product. See the documentation for the Oracle GoldenGate product for more information:

`http://download.oracle.com/docs/cd/E15881_01/index.htm`

XStream and Security

XStream Out allows a user to receive logical change records (LCRs). After an XStream Out user receives LCRs, the user might save the contents of LCRs to a file or generate the SQL statements to execute the LCRs on a non-Oracle database. XStream In allows a user to update tables in its own schema. XStream does not assume that the connected user to the inbound server or outbound server is trusted.

Java and Oracle Call Interface (OCI) client applications must connect to an Oracle database before attaching to an XStream outbound server created on that database. The connected user must be the same as the connect user configured for outbound server. Otherwise, an error is raised.

Java and Oracle Call Interface (OCI) client applications must connect to an Oracle database before attaching to an XStream inbound server created on that database. The connected user must be the same as the apply user configured for the inbound server. Otherwise, an error is raised.

The XStream Java layer API relies on Oracle JDBC security because XStream accepts the Oracle JDBC connection instance created by client applications in the XStream attach API. The connected user is then validated as an XStream user.

See Also:

- "Security Model" on page 7-4 for information about the security requirements for configuring and managing XStream
- *Oracle Streams Concepts and Administration* for information about apply users

Other Ways to Share Information in a Heterogeneous Environment

This chapter describes XStream, a new feature in Oracle Database 11*g* Release 2 (11.2). XStream enables heterogeneous information sharing with outstanding performance and usability.

Oracle Streams provides other ways to implement heterogeneous information sharing, both in past releases and in the current release. These ways include:

- Replicating data changes to a non-Oracle database using an Oracle Database Gateway.
- Dequeuing messages from an Oracle database using a Java Message Service (JMS) client.
- Enqueuing messages directly into an Oracle database queue with a client application.

See Also:

- *Oracle Streams Replication Administrator's Guide*
- *Oracle Database 2 Day + Data Replication and Integration Guide*
- *Oracle Streams Advanced Queuing User's Guide*

XStream Out

XStream Out can capture transactions from the redo log of an Oracle database and send them efficiently to a client application. XStream Out provides a transaction-based interface for streaming these changes to client applications. The client application can interact with other systems, including non-Oracle systems, such as non-Oracle databases or a file systems.

XStream Out has both Oracle Call Interface (OCI) and Java interfaces and supports all of the data types that are supported by Oracle Streams, including LOBs, `LONG`, `LONG RAW`, and `XMLType`.

This section contains these topics:

- The Outbound Server
- Considerations for XStream Outbound Servers
- XStream Out and Distributed Transactions

See Also:

- Part IV, "XStream OCI API Reference"
- *Oracle Database XStream Java API Reference*

The Outbound Server

With XStream Out, an Oracle Streams apply process functions as an **outbound server**. A client application can attach to an outbound server and extract row changes from logical change records (LCRs). A client application attaches to the outbound server using the OCI or Java interface. Only one client application at a time can attach to an outbound server.

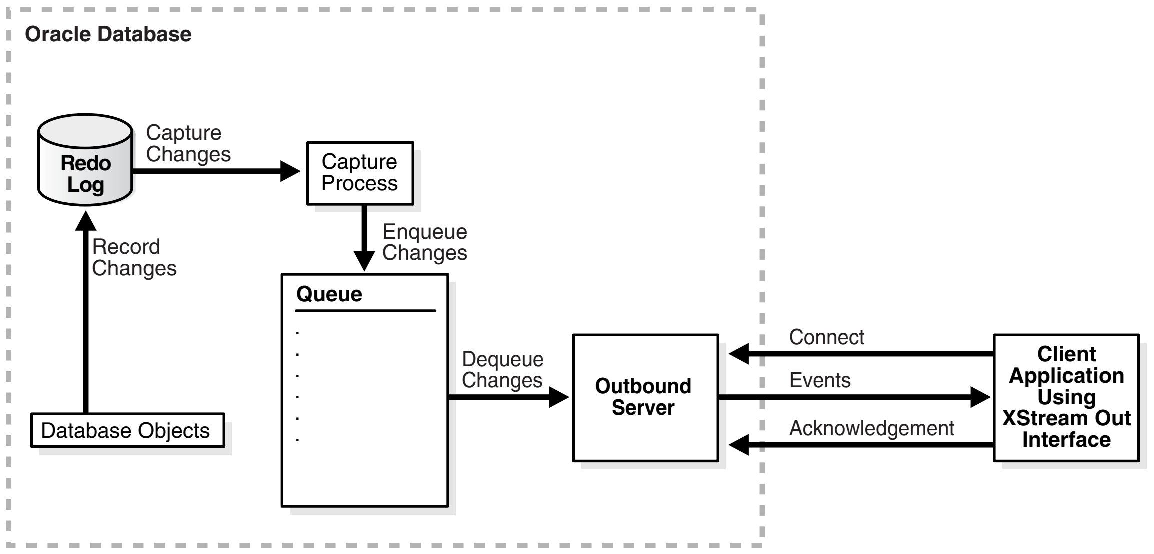
In an XStream Out configuration, an Oracle Streams capture process captures database changes and sends these changes to an outbound server. Change capture can be performed on the same database as the outbound server or on a different database. When change capture is performed on a different database than the one that contains the outbound server, a propagation sends the changes from the change capture database to the outbound server database. Downstream capture is also a supported mode to reduce the load on the source database.

When both the capture process and the outbound server are enabled, data changes, encapsulated in row LCRs and DDL LCRs, are sent to the outbound server. The outbound server can publish LCRs in various formats, such as OCI and Java. The client application can process LCRs that are passed to it from the outbound server or wait for LCRs from the outbound server by using a loop.

An outbound server sends LOB, `LONG`, `LONG RAW`, and `XMLType` data to the client application in chunks. Several chunks comprise a single column value of LOB, `LONG`, `LONG RAW`, or `XMLType` data type.

Figure 1–1 shows an outbound server configuration.

Figure 1–1 XStream Out Outbound Server



The client application can detach from the outbound server whenever necessary. When the client application re-attaches, the outbound server automatically determines where in the stream of LCRs the client application was when it detached. The outbound server starts sending LCRs from this point forward.

Outbound Servers and Apply Process Features

An Oracle Streams apply process functions as an outbound server, but some apply process features are not applicable to an outbound server. The following sections describe which apply process features are applicable to outbound servers and which are not:

- Apply Process Features That Are Applicable to Outbound Servers
- Apply Process Features That Are Not Applicable to Outbound Servers

Apply Process Features That Are Applicable to Outbound Servers The following apply process features can be used with outbound servers:

- Rules and rule sets

  See *Oracle Streams Concepts and Administration*.

- Rule-based transformations

  When a custom rule-based transformation is specified on a rule used by an outbound server, the user who calls the transformation function is the connect user for the outbound server.

See *Oracle Streams Concepts and Administration*.

- The following apply process parameters:
  - `disable_on_limit`
  - `maximum_scn`
  - `startup_seconds`
  - `time_limit`
  - `trace_level`
  - `transaction_limit`
  - `txn_age_spill_threshold`
  - `txn_lcr_spill_threshold`
  - `write_alert_log`

  These apply process parameters control the behavior of outbound servers

  See *Oracle Database PL/SQL Packages and Types Reference*.

- Transaction assembly by reader servers

  See *Oracle Streams Concepts and Administration*.

- The spilling of unapplied LCRs to hard disk

  See *Oracle Streams Concepts and Administration*.

- Instantiation SCN settings

  Instantiation SCNs are not required for database objects processed by an outbound server. If an instantiation SCN is set for a database object, then the outbound server only sends the LCRs for the database object with SCN values that are greater than the instantiation SCN value. If a database object does not have an instantiation SCN set, then the outbound server skips the instantiation SCN check and sends all LCRs for that database object that satisfy its rule sets.

  See *Oracle Streams Replication Administrator's Guide*.

Apply Process Features That Are Not Applicable to Outbound Servers The following apply process features cannot be used with outbound servers:

- Apply handlers

  You cannot specify an apply handler for an outbound server. The client application can perform custom processing of the logical change records (LCRs) instead if necessary. However, if there are apply processes configured in the same database as the outbound server, then you can specify apply handlers for these apply processes. In addition, you can configure general apply handlers for the database. An outbound server ignores general apply handlers.

  See *Oracle Streams Concepts and Administration*.

- The following apply process parameters:
  - `allow_duplicate_rows`
  - `commit_serialization`
  - `disable_on_error`
  - `parallelism`

  - `preserve_encryption`
  - `rtrim_on_implicit_conversion`

  Outbound servers ignore the settings for these apply process parameters.

  The `commit_serialization` parameter is always set to `FULL` for an outbound server, and the `parallelism` parameter is always set to `1` for an outbound server.

  See *Oracle Database PL/SQL Packages and Types Reference*.

- Apply tags

  An outbound server cannot set an apply tag for the changes it processes.

  See *Oracle Streams Replication Administrator's Guide*.

- Apply database links

  Outbound servers cannot use database links.

  See *Oracle Streams Replication Administrator's Guide*.

- Conflict detection and resolution

  An outbound server does not detect conflicts, and conflict resolution cannot be set for an outbound server.

  See *Oracle Streams Replication Administrator's Guide*.

- Dependency scheduling

  An outbound server does not evaluate dependencies because its parallelism must be 1.

  See *Oracle Streams Concepts and Administration*.

- Substitute key column settings

  An outbound server ignores substitute key column settings.

  See *Oracle Streams Concepts and Administration*.

- Enqueue directives specified by the `SET_ENQUEUE_DESTINATION` procedure in the `DBMS_APPLY_ADM` package

  An outbound server cannot enqueue changes into an Oracle database queue automatically using the `SET_ENQUEUE_DESTINATION` procedure.

  See *Oracle Database PL/SQL Packages and Types Reference*.

- Execute directives specified by the `SET_EXECUTE` procedure in the `DBMS_APPLY_ADM` package

  An outbound server ignores execute directives.

  See *Oracle Database PL/SQL Packages and Types Reference*.

- Error creation and execution

  An outbound server does not create an error transaction when it encounters an error. It records information about errors in the `ALL_APPLY` and `DBA_APPLY` views, but it does not enqueue the transaction into the error queue.

  See *Oracle Streams Concepts and Administration*.

Considerations for XStream Outbound Servers

The following are considerations for XStream outbound servers:

- LCRs processed by an outbound server must be LCRs that were captured by a capture process. An outbound server does not support LCRs that were captured by synchronous captures or LCRs that were constructed by applications.
- A single outbound server can process captured LCRs from only one source database. The source database is the database where the change encapsulated in the LCR was generated in the redo log.
- The source database for the changes captured by a capture process must be at 10.2.0 or higher compatibility level for these changes to be processed by an outbound server.
- The capture process for an outbound server must be running on an Oracle Database 11*g* Release 2 (11.2) or later database.
- An outbound server appears as an Oracle Streams apply process in Oracle Enterprise Manager.
- Automatic split and merge of a stream is possible when the capture process and the outbound server for the stream run on different database instances. However, when the capture process and outbound server for a stream run on the same database instance, automatic split and merge of the stream is not possible. See *Oracle Streams Replication Administrator's Guide* for information about automatic split and merge.

XStream Out and Distributed Transactions

You can perform distributed transactions using either of the following methods:

- Modify tables in multiple databases in a coordinated manner using database links.
- Use the XA interface, as exposed by the `DBMS_XA` supplied PL/SQL package or by the OCI or JDBC libraries. The XA interface implements X/Open Distributed Transaction Processing (DTP) architecture.

In an XStream Out configuration, changes made to the source database during a distributed transaction using either of these two methods are streamed to an XStream outbound server. The outbound server sends the changes in a transaction to the XStream client application after the transaction has committed.

However, the distributed transaction state is not replicated or sent. The client application does not inherit the in-doubt or prepared state of such a transaction. Also, XStream does not replicate or send the changes using the same global transaction identifier used at the source database for XA transactions.

XA transactions can be performed in two ways:

- Tightly coupled, where different XA branches share locks
- Loosely coupled, where different XA branches do not share locks

XStream supports replication of changes made by loosely coupled XA branches regardless of the `COMPATIBLE` initialization parameter value. XStream supports replication of changes made by tightly coupled branches on an Oracle RAC source database only if the `COMPATIBLE` initialization parameter set to `11.2.0` or higher.

See Also:

- *Oracle Database Administrator's Guide* for more information about distributed transactions
- *Oracle Database Advanced Application Developer's Guide* for more information about Oracle XA

XStream In

XStream In enables a remote client application to send information into an Oracle database from another system, such as a non-Oracle database or a file system. XStream In provides an efficient, transaction-based interface for sending information to an Oracle database from client applications. XStream In can consume the information coming into the Oracle database in several ways, including data replication, auditing, and change data capture. XStream In supports both Oracle Call Interface (OCI) and Java interfaces.

When compared with OCI client applications that make data manipulation language (DML) changes to an Oracle database directly, XStream In is more efficient for near real-time, transaction-based, heterogeneous DML changes to Oracle databases.

XStream In uses the following features of Oracle Streams:

- High performance processing of DML changes using an apply process and, optionally, apply process parallelism
- Apply process features such as SQL generation, conflict detection and resolution, error handling, and customized processing with apply handlers
- Streaming network transmission of information with minimal network round-trips
- Rules, rule sets, and rule-based transformations. When a custom rule-based transformation is specified on a rule used by an inbound server, the user who calls the transformation function is the apply user for the inbound server.

XStream In supports all of the data types that are supported by Oracle Streams, including LOBs, `LONG`, `LONG RAW`, and `XMLType`. A client application sends LOB and `XMLType` data to the inbound server in chunks. Several chunks comprise a single column value of LOB, `LONG`, `LONG RAW`, or `XMLType` data type.

This section contains these topics:

- The Inbound Server
- Considerations for XStream Inbound Servers

See Also:

- Part IV, "XStream OCI API Reference"
- *Oracle Database XStream Java API Reference*
- *Oracle Streams Concepts and Administration*

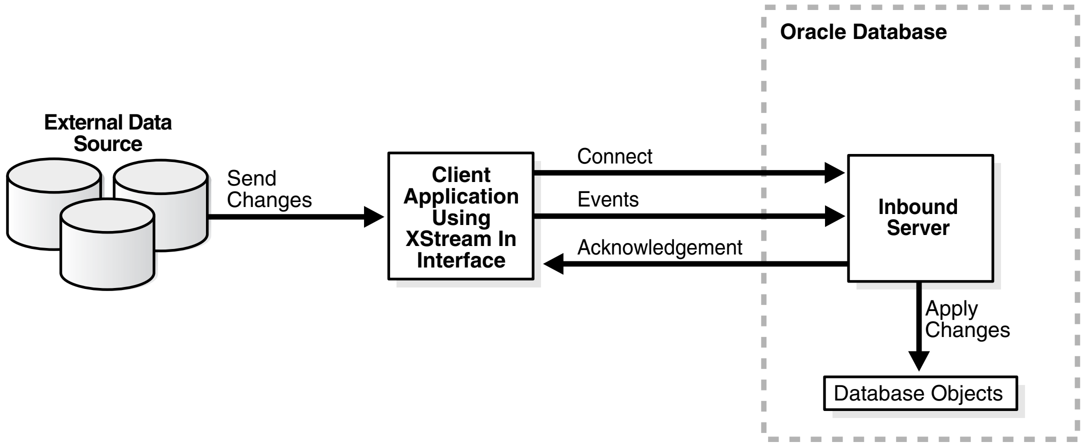
The Inbound Server

With XStream In, an Oracle Streams apply process functions as an **inbound server**. A client application can attach to an inbound server and send row changes and DDL changes encapsulated in logical change records (LCRs).

An external client application connects to the inbound server using the Oracle Call Interface (OCI) or the Java interface. After the connection is established, the client application acts as the capture agent for the inbound server by streaming LCRs to it. A client application can attach to only one inbound server at a time, and it can detach from the inbound server whenever necessary.

Figure 1–2 shows an inbound server configuration.

Figure 1–2 XStream In Inbound Server



Note: An inbound server uses a queue that is not shown in Figure 1–2. An inbound server's queue only is used to store error transactions.

Considerations for XStream Inbound Servers

The following are considerations for XStream inbound servers:

- The inbound server ignores the setting for the `maximum_scn` apply process parameter because LCRs sent to the inbound server by the client application might not have SCN values.
- Currently, an inbound server appears as an Oracle Streams apply process in Oracle Enterprise Manager.

Position Order in an LCR Stream

The following sections describe the position order in a logical change record (LCR) stream for both XStream Out and XStream In:

- About Position Order
- Position of LCRs and XStream Out
- Position of LCRs and XStream In
- Summary of Position Use in XStream Out and XStream In

About Position Order

Both XStream Out and XStream In use logical change record (LCR) streams to share transactions. XStream Out sends LCR streams to a client application. XStream In receives LCR streams from a client application.

Each LCR has a position attribute. The position of an LCR identifies its placement in the stream of LCRs in a transaction. Each LCR position has the following properties:

- The position is unique for each LCR.
- The position is of `RAW` data type.
- The position is strictly increasing within the LCR stream, within a transaction, and across transactions.
- The position is byte-comparable, and the comparison results for multiple positions determines the ordering of the LCRs in the stream.
- The position of an LCR remains identical when the database, the client application, or an Oracle Streams components restarts.
- The position is not affected by any Oracle Streams rule changes that might reduce or increase the number of LCRs in the stream.

XStream Out only sends committed data, and XStream In only receives committed data.

The following are the properties related to an LCR stream:

- An LCR stream must be repeatable.
- An LCR stream must contain a list of assembled, committed transactions. LCRs from one transaction are contiguous. There is no interleaving of transactions in an LCR stream.
- Each transaction within an LCR stream must have an ordered list of LCRs and a transaction ID.
- The last LCR in each transaction must be a commit LCR.
- Each LCR must have a unique position.
- The position of all LCRs within a single transaction and across transactions must be strictly increasing.

An LCR stream can batch LCRs from multiple transactions and arrange them in increasing position order. LCRs from one transaction are contiguous, and the position must be increasing in the transaction. Also, the position must be nonzero for all LCRs.

Position of LCRs and XStream Out

A **position** identifies the placement of a logical change record (LCR) in a stream of LCRs. The position of each LCR both within a transaction and across transactions is strictly increasing. See "Position Order in an LCR Stream" on page 1-9 for more information about positions.

Additional Logical Change Record (LCR) Attributes Related to Position

LCRs that were captured by a capture process contain the following additional attributes related to LCR position:

- The `scn_from_position` attribute contains the system change number (SCN) of the LCR.
- The `commit_scn_from_position` attribute contains the commit SCN of the transaction to which the LCR belongs.

Note: The `scn_from_position` and `commit_scn_from_position` attributes are not present in row LCRs captured by a synchronous capture nor in explicitly captured row LCRs

See Also:

- "GET_SCN_FROM_POSITION Static Function" on page 10-6
- "GET_COMMIT_SCN_FROM_POSITION Static Function" on page 10-5

The Processed Low Position and Restartability for XStream Out

If the outbound server or the client application stops abnormally, then the connection between the two is broken automatically. In this case, the client application must roll back all incomplete transactions. The client application must maintain its processed low position to recover properly after either it or the outbound server (or both) are restarted. The **processed low position** is a position below which all transactions have been processed by the client application. The processed low position indicates that the client has processed all LCRs that are less than or equal to this value. The client application can update the processed low position for each transaction that it consumes.

There are three possibilities related to the processed low position when the client application attaches to the outbound server:

- The client application can pass a processed low position to the outbound server that is equal to or greater than the outbound server's processed low position. In this case, the outbound server resumes streaming LCRs from the first LCR that has a position greater than the client application's processed low position.
- The client application can pass a processed low position to the outbound server that is less than the outbound server's processed low position. In this case, the outbound server raises an error.
- The client application can pass `NULL` to the outbound server. In this case, the outbound server determines the processed low position automatically and starts streaming LCRs from the LCR that has a position greater than this processed low position. When this happens, the client must suppress or discard each LCR with a position less than or equal to the client application's processed low position.

See Also: "Displaying the Processed Low Position for an Outbound Server" on page 5-3

Streaming Network Transmission

To minimize network latency, the outbound server streams logical change records (LCRs) to the client application with time-based acknowledgments. For example, the outbound server might send an acknowledgment every 30 seconds. This streaming protocol fully utilizes the available network bandwidth, and the performance is unaffected by the presence of a wide area network (WAN) separating the sender and the receiver. The outbound server extends the underlying Streams infrastructure, and the outbound server maintains the streaming performance rate.

Using OCI, you can control the time period of the interval by setting the `OCI_ATTR_XSTREAM_ACK_INTERVAL` attribute through the OCI client application. The default is 30 seconds. If this value is large, then the outbound server can stream out more LCRs for each acknowledgment interval. However, a longer interval delays how often the client application can send the processed low position to the outbound server. Therefore, a longer interval might mean that the processed low position maintained by the outbound server is not current. In this case, when the outbound server restarts, it must start processing LCRs at an earlier position than the one that corresponds to the processed low position maintained by the client application. In this case, more LCRs might be retransmitted, and the client application must discard the ones that have been applied.

Using Java, you can control the time period of the interval by setting the `batchInterval` parameter in the `attach` method in the `XStreamOut` class. The client application can specify this interval when it invokes the `attach` method.

Position of LCRs and XStream In

A position identifies the placement of a logical change record (LCR) in a stream of LCRs. The position of each LCR within a transaction is strictly increasing, and the position of each LCR across transactions also is strictly increasing. See "Position Order in an LCR Stream" on page 1-9 for more information about positions.

Each position must be encoded in a format (such as base-16 encoding) that supports byte comparison. The position is essential to the total order of the transaction stream sent by client applications using the XStream In interface.

The following positions are important for inbound servers:

- The **applied low position** indicates that the LCRs less than or equal to this value have been applied.

  An LCR is applied by an inbound server when the LCR has either been executed, sent to an apply handler, or moved to the error queue.

- The **spill position** indicates that the position less than or equal to this value have either been applied or spilled from memory to hard disk.
- The **applied high position** indicates the highest position of an LCR that has been applied.

  When the `commit_serialization` apply process parameter is set to `DEPENDENT_TRANSACTIONS` for an inbound server, an LCR with a higher commit position might be applied before an LCR with a lower commit position. When this happens, the applied high position is different from the applied low position.

- The **processed low position** is the higher value of either the applied low position or the spill position.

  The processed low position is the position below which the inbound server no longer requires any LCRs. This position corresponds with the oldest SCN for an Oracle Streams apply process that applies changes captured by a capture process.

  The processed low position indicates that the LCRs less than or equal to this position have been processed by the inbound server. If the client re-attaches to the inbound server, it only needs to send LCRs greater than the processed low position because the inbound server discards any LCRs that are less than or equal to the processed low position.

If the client application aborts abnormally, then the connection between the client application and the inbound server is automatically broken. Upon restart, the client application retrieves the processed low position from the inbound server and instructs its capture agent to retrieve changes starting from this processed low position.

To limit the recovery time of a client application using the XStream In interface, the client application can send activity, such as empty transactions, periodically to the inbound server. Row LCRs can include commit transaction control directives. When there are no LCRs to send to the server, the client can send a row LCR with a commit directive to advance the inbound server's processed low position. The activity acts as an acknowledgment so that the inbound server's processed low position can be advanced.

Example 1–1 Advancing the Processed Low Position of an Inbound Server

Consider a client and an external data source. The client application sends change made to the `hr.employees` table to the inbound server for processing, but the external data source includes many other tables, including the `oe.orders` table.

Assume that the following changes are made to the external data source:

Position	**Change**	**Client Application Activity**
1	Insert into the `hr.employees` table	Send row LCR including the change to the inbound server
2	Insert into the `oe.orders` table	None
3	Commit	Send a row LCR with a commit directive to inbound server
4	Insert into the `oe.orders` table	None
5	Update the `oe.orders` table	None
6	Commit	None
7	Commit	None
...	... (Activity on the external data source, but no changes to the `hr.employees` table)	None
100	Insert into the `oe.orders` table	None
101	Commit	None

The client application gets the changes from the external data source, generates appropriate LCRs, and sends the LCRs to the inbound server. Therefore, the inbound server receives the following LCRs:

- Row LCR for position 1
- Row LCR for position 3

After position 3 there are no relevant changes to send to the inbound server. If the inbound server restarts when the client application has processed all the changes up to position 101, then, after restarting, the client application must recheck all of the external database changes from position 4 forward because the inbound server's processed low position is 3.

Instead, assume that the client application sends commits to the inbound server periodically, even when there are no relevant changes to the `hr.employees` table:

Position	**Change**	**Client Application Activity**
1	Insert into the `hr.employees` table	Send row LCR including the change to the inbound server
2	Insert into the `oe.orders` table	None

Position	Change	Client Application Activity
3	Commit	Send a row LCR with a commit directive to inbound server
4	Insert into the `oe.orders` table	None
5	Update the `oe.orders` table	None
6	Commit	None
7	Commit	None
...	... (Activity on the external data source, but no changes to the `hr.employees` table)	Send several row LCR with a commit directives to the inbound server
100	Insert into the `oe.orders` table	None
101	Commit	Send a row LCR with a commit directive to the inbound server

In this case, the inbound server moves its processed low position to 101 when it has processed all of the row LCRs sent by the client application. If the inbound server restarts, its processed low position is 101, and the client application does not need to check all of the changes back to position 3.

The sample applications in "Sample XStream Client Application" on page 3-14 include code that sends a row LCR with a commit directive to an inbound server. Search for the word "ping" to find the parts of the applications that include this code.

See Also: "Displaying the Position Information for an Inbound Server" on page 5-6

Summary of Position Use in XStream Out and XStream In

Table 1–1 compares how the XStream Out outbound server and the XStream In inbound server use positions.

Table 1–1 Position Use in the Outbound Server and the Inbound Server

XStream Out Outbound Server	XStream In Inbound Server
The outbound server exposes the position.	The client application sets the position.
If the outbound server or client application stops abnormally, then all LCRs above the processed low position are resent. The processed low position is the apply process low watermark (LWM), and the apply process obtains the oldest SCN value by using this value.	If the inbound server or client application stops abnormally, then the client application must retransmit all LCRs with a position greater than or equal to the processed low position. The processed low position is the applied low water mark (LWM).

XStream and SQL Generation

SQL generation is the ability to generate the SQL statement required to perform the change encapsulated in a row logical change record (row LCR). Apply processes, XStream outbound servers, and XStream inbound servers can generate the SQL statement necessary to perform the insert, update, or delete operation in a row LCR.

This section contains these topics:

- Interfaces for Performing SQL Generation
- SQL Generation Formats

- Data Types and Character Sets
- Interfaces for Performing SQL Generation

See Also: *Oracle Streams Concepts and Administration*

Interfaces for Performing SQL Generation

You can use the following interfaces to perform SQL generation:

- The PL/SQL interface, which uses the `GET_ROW_TEXT` and `GET_WHERE_CLAUSE` member procedures for row LCRs
- The Oracle Call Interface (OCI) for XStream
- The Java interface for XStream

The PL/SQL interface generates SQL in a `CLOB` data type, while the OCI and Java interfaces generate SQL in plain text. In the Java interface, the size of the text is limited by the size of `String` data type.

See Also:

- Part IV, "XStream OCI API Reference"
- *Oracle Database XStream Java API Reference* for information about the Java interface for XStream
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `GET_ROW_TEXT` and `GET_WHERE_CLAUSE` row LCR member procedures

SQL Generation Formats

SQL statement can be generated in one of two formats: inline values or bind variables. Use inline values when the returned SQL statement is relatively small. For larger SQL statements, use bind variables. In this case, the bind variables are passed to the client application in a separate list that includes pointers to both old and new column values.

For information about using bind variables with each interface, refer to the following documentation:

- The documentation about the `GET_ROW_TEXT` and `GET_WHERE_CLAUSE` row LCR member procedures in *Oracle Database PL/SQL Packages and Types Reference*
- "OCILCRRowStmtWithBindVarGet()" on page 12-15
- The documentation about `DefaultRowLCR.getBinds()` in *Oracle Database XStream Java API Reference*

Note: For generated SQL statements with the values inline, SQL injection is possible. SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements, thereby gaining unauthorized access to a database to view or manipulate restricted data. Oracle strongly recommends using bind variables if you plan to execute the generated SQL statement. See *Oracle Database PL/SQL Language Reference* for more information about SQL injection.

Data Types and Character Sets

XStream outbound servers and inbound servers work the same as apply processes with regards to SQL generation and data types and character sets. For detailed information, see *Oracle Streams Concepts and Administration*.

SQL Generation Demo

A demo that performs SQL generation is available. The demo uses the `DBMS_XSTREAM_ADM` PL/SQL package to configure an XStream Out environment, and it uses either an Oracle Call Interface (OCI) client application or a Java client application to perform SQL generation.

The demo uses SQL generation to replicate data manipulation language (DML) changes from a source database to a destination database. Specifically, the demo creates the `xsdemosg` schema in both the source database and the destination database. It creates various types of tables in the `xsdemosg` schema at each database, including tables with LOB columns. It executes a set of DML statements on the tables in `xsdemosg` schema in the source database. Oracle Streams components, such as a capture process, queues, and a propagation, send the changes in the form of logical change records (LCRs) to an XStream outbound server that is also running on the source database. The outbound server makes the LCRs available to the client application.

The demo client application, when run, uses the OCI or Java API to connect to the outbound server and receive the LCRs. The demo client application uses SQL generation to execute the changes that are encapsulated in the LCRs. Therefore, the client application replicates the changes made to `xsdemosg` schema in the source database to the `xsdemosg` in the destination database.

You can modify the demo to replicate changes to any schema. Both the schema and the replicated tables must exist on both the source database and the destination database. SQL generation is only possible for DML changes. Therefore, this demo cannot be used to replicate data definition language (DDL) changes.

This demo is available in the following location:

```
$ORACLE_HOME/rdbms/demo/xstream/sqlgen
```

2 XStream Use Cases

XStream provides a flexible infrastructure for sharing information between Oracle data sources and non-Oracle data sources. Therefore, XStream can be used in many different ways to meet the needs of various organizations. This chapter describes the most common use cases for XStream.

This chapter contains these topics:

- Introduction to XStream Use Cases
- Replicating Data Changes With Non-Oracle Databases
- Using Files to Store Data Changes
- Sharing Data Changes With a Client-Side Memory Cache

See Also:

- Chapter 1, "XStream Concepts"
- Chapter 3, "Configuring XStream"
- Chapter 4, "Managing XStream"
- Chapter 5, "Monitoring XStream"
- Chapter 6, "Troubleshooting XStream"

Introduction to XStream Use Cases

Each XStream use case in this chapter contains three main elements:

- A general description of the use case as it applies to both XStream Out and XStream In
- A specific scenario for XStream Out with a reference to sample code in the Oracle Database installation
- A specific scenario for XStream In with a reference to sample code in the Oracle Database installation

Introduction to XStream Out Use Cases

In each XStream Out use case, the following components and actions send Oracle Database changes to a client application:

- Oracle Streams captures data changes made to an Oracle database.
- Oracle Streams sends these changes, in the form of logical change records (LCRs), to an outbound server.

- The outbound server sends the LCRs to a client application.

How the client application processes the LCRs is different for each use case.

See Also: "XStream Out" on page 1-3

Introduction to XStream In Use Cases

In each XStream In use case, a client application sends data changes made to an external data source to an Oracle Database:

- A client application gathers data changes and sends them to an inbound server.
- The inbound server receives the LCRs from a client application.
- The inbound server can apply the data changes to database objects in an Oracle database. The inbound server can also process the LCRs in a customized way.

How the client application gathers the data changes is different for each use case.

See Also: "XStream In" on page 1-8

Replicating Data Changes With Non-Oracle Databases

You can configure a heterogeneous replication environment with XStream. Replication is generally used to improve availability and to improve performance by spreading the network load over multiple regions and servers. In a heterogeneous replication environment, data is replicated between databases from different vendors. See *Oracle Streams Replication Administrator's Guide* for common reasons to use replication.

XStream Out can send data changes made to an Oracle database to a non-Oracle database. Specifically, the client application connects to the outbound server and receives changes made to tables within the Oracle database. The client application then applies the data changes in the LCRs to the non-Oracle database. The client application can process the LCRs in any customized way before applying them.

XStream In can receive data changes made to a non-Oracle database. Specifically, the client application gathers the data changes made to the non-Oracle database, formats these changes into logical change records (LCRs), and sends these LCRs to an inbound server. The inbound server applies these changes in the LCRs to the Oracle database.

Using Files to Store Data Changes

Some environments use files to store data changes. Typically, files store data changes for the following reasons.

- To process data changes in an environment that has no physical network or a limited physical network. For example, some locations do not have a physical network for security reasons.
- To process data changes in an environment that uses disconnected computing. For example, a salesperson might fill orders on a laptop at various locations without a network connection, and then update a primary database over the network once a day.
- To process data changes in an environment that uses satellite communications. In this case, bulk transfers of files is more efficient than incremental changes over the network.

XStream Out can send Oracle Database changes to a file in a file system. Specifically, the client application writes the data changes in logical change records (LCRs) to the file. The client application can process the LCRs in any customized way before writing them to the file, and the file can reside on the computer system running the client application or on a different computer system. Using SQL generation, the client application can also write the SQL statement required to perform the change encapsulated in a row LCR to a file.

XStream In can send data changes from a file to an Oracle Database. Specifically, the client application reads the data changes from the file and sends the changes, in the form of LCRs, to an inbound server.

See Also:

- "XStream and SQL Generation" on page 1-14
- *Oracle Streams Concepts and Administration*

XStream Demo That Replicates Database Changes Using Files

A demo is available that creates sample client applications that perform file-based replication using the XStream APIs. Specifically, at one database, the demo creates an XStream Out configuration captures database changes and sends the logical change records (LCRs) to an outbound server. A client application attaches to the outbound server and writes the database changes to a file.

At a different database, the demo creates an XStream In client application that attaches to an inbound server, reads the changes in the file, and sends them in the form of LCRs to the inbound server. The inbound server applied the changes to the database objects at the destination database.

This demo is available in the following location:

```
$ORACLE_HOME/rdbms/demo/xstream/fbr
```

Sharing Data Changes With a Client-Side Memory Cache

Some environments cache data in memory to improve performance. Cached data can provide low response times and high throughput for systems that require the best possible performance. XStream can share data changes incrementally with a client side memory cache.

XStream Out can incrementally refresh a client-side memory cache by sending Oracle Database changes to a memory cache. Specifically, the client application applies the data changes in the logical change records (LCRs) to the memory cache. The client application can process the LCRs in any customized way before applying them, and the memory cache can reside on the computer system running the client application or on a different computer system.

XStream In can incrementally retrieve data changes from a memory cache. Specifically, the client application retrieves the data changes and sends the changes, in the form of LCRs, to an inbound server. The memory cache can reside on the computer system running the client application or on a different computer system.

Part II

XStream Administration

This part describes XStream administration. This part contains the following chapters:

3

Configuring XStream

This chapter describes configuring the Oracle Database components that are used by XStream. This chapter also includes sample client applications that communicate with an XStream outbound server and inbound server.

This chapter contains these topics:

- Configuring XStream Out
- Configuring XStream In
- Sample XStream Client Application

See Also:

- Chapter 1, "XStream Concepts"
- Chapter 2, "XStream Use Cases"
- Chapter 5, "Monitoring XStream"
- Chapter 6, "Troubleshooting XStream"
- Part IV, "XStream OCI API Reference"
- *Oracle Database XStream Java API Reference*

Configuring XStream Out

An outbound server in an XStream Out configuration streams Oracle database changes to a client application. The client application attaches to the outbound server using the Oracle Call Interface (OCI) or Java interface to receive these changes.

Configuring an outbound server involves creating the Oracle Streams components that send captured database changes to the outbound server. It also involves configuring the outbound server itself, which includes specifying the connect user that the client application will use to attach to the outbound server.

This section contains these topics:

- Preparing for XStream Out
- Configuring an XStream Outbound Server
- Adding an Additional Outbound Server to a Capture Process Stream

Preparing for XStream Out

This section describes the decisions to make and the tasks to complete to prepare for an XStream Out configuration.

- Decide How to Configure the Oracle Streams Components
- Tasks to Complete Before Configuring XStream Out

Decide How to Configure the Oracle Streams Components

When you configure XStream Out, you must configure Oracle Streams components to capture database changes and send these changes to the outbound server in the form of logical change records (LCRs). These components include a capture process and at least one queue. The capture process can be a local capture process or a downstream capture process. For some configurations, you must also configure a propagation and an apply process.

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. The primary reason to use downstream capture is to reduce the load on the source database, thereby improving its performance. The primary reason to use a local capture is because it is easier to configure and maintain.

The database that captures changes made to the source database is called the capture database. One of the following databases can be the capture database:

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

If the source database or a third database is the capture database, then a propagation sends changes from the capture database to the database running the outbound server. If the database running the outbound server is the capture database, then this propagation between databases is not needed because the capture process and apply process use the same queue.

You can configure the Oracle Streams components in the following ways:

- **Local capture and outbound server on the same database:** The database objects, capture process, and outbound server are all in the same database. This configuration is the easiest to configure and maintain because all of the components are contained in one database.
- **Local capture and outbound server on different databases:** The database objects and capture process are in one database, and the outbound server is in another database. A propagation sends LCRs from the source database to the outbound server database. This configuration is best when you want easy configuration and maintenance and when you want to optimize the performance of the outbound server database.
- **Downstream capture and outbound server on the same database:** The database objects are in one database, and the capture process and outbound server are in another database. This configuration is best when you want to optimize the performance of the database with the database objects and want to offload change capture to another database. With this configuration, most of the components run on the database with the outbound server.
- **Downstream capture and outbound server on different databases:** The database objects are in one database, the outbound server is in another database, and the capture process is in a third database. This configuration is best when you want to optimize the performance of both the database with the database objects and the database with the outbound server. With this configuration, the capture process runs on a third database, and a propagation sends LCRs from the capture database to the outbound server database.

If you decide to configure a downstream capture process, then you must decide which type of downstream capture process you want to configure. The following types are available:

- A **real-time downstream capture process** configuration means that redo transport services use the log writer process (LGWR) at the source database to send redo data to the downstream database, and a remote file server process (RFS) at the downstream database receives the redo data over the network and stores the redo data in the standby redo log.
- An **archived-log downstream capture process** 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. These log files can be transferred automatically using redo transport services, or they can be transferred manually using a method such as FTP.

The advantage of real-time downstream capture over archived-log downstream capture is that real-time downstream capture reduces the amount of time required to capture changes made at the source database. The time is reduced because the real-time downstream capture process does not need to wait for the redo log file to be archived before it can capture changes from it. You can configure more than one real-time downstream capture process that captures changes from the same source database, but you cannot configure real-time downstream capture for multiple source databases at one downstream database.

The advantage of archived-log downstream capture over real-time downstream capture is that archived-log downstream capture allows downstream capture processes for multiple source databases at a downstream database. You can copy redo log files from multiple source databases to a single downstream database and configure multiple archived-log downstream capture processes to capture changes in these redo log files.

See Also: *Oracle Streams Concepts and Administration*

Tasks to Complete Before Configuring XStream Out

Preparing for an XStream Out outbound server is similar to preparing for an Oracle Streams replication environment. The components used in an Oracle Streams replication environment to capture changes and send them to an apply process are the same components used to capture changes and send them to an outbound server. These components include a capture process and one or more queues. If the capture process runs on a different database than the outbound server, then a propagation is also required.

Several of the tasks describe in this section are described in more detail in *Oracle Streams Replication Administrator's Guide*. This section provides an overview of each task and specific information about completing the task for an XStream Out configuration.

Complete the following tasks before configuring XStream Out:

- Configure an Oracle Streams Administrator on All Databases
- If Required, Configure Network Connectivity and Database Links
- Ensure That Each Source Database Is in ARCHIVELOG Mode
- Set Initialization Parameters Relevant to Oracle Streams
- Configure the Oracle Streams Pool

- If Required, Configure Log File Transfer to a Downstream Database
- If Required, Add Standby Redo Logs for Real-Time Downstream Capture

Configure an Oracle Streams Administrator on All Databases To configure and manage an XStream configuration, either create a new user with the appropriate privileges or grant these privileges to an existing user. You should not use the `SYS` or `SYSTEM` user as an Oracle Streams administrator, and the Oracle Streams administrator should not use the `SYSTEM` tablespace as its default tablespace. Create an Oracle Streams administrator on each database that is involved in the XStream configuration.

> **See Also:** *Oracle Streams Replication Administrator's Guide* for instructions about creating an Oracle Streams administrator

If Required, Configure Network Connectivity and Database Links Network connectivity and database links are not required when all of the Oracle Streams components run on the same database. These components include the capture process, queue, and outbound server.

You must configure network connectivity and database links if you decided to configure XStream in either of the following ways:

- The capture process and the outbound server will run on different databases.
- Downstream capture will be used.

See "Decide How to Configure the Oracle Streams Components" on page 3-2 for more information about these decisions.

If network connectivity is required, then configure your network and Oracle Net so that the databases can communicate with each other.

The following database links are required:

- When the capture process runs on a different database than the outbound server, create a database link from the capture database to the outbound server database. A propagation uses this database link to send changes from the capture database to the outbound server database.
- When you use downstream capture, create a database link from the capture database to the source database. The source database is the database that generates the redo data that the capture process uses to capture changes. The capture process uses this database link to perform administrative tasks at the source database.

The name of each database link must match the global name of the destination database, and each database link should be created in the Oracle Streams administrator's schema.

For example, if the global name of the source database is `dbs1.example.com`, the global name of the destination database is `dbs2.example.com`, and the Oracle Streams administrator is `strmadmin` at each database, then the following statement creates the database link from `dbs1.example.com` to `dbs2.example.com`:

```
CONNECT strmadmin@dbs1.example.com
Enter password: password

CREATE DATABASE LINK dbs2.example.com CONNECT TO strmadmin
   IDENTIFIED BY password USING 'dbs2.example.com';
```

See Also:

- *Oracle Database 2 Day DBA*
- *Oracle Database 2 Day + Data Replication and Integration Guide* for instructions about creating database links using Oracle Enterprise Manager
- *Oracle Database Administrator's Guide* for more information about database links

Ensure That Each Source Database Is in ARCHIVELOG Mode Each source database that generates changes that will be captured by a capture process must be in `ARCHIVELOG` mode. For downstream capture processes, the downstream database also must run in `ARCHIVELOG` mode if you plan to configure a real-time downstream capture process. The downstream database does not need to run in `ARCHIVELOG` mode if you plan to run only archived-log downstream capture processes on it.

If you are configuring Oracle Streams in an Oracle Real Application Clusters (Oracle RAC) environment, then the archive log files of all threads from all instances must be available to any instance running a capture process. This requirement pertains to both local and downstream capture processes.

See Also: *Oracle Database Administrator's Guide* for instructions about running a database in `ARCHIVELOG` mode

Set Initialization Parameters Relevant to Oracle Streams Some initialization parameters are important for the configuration, operation, reliability, and performance of Oracle Streams components. Set these parameters appropriately.

Oracle Streams Replication Administrator's Guide contains detailed information about all of the initialization parameters that are important for an Oracle Streams environment. In addition to the requirements described in *Oracle Streams Replication Administrator's Guide* for all Oracle Streams components, the following requirements apply to XStream outbound servers:

- Ensure that the `PROCESSES` initialization parameter is set to a value large enough to accommodate the outbound server background processes and all of the other Oracle Database background processes.
- Ensure that the `SESSIONS` initialization parameter is set to a value large enough to accommodate the sessions used by the outbound server background processes and all of the other Oracle Database sessions.

Configure the Oracle Streams Pool The Oracle Streams pool is a portion of memory in the System Global Area (SGA) that is used by Oracle Streams. The Oracle Streams pool stores buffered queue messages in memory, and it provides memory for capture processes and outbound servers. The Oracle Streams pool always stores LCRs captured by a capture process, and it stores LCRs and messages that are enqueued into a buffered queue by applications. Ensure that there is enough space in the Oracle Streams pool at each database to store LCRs and run the Oracle Streams components properly.

An outbound server requires 1 MB for each outbound server. The Oracle Streams pool is initialized the first time an inbound server is started.

See Also: *Oracle Streams Replication Administrator's Guide* for information about Oracle Streams pool requirements

If Required, Configure Log File Transfer to a Downstream Database If you decided to use a local capture process, then log file transfer is not required. However, if you decided to use downstream capture that uses redo transport services to transfer archived redo log files to the downstream database automatically, then configure log file transfer from the source database to the capture database. See "Decide How to Configure the Oracle Streams Components" on page 3-2 for information about this decision.

> **See Also:** *Oracle Streams Replication Administrator's Guide* for instructions

If Required, Add Standby Redo Logs for Real-Time Downstream Capture If you decided to configure real-time downstream capture, then add standby redo logs to the capture database. See "Decide How to Configure the Oracle Streams Components" on page 3-2 for information about this decision.

> **See Also:** *Oracle Streams Replication Administrator's Guide* for instructions

Configuring an XStream Outbound Server

You can create an outbound server using the following procedures in the DBMS_XSTREAM_ADM package:

- The CREATE_OUTBOUND procedure creates an outbound server, a queue, and a capture process on a single database with one procedure call.
- The ADD_OUTBOUND procedure only creates an outbound server. You must create the capture process and queue separately, and they must exist before you run the ADD_OUTBOUND procedure. You can configure the capture process on the same database as the outbound server or on a different database.

In both cases, you must create the client application that communicates with the outbound server and receives LCRs from the outbound server.

If you require multiple outbound servers, then you can use the CREATE_OUTBOUND procedure to create the capture process that captures database changes for the first outbound server. Next, you can run the ADD_OUTBOUND procedure to add additional outbound servers that receive the same captured changes. The capture process can reside on the same database as the outbound servers or on a different database.

This section contains these topics:

- Configuring an Outbound Server, Queue, and Capture Process
- Configuring an Outbound Server

Configuring an Outbound Server, Queue, and Capture Process

The CREATE_OUTBOUND procedure in the DBMS_XSTREAM_ADM package creates a capture process, queue, and outbound server on a single database. Both the capture process and the outbound server use the queue created by the procedure. When you run the procedure, you provide the name of the new outbound server, while the procedure generates a name for the capture process and queue.

The instructions in this section can only set up the configuration described in "local capture and outbound server on the same database" in "Decide How to Configure the Oracle Streams Components" on page 3-2. If you want all of the Oracle Streams components to run on the same database, then the CREATE_OUTBOUND procedure is the fastest and easiest way to create an outbound server.

To create an outbound server using the CREATE_OUTBOUND procedure:

1. Complete the tasks described in "Tasks to Complete Before Configuring XStream Out" on page 3-3.
2. In SQL*Plus, connect to the database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.
3. Run the CREATE_OUTBOUND procedure.

   For example, assume that you want to configure XStream Out in the following way:

   - The name of the outbound server is xout.
   - Data manipulation language (DML) and data definition language (DDL) changes made to the oe.orders and oe.order_items tables are sent to the outbound server.
   - DML and DDL changes made to the hr schema are sent to the outbound server.

   Given these assumptions, run the following CREATE_OUTBOUND procedure:

```
DECLARE
  tables  DBMS_UTILITY.UNCL_ARRAY;
  schemas DBMS_UTILITY.UNCL_ARRAY;
  BEGIN
    tables(1)  := 'oe.orders';
    tables(2)  := 'oe.order_items';
    schemas(1) := 'hr';
  DBMS_XSTREAM_ADM.CREATE_OUTBOUND(
    server_name     =>  'xout',
    table_names     =>  tables,
    schema_names    =>  schemas);
END;
/
```

   Running this procedure performs the following actions:

   - Configures supplemental logging for the oe.orders and oe.order_items tables and for all of the tables in the hr schema.
   - Creates a queue with a system-generated name that is used by the capture process and the outbound server.
   - Creates and starts a capture process with a system-generated name with rule sets that instruct it to capture DML and DDL changes to the oe.orders table, the oe.order_items table, and the hr schema.
   - Creates and starts an outbound server named xout with rule sets that instruct it to send DML and DDL changes to the oe.orders table, the oe.order_items table, and the hr schema to the client application.
   - Sets the current user as the connect user for the outbound server. In this example, the current user is the Oracle Streams administrator. The client application must connect to the database as the connect user to interact with the outbound server.

   Tip: To capture and send all database changes to the outbound server, specify NULL (the default) for the table_names and schema_names parameters.

4. Create and run the client application that will connect to the outbound server and receive the LCRs. See "Sample XStream Client Application" on page 3-14 for a sample application.
5. To add one or more additional outbound servers that receive LCRs from the capture process created in Step 3, follow the instructions in "Adding an Additional Outbound Server to a Capture Process Stream" on page 3-10.

Configuring an Outbound Server

The ADD_OUTBOUND procedure in the DBMS_XSTREAM_ADM package creates an outbound server. This procedure does not create the capture process or the queue. You must configure these components manually.

The instructions in this section can set up any of the configurations described in "Decide How to Configure the Oracle Streams Components" on page 3-2. However, if you chose "local capture and outbound server on the same database," then it is usually easier to use the CREATE_OUTBOUND procedure to configure all of the components simultaneously. See "Configuring an Outbound Server, Queue, and Capture Process" on page 3-6.

To create an outbound server using the ADD_OUTBOUND procedure:

1. Complete the tasks described in "Tasks to Complete Before Configuring XStream Out" on page 3-3.
2. In SQL*Plus, connect to the database that will run the capture process (the capture database) as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.
3. Create the queue that will be used by the capture process.

   See *Oracle Streams Replication Administrator's Guide* for instructions.
4. Create the capture process.

   See *Oracle Streams Replication Administrator's Guide* for instructions.
5. Connect to the source database. The source database is the database that contains the database objects for which the capture process will capture changes.
6. Ensure that required supplemental logging is specified for the database objects at the source database. Supplemental logging is required for the database objects for which the capture process will capture changes.

   If the capture database and the source database are the same database, then supplemental logging might have been specified during capture process creation.

   Ensure that the following supplemental logging is specified at the source database:

   - Any columns at the source database that are used in a primary key in tables for which changes are process by the outbound server must be unconditionally logged in a log group or by database supplemental logging of primary key columns.
   - Any columns at the source database that are used by a rule or a rule-based transformation must be unconditionally logged.

   See *Oracle Streams Replication Administrator's Guide* for instructions about specifying supplemental logging.
7. Connect to the database that will run the outbound server as the Oracle Streams administrator.

8. Create the queue that will be used by the outbound server.

   See *Oracle Streams Replication Administrator's Guide* for instructions.

9. Run the `ADD_OUTBOUND` procedure.

   For example, assume that you want to configure XStream Out in the following way:

   - The name of the outbound server is `xout`.
   - The queue used by the outbound server is `strmadmin.streams_queue`.
   - The source database is `db1.example.com`.
   - Data manipulation language (DML) and data definition language (DDL) changes made to the `oe.orders` and `oe.order_items` tables are sent to the outbound server.
   - DML and DDL changes made to the `hr` schema are sent to the outbound server.

   Given these assumptions, run the following `ADD_OUTBOUND` procedure:

```
DECLARE
  tables  DBMS_UTILITY.UNCL_ARRAY;
  schemas DBMS_UTILITY.UNCL_ARRAY;
  BEGIN
    tables(1)  := 'oe.orders';
    tables(2)  := 'oe.order_items';
    schemas(1) := 'hr';
  DBMS_XSTREAM_ADM.ADD_OUTBOUND(
    server_name     =>  'xout',
    queue_name      =>  'strmadmin.streams_queue',
    source_database =>  'db1.example.com',
    table_names     =>  tables,
    schema_names    =>  schemas);
END;
/
```

   Running this procedure performs the following actions:

   - Creates an outbound server named `xout`. The outbound server has rule sets that instruct it to send DML and DDL changes to the `oe.orders` table, the `oe.order_items` table, and the `hr` schema to the client application. The rules specify that these changes must have originated at the `db1.example.com` database. The outbound server dequeues LCRs from the queue `strmadmin.streams_queue`.
   - Sets the current user as the connect user for the outbound server. In this example, the current user is the Oracle Streams administrator. The client application must connect to the database as the connect user to interact with the outbound server.

     Tip: For the outbound server to receive all of the LCRs sent by the capture process, specify `NULL` (the default) for the `table_names` and `schema_names` parameters.

10. Connect to the capture database as the Oracle Streams administrator.

11. Create the propagation that sends logical change records (LCRs) from the capture process's queue on the local database to the queue used by the outbound server on the outbound server database.

    See *Oracle Streams Replication Administrator's Guide* for instructions.

12. Create and run the client application that will connect to the outbound server and receive the LCRs. See "Sample XStream Client Application" on page 3-14 for a sample application.

13. Start the outbound server. For example:

    ```
    exec DBMS_APPLY_ADM.START_APPLY('xout');
    ```

14. Start the capture process created in Step 4.

    See *Oracle Streams Concepts and Administration* for instructions.

15. To add one or more additional outbound servers that receive LCRs from the capture process created in Step 4, follow the instructions in "Adding an Additional Outbound Server to a Capture Process Stream" on page 3-10.

Adding an Additional Outbound Server to a Capture Process Stream

XStream Out configurations often require multiple outbound servers that process a stream of logical change records (LCRs) from a single capture process. This section describes adding an additional outbound server to a database that already includes at least one outbound server. The additional outbound server uses the same queue as another outbound server to receive the LCRs from the capture process.

Before completing the steps in this section, configure an XStream Out environment that includes at least one outbound server. The following sections describe configuring and XStream Out environment:

- "Configuring an Outbound Server, Queue, and Capture Process" on page 3-6
- "Configuring an Outbound Server" on page 3-8

When an XStream Out environment exists, use the ADD_OUTBOUND procedure in the DBMS_XSTREAM_ADM package to add another outbound server to a capture process stream.

To add another outbound server to a capture process stream using the ADD_OUTBOUND procedure:

1. In SQL*Plus, connect to the database that will run the additional outbound server as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Determine the name of the queue used by an existing outbound server that receives LCRs from the capture process.

   Run the "Displaying General Information About an Outbound Server" on page 5-2 query in to determine the owner and name of the queue. This query also shows the name of the capture process and the source database name. For this example, assume that the name of the queue is streams_queue and that the owner is strmadmin.

3. Run the ADD_OUTBOUND procedure.

   For example, assume that you want to configure the additional outbound server in the following way:

- The name of the outbound server is `xout2`.
- Data manipulation language (DML) and data definition language (DDL) changes made to the `oe.orders` and `oe.order_items` tables are sent to the outbound server.
- DML and DDL changes made to the `hr` schema are sent to the outbound server.

Given these assumptions, run the following `ADD_OUTBOUND` procedure:

```
DECLARE
  tables  DBMS_UTILITY.UNCL_ARRAY;
  schemas DBMS_UTILITY.UNCL_ARRAY;
  BEGIN
    tables(1)  := 'oe.orders';
    tables(2)  := 'oe.order_items';
    schemas(1) := 'hr';
  DBMS_XSTREAM_ADM.ADD_OUTBOUND(
    server_name     =>  'xout2',
    queue_name      =>  'strmadmin.streams_queue',
    source_database =>  'db1.example.com',
    table_names     =>  tables,
    schema_names    =>  schemas);
END;
/
```

Running this procedure performs the following actions:

- Creates an outbound server named `xout2`. The outbound server has rule sets that instruct it to send DML and DDL changes to the `oe.orders` table, the `oe.order_items` table, and the `hr` schema to the client application. The rules specify that these changes must have originated at the `db1.example.com` database. The outbound server dequeues LCRs from the queue `strmadmin.streams_queue`.
- Sets the current user as the connect user for the outbound server. In this example, the current user is the Oracle Streams administrator. The client application must connect to the database as the connect user to interact with the outbound server.

  Tip: For the outbound server to receive all of the LCRs sent by the capture process, specify `NULL` (the default) for the `table_names` and `schema_names` parameters.

4. If a client application does not already exist, then create and run the client application that will connect to the outbound server and receive the LCRs. See "Sample XStream Client Application" on page 3-14 for a sample application.

Configuring XStream In

An inbound server in an XStream In configuration receives a stream of changes from a client application. The inbound server can apply these changes to database objects in an Oracle database, or it can process the changes in a customized way. A client application can attach to an inbound server and send row changes and DDL changes encapsulated in logical change records (LCRs) using the OCI or Java interface.

Tasks to Complete Before Configuring XStream In

Complete the following tasks before configuring XStream In:

- Configure an Oracle Streams Administrator
- Set Initialization Parameters Relevant to Oracle Streams
- Configure the Oracle Streams Pool

Configure an Oracle Streams Administrator

To configure and manage an XStream configuration, either create a new user with the appropriate privileges or grant these privileges to an existing user. You should not use the `SYS` or `SYSTEM` user as an Oracle Streams administrator, and the Oracle Streams administrator should not use the `SYSTEM` tablespace as its default tablespace. Create an Oracle Streams administrator on the database that will run the XStream inbound server.

See Also: *Oracle Streams Replication Administrator's Guide* for instructions about creating an Oracle Streams administrator

Set Initialization Parameters Relevant to Oracle Streams

Some initialization parameters are important for the configuration, operation, reliability, and performance of XStream inbound servers. Set these parameters appropriately.

Oracle Streams Replication Administrator's Guide contains detailed information about all of the initialization parameters that are important for an Oracle Streams environment. In addition to the requirements described in *Oracle Streams Replication Administrator's Guide* for all Oracle Streams components, the following requirements apply to XStream inbound servers:

- Ensure that the `PROCESSES` initialization parameter is set to a value large enough to accommodate the inbound server background processes and all of the other Oracle Database background processes.
- Ensure that the `SESSIONS` initialization parameter is set to a value large enough to accommodate the sessions used by the inbound server background processes and all of the other Oracle Database sessions.

Configure the Oracle Streams Pool

The Oracle Streams pool is a portion of memory in the System Global Area (SGA) that provides memory for inbound servers. Ensure that there is enough space in the Oracle Streams pool for the inbound server to run properly. An inbound server requires 1 MB for each inbound server parallelism. For example, if parallelism is set to 4 for an inbound server, then at least 4 MB is required for the inbound server. The Oracle Streams pool also must have enough space to store LCRs before they are applied. The Oracle Streams pool is initialized the first time an inbound server is started.

See Also: *Oracle Streams Replication Administrator's Guide*

Configuring an XStream Inbound Server

The `CREATE_INBOUND` procedure in the `DBMS_XSTREAM_ADM` package creates an inbound server. You must create the client application that communicates with the inbound server and sends LCRs to the inbound server.

To create an inbound server:

1. Complete the tasks described in "Tasks to Complete Before Configuring XStream In" on page 3-12.
2. In SQL*Plus, connect to the database that will run the inbound server as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.
3. Run the CREATE_INBOUND procedure.

   For example, the following CREATE_INBOUND procedure configures inbound server named xin:

   ```
   BEGIN
     DBMS_XSTREAM_ADM.CREATE_INBOUND(
       server_name => 'xin',
       queue_name  => 'xin_queue');
   END;
   /
   ```

   Running this procedure performs the following actions:

   - Creates an inbound server named xin.
   - Sets the queue with the name xin_queue as the inbound server's queue, and creates this queue if it does not exist. This queue does not store LCRs sent by the client application. Instead, it stores error transactions if an LCR raises an error. The current user is the queue owner. In this example, the current user is the Oracle Streams administrator.
   - Sets the current user as the apply user for the inbound server. In this example, the current user is the Oracle Streams administrator. The client application must connect to the database as the apply user to interact with the inbound server.

   Tip: By default, an inbound server does not use rules or rule sets. Therefore, it processes all LCRs sent to it by the client application. To add rules and rule sets, use the DBMS_STREAMS_ADM package or the DBMS_RULE_ADM package. See *Oracle Streams Concepts and Administration*.
4. Create and run the client application that will connect to the inbound server and send LCRs to it. See "Sample XStream Client Application" on page 3-14 for a sample application.
5. If necessary, create apply handlers for the inbound server.

   Apply handlers are optional. Apply handlers process LCRs sent to an inbound server in a customized way.

   See Also: *Oracle Streams Concepts and Administration*
6. Start the inbound server. For example:

   ```
   exec DBMS_APPLY_ADM.START_APPLY('xin');
   ```

Sample XStream Client Application

This section contains a sample XStream client application. This application illustrates the basic tasks that are required of an XStream Out and XStream In application.

The application performs the following tasks:

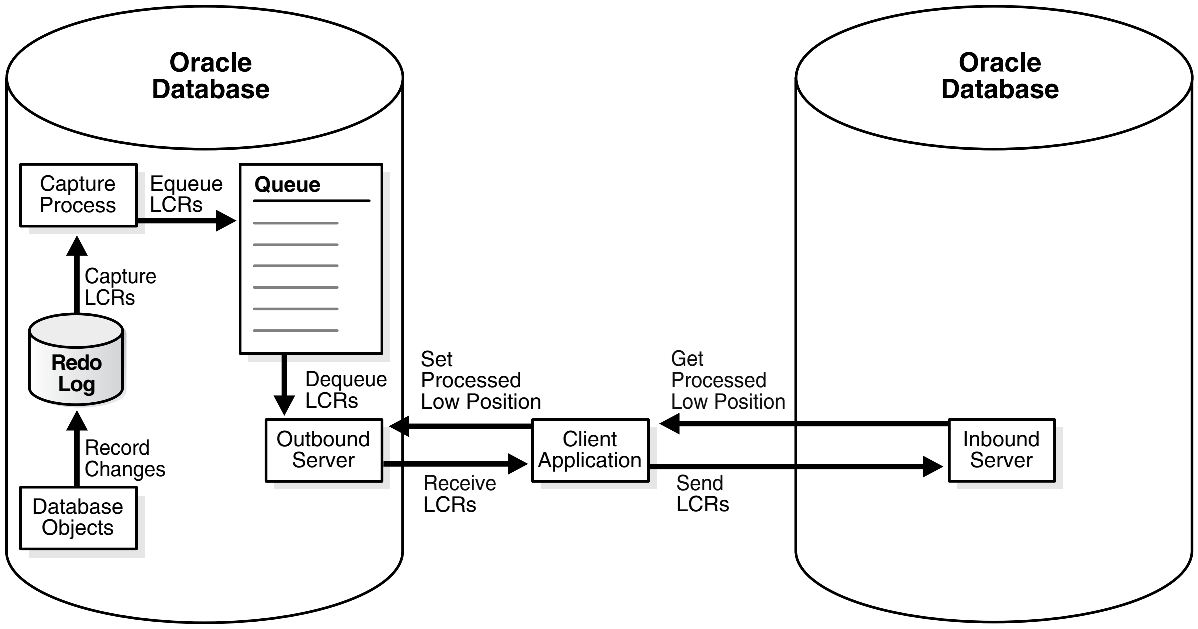
- It attaches to an XStream outbound server and inbound server and waits for logical change records (LCRs). The outbound server and inbound server are in two different databases.
- When it receives an LCR from the outbound server, it immediately sends the LCR to the inbound server.
- It periodically gets the processed low position from the inbound server and sends this value to the outbound server.
- It periodically sends a "ping" LCR from the outbound server to the inbound server to move the inbound server's processed low position forward in times of low activity.

In an XStream Out configuration that does not send LCRs to an inbound server, the client application must obtain the processed low position in another way.

This application waits indefinitely for transactions from the outbound server. To interrupt the application, enter `control-C`. If the program is restarted, then the outbound server starts sending LCRs from the processed low position that was set during the previous run.

Figure 3–1 provides an overview of the XStream environment configured in this section.

Figure 3–1 Sample XStream Configuration



Before running the sample application, ensure that the following components exist:

- Two Oracle databases with network connectivity between them
- An Oracle Streams administrator on both databases

- An outbound server configuration on one database, including a capture process, queue, and outbound server
- An inbound server configuration on another database

The sample applications in the following sections perform the same tasks. One sample application uses the OCI API, and the other uses the Java API.

- Sample XStream Client Application for the Oracle Call Interface API
- Sample XStream Client Application for the Java API

Note: An Oracle Database installation includes several XStream demos. These demos are in the following location:

```
$ORACLE_HOME/rdbms/demo/xstream
```

See Also:

- "Position of LCRs and XStream In" on page 1-12
- "Configuring XStream Out" on page 3-1
- "Configuring XStream In" on page 3-11
- *Oracle Streams Replication Administrator's Guide* for information about creating an Oracle Streams administrator

Sample XStream Client Application for the Oracle Call Interface API

To run the sample XStream client application for the Oracle Call Interface API, compile and link the application file, and enter the following on a command line:

```
xio -ob_svr xout_name -ob_db sn_xout_db -ob_usr xout_cu -ob_pwd xout_cu_pass
-ib_svr xin_name -ib_db sn_xin_db -ib_usr xin_au -ib_pwd xin_au_pass
```

where:

- *xout_name* is the name of the outbound server
- *sn_xout_db* is the service name for the outbound server's database
- *xout_cu* is the outbound server's connect user
- *xout_cu_pass* is the password for the outbound server's connect user
- *xin_name* is the name of the inbound server
- *sn_xin_db* is the service name for the inbound server's database
- *xin_au* is the inbound server's apply user
- *xin_au_pass* is the password for the inbound server's apply user

When the sample client application is running, it prints information about the row logical change records (LCRs) it is processing. The output looks similar to the following:

```
 ----------- ROW LCR Header  -----------------
  src_db_name=DB.EXAMPLE.COM
  cmd_type=UPDATE txid=17.0.74
  owner=HR oname=COUNTRIES
 ----------- ROW LCR Header  -----------------
  src_db_name=DB.EXAMPLE.COM
  cmd_type=COMMIT txid=17.0.74

 ----------- ROW LCR Header  -----------------
  src_db_name=DB.EXAMPLE.COM
  cmd_type=UPDATE txid=12.25.77
  owner=OE oname=ORDERS

 ----------- ROW LCR Header  -----------------
  src_db_name=DB.EXAMPLE.COM
  cmd_type=UPDATE txid=12.25.77
  owner=OE oname=ORDERS
```

This output contains the following information for each row LCR:

- `src_db_name` shows the source database for the change encapsulated in the row LCR.
- `cmd_type` shows the type of data manipulation language statement that made the change.
- `txid` shows the transaction ID of the transaction that includes the row LCR.
- `owner` shows the owner of the database object that was changed.
- `oname` shows the name of the database object that was changed.

This demo is available in the following location:

```
$ORACLE_HOME/rdbms/demo/xstream/oci
```

The file name for the demo is `xio.c`. See the `README.txt` file in the demo directory for more information about compiling and running the application.

The code for the sample application that uses the Oracle Call Interface API follows:

```
*/

#ifndef OCI_ORACLE
#include <oci.h>
#endif

#ifndef _STDIO_H
#include <stdio.h>
#endif

#ifndef _STDLIB_H
#include <stdlib.h>
#endif

#ifndef _STRING_H
#include <string.h>
#endif

#ifndef _MALLOC_H
#include <malloc.h>
#endif

/*----------------------------------------------------------------------
 *           Internal structures
 *----------------------------------------------------------------------*/
```

```
#define M_DBNAME_LEN    (128)

typedef struct conn_info                                     /* connect info */
{
  oratext * user;
  ub4       userlen;
  oratext * passw;
  ub4       passwlen;
  oratext * dbname;
  ub4       dbnamelen;
  oratext * svrnm;
  ub4       svrnmlen;
} conn_info_t;

typedef struct params
{
  conn_info_t  xout;                                        /* outbound info */
  conn_info_t  xin;                                          /* inbound info */
} params_t;

typedef struct oci                                            /* OCI handles */
{
  OCIEnv      *envp;                                 /* Environment handle */
  OCIError    *errp;                                       /* Error handle */
  OCIServer   *srvp;                                      /* Server handle */
  OCISvcCtx   *svcp;                                     /* Service handle */
  OCISession  *authp;
  OCIStmt    *stmtp;
  boolean     attached;
  boolean     outbound;
} oci_t;

static void connect_db(conn_info_t *opt_params_p, oci_t ** ocip, ub2 char_csid,
                       ub2 nchar_csid);
static void disconnect_db(oci_t * ocip);
static void ocierror(oci_t * ocip, char * msg);
static void attach(oci_t * ocip, conn_info_t *conn, boolean outbound);
static void detach(oci_t *ocip);
static void get_lcrs(oci_t *xin_ocip, oci_t *xout_ocip);
static void get_chunks(oci_t *xin_ocip, oci_t *xout_ocip);
static void print_lcr(oci_t *ocip, void *lcrp, ub1 lcrtype,
                      oratext **src_db_name, ub2  *src_db_namel);
static void print_chunk (ub1 *chunk_ptr, ub4 chunk_len, ub2 dty);
static void get_inputs(conn_info_t *xout_params, conn_info_t *xin_params,
                       int argc, char ** argv);
static void get_db_charsets(conn_info_t *params_p, ub2 *char_csid,
                            ub2 *nchar_csid);
static void set_client_charset(oci_t *outbound_ocip);

#define OCICALL(ocip, function) do {\
sword status=function;\
if (OCI_SUCCESS==status) break;\
else if (OCI_ERROR==status) \
{ocierror(ocip, (char *)"OCI_ERROR");\
exit(1);}\
else {printf("Error encountered %d\n", status);\
exit(1);}\
} while(0)
```

```
/*----------------------------------------------------------------------
 *                M A I N   P R O G R A M
 *---------------------------------------------------------------------*/
main(int argc, char **argv)
{
  /* Outbound and inbound connection info */
  conn_info_t   xout_params;
  conn_info_t   xin_params;
  oci_t        *xout_ocip = (oci_t *)NULL;
  oci_t        *xin_ocip = (oci_t *)NULL;
  ub2           obdb_char_csid = 0;                 /* outbound db char csid */
  ub2           obdb_nchar_csid = 0;               /* outbound db nchar csid */

  /* parse command line arguments */
  get_inputs(&xout_params, &xin_params, argc, argv);

  /* Get the outbound database CHAR and NCHAR character set info */
  get_db_charsets(&xout_params, &obdb_char_csid, &obdb_nchar_csid);

  /* Connect to the outbound db and set the client env to the outbound charsets
   * to minimize character conversion when transferring LCRs from outbound
   * directly to inbound server.
   */
  connect_db(&xout_params, &xout_ocip, obdb_char_csid, obdb_nchar_csid);

  /* Attach to outbound server */
  attach(xout_ocip, &xout_params, TRUE);

  /* connect to inbound db and set the client charsets the same as the
   * outbound db charsets.
   */
  connect_db(&xin_params, &xin_ocip, obdb_char_csid, obdb_nchar_csid);

  /* Attach to inbound server */
  attach(xin_ocip, &xin_params, FALSE);

  /* Get lcrs from outbound server and send to inbound server */
  get_lcrs(xin_ocip, xout_ocip);

  /* Detach from XStream servers */
  detach(xout_ocip);
  detach(xin_ocip);

  /* Disconnect from both databases */
  disconnect_db(xout_ocip);
  disconnect_db(xin_ocip);

  free(xout_ocip);
  free(xin_ocip);
  exit (0);
}

/*----------------------------------------------------------------------
 * connect_db - Connect to the database and set the env to the given
 * char and nchar character set ids.
 *---------------------------------------------------------------------*/
static void connect_db(conn_info_t *params_p, oci_t **ociptr, ub2 char_csid,
                ub2 nchar_csid)
{
  oci_t        *ocip;
  printf ("Connect to Oracle as %.*s@%.*s ",
          params_p->userlen, params_p->user,
          params_p->dbnamelen, params_p->dbname);

  if (char_csid && nchar_csid)
    printf ("using char csid=%d and nchar csid=%d", char_csid, nchar_csid);

  printf("\n");

  ocip = (oci_t *)malloc(sizeof(oci_t));

  if (OCIEnvNlsCreate(&ocip->envp, OCI_OBJECT, (dvoid *)0,
                    (dvoid * (*)(dvoid *, size_t)) 0,
                    (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                    (void (*)(dvoid *, dvoid *)) 0,
                    (size_t) 0, (dvoid **) 0, char_csid, nchar_csid))
  {
    ocierror(ocip, (char *)"OCIEnvCreate() failed");
  }

  if (OCIHandleAlloc((dvoid *) ocip->envp, (dvoid **) &ocip->errp,
                    (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    ocierror(ocip, (char *)"OCIHandleAlloc(OCI_HTYPE_ERROR) failed");
  }

  /* Logon to database */
  OCICALL(ocip,
          OCILogon(ocip->envp, ocip->errp, &ocip->svcp,
                   params_p->user, params_p->userlen,
                   params_p->passw, params_p->passwlen,
                   params_p->dbname, params_p->dbnamelen));

  /* allocate the server handle */
  OCICALL(ocip,
          OCIHandleAlloc((dvoid *) ocip->envp, (dvoid **) &ocip->srvp,
                         OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0));

  OCICALL(ocip,
          OCIHandleAlloc((dvoid *) ocip->envp, (dvoid **) &ocip->stmtp,
                    (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  if (*ociptr == (oci_t *)NULL)
  {
    *ociptr = ocip;
  }
}

/*---------------------------------------------------------------------
 * get_db_charsets - Get the database CHAR and NCHAR character set ids.
 *---------------------------------------------------------------------*/
static const oratext GET_DB_CHARSETS[] =  \
 "select parameter, value from nls_database_parameters where parameter = \
 'NLS_CHARACTERSET' or parameter = 'NLS_NCHAR_CHARACTERSET'";

#define PARM_BUFLEN      (30)

static void get_db_charsets(conn_info_t *params_p, ub2 *char_csid,
                            ub2 *nchar_csid)
```

```
{
  OCIDefine  *defnp1 = (OCIDefine *) NULL;
  OCIDefine  *defnp2 = (OCIDefine *) NULL;
  oratext     parm[PARM_BUFLEN];
  oratext     value[OCI_NLS_MAXBUFSZ];
  ub2         parm_len = 0;
  ub2         value_len = 0;
  oci_t       ocistruct;
  oci_t      *ocip = &ocistruct;

  *char_csid = 0;
  *nchar_csid = 0;
  memset (ocip, 0, sizeof(ocistruct));

  if (OCIEnvCreate(&ocip->envp, OCI_OBJECT, (dvoid *)0,
                   (dvoid * (*)(dvoid *, size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                   (void (*)(dvoid *, dvoid *)) 0,
                   (size_t) 0, (dvoid **) 0))
  {
    ocierror(ocip, (char *)"OCIEnvCreate() failed");
  }

  if (OCIHandleAlloc((dvoid *) ocip->envp, (dvoid **) &ocip->errp,
                   (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    ocierror(ocip, (char *)"OCIHandleAlloc(OCI_HTYPE_ERROR) failed");
  }

  OCICALL(ocip,
          OCILogon(ocip->envp, ocip->errp, &ocip->svcp,
                   params_p->user, params_p->userlen,
                   params_p->passw, params_p->passwlen,
                   params_p->dbname, params_p->dbnamelen));

  OCICALL(ocip,
          OCIHandleAlloc((dvoid *) ocip->envp, (dvoid **) &ocip->stmtp,
                      (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /* Execute stmt to select the db nls char and nchar character set */
  OCICALL(ocip,
          OCIStmtPrepare(ocip->stmtp, ocip->errp,
                         (CONST text *)GET_DB_CHARSETS,
                         (ub4)strlen((char *)GET_DB_CHARSETS),
                         (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));

  OCICALL(ocip,
          OCIDefineByPos(ocip->stmtp, &defnp1,
                         ocip->errp, (ub4) 1, parm,
                         PARM_BUFLEN, SQLT_CHR, (void*) 0,
                         &parm_len, (ub2 *)0, OCI_DEFAULT));

  OCICALL(ocip,
          OCIDefineByPos(ocip->stmtp, &defnp2,
                         ocip->errp, (ub4) 2, value,
                         OCI_NLS_MAXBUFSZ, SQLT_CHR, (void*) 0,
                         &value_len, (ub2 *)0, OCI_DEFAULT));

  OCICALL(ocip,
          OCIStmtExecute(ocip->svcp, ocip->stmtp,
                       ocip->errp, (ub4)0, (ub4)0,
                       (const OCISnapshot *)0,
                       (OCISnapshot *)0, (ub4)OCI_DEFAULT));

  while (OCIStmtFetch(ocip->stmtp, ocip->errp, 1,
                    OCI_FETCH_NEXT, OCI_DEFAULT) == OCI_SUCCESS)
  {
    value[value_len] = '\0';
    if (parm_len == strlen("NLS_CHARACTERSET") &&
        !memcmp(parm, "NLS_CHARACTERSET", parm_len))
    {
      *char_csid = OCINlsCharSetNameToId(ocip->envp, value);
      printf("Outbound database NLS_CHARACTERSET = %.*s (csid = %d) \n",
             value_len, value, *char_csid);
    }
    else if (parm_len == strlen("NLS_NCHAR_CHARACTERSET") &&
             !memcmp(parm, "NLS_NCHAR_CHARACTERSET", parm_len))
    {
      *nchar_csid = OCINlsCharSetNameToId(ocip->envp, value);
      printf("Outbound database NLS_NCHAR_CHARACTERSET = %.*s (csid = %d) \n",
             value_len, value, *nchar_csid);
    }
  }

  disconnect_db(ocip);
}

/*---------------------------------------------------------------------
 * attach - Attach to XStream server specified in connection info
 *---------------------------------------------------------------------*/
static void attach(oci_t * ocip, conn_info_t *conn, boolean outbound)
{
  sword       err;

  printf ("Attach to XStream %s server '%.*s'\n",
          outbound ? "outbound" : "inbound",
          conn->svrnmlen, conn->svrnm);

  if (outbound)
  {
    OCICALL(ocip,
            OCIXStreamOutAttach(ocip->svcp, ocip->errp, conn->svrnm,
                              (ub2)conn->svrnmlen, (ub1 *)0, 0, OCI_DEFAULT));
  }
  else
  {
    OCICALL(ocip,
            OCIXStreamInAttach(ocip->svcp, ocip->errp, conn->svrnm,
                               (ub2)conn->svrnmlen,
                               (oratext *)"From_XOUT", 9,
                               (ub1 *)0, 0, OCI_DEFAULT));
  }

  ocip->attached = TRUE;
  ocip->outbound = outbound;
}

/*---------------------------------------------------------------------
 * ping_svr - Ping inbound server by sending a commit LCR.
 *---------------------------------------------------------------------*/
```

```
static void ping_svr(oci_t *xin_ocip, void *commit_lcr,
                     ub1 *cmtpos, ub2 cmtpos_len,
                     oratext *source_db, ub2 source_db_len)
{
  OCIDate     src_time;
  oratext     txid[128];

  OCICALL(xin_ocip, OCIDateSysDate(xin_ocip->errp, &src_time));
  sprintf(txid, "Ping %2d:%2d:%2d",
          src_time.OCIDateTime.OCITimeHH,
          src_time.OCIDateTime.OCITimeMI,
          src_time.OCIDateTime.OCITimeSS);

  /* Initialize LCR with new txid and commit position */
  OCICALL(xin_ocip,
          OCILCRHeaderSet(xin_ocip->svcp, xin_ocip->errp,
                          source_db, source_db_len,
                          (oratext *)OCI_LCR_ROW_CMD_COMMIT,
                          (ub2)strlen(OCI_LCR_ROW_CMD_COMMIT),
                          (oratext *)0, 0,                     /* null owner */
                          (oratext *)0, 0,                    /* null object */
                          (ub1 *)0, 0,                           /* null tag */
                          txid, (ub2)strlen((char *)txid),
                          &src_time, cmtpos, cmtpos_len,
                          0, commit_lcr, OCI_DEFAULT));

  /* Send commit lcr to inbound server. */
  if (OCIXStreamInLCRSend(xin_ocip->svcp, xin_ocip->errp, commit_lcr,
                          OCI_LCR_XROW, 0, OCI_DEFAULT) == OCI_ERROR)
  {
    ocierror(xin_ocip, (char *)"OCIXStreamInLCRSend failed in ping_svr()");
  }
}

/*---------------------------------------------------------------------
 * get_lcrs - Get LCRs from outbound server and send to inbound server.
 *---------------------------------------------------------------------*/
static void get_lcrs(oci_t *xin_ocip, oci_t *xout_ocip)
{
  sword       status = OCI_SUCCESS;
  void       *lcr;
  ub1         lcrtype;
  oraub8      flag;
  ub1         proclwm[OCI_LCR_MAX_POSITION_LEN];
  ub2         proclwm_len = 0;
  ub1         sv_pingpos[OCI_LCR_MAX_POSITION_LEN];
  ub2         sv_pingpos_len = 0;
  ub1         fetchlwm[OCI_LCR_MAX_POSITION_LEN];
  ub2         fetchlwm_len = 0;
  void       *commit_lcr = (void *)0;
  oratext    *lcr_srcdb = (oratext *)0;
  ub2         lcr_srcdb_len = 0;
  oratext     source_db[M_DBNAME_LEN];
  ub2         source_db_len = 0;
  ub4         lcrcnt = 0;

  /* create an lcr to ping the inbound server periodically by sending a
   * commit lcr.
   */
  commit_lcr = (void*)0;
```

```
OCICALL(xin_ocip,
        OCILCRNew(xin_ocip->svcp, xin_ocip->errp, OCI_DURATION_SESSION,
                  OCI_LCR_XROW, &commit_lcr, OCI_DEFAULT));

while (status == OCI_SUCCESS)
{
  lcrcnt = 0;                         /* reset lcr count before each batch */

  while ((status =
              OCIXStreamOutLCRReceive(xout_ocip->svcp, xout_ocip->errp,
                                      &lcr, &lcrtype, &flag,
                                      fetchlwm, &fetchlwm_len, OCI_DEFAULT))
                                             == OCI_STILL_EXECUTING)
  {
    lcrcnt++;

    /* print header of LCR just received */
    print_lcr(xout_ocip, lcr, lcrtype, &lcr_srcdb, &lcr_srcdb_len);

    /* save the source db to construct ping lcr later */
    if (!source_db_len && lcr_srcdb_len)
    {
      memcpy(source_db, lcr_srcdb, lcr_srcdb_len);
      source_db_len = lcr_srcdb_len;
    }

    /* send the LCR just received */
    if (OCIXStreamInLCRSend(xin_ocip->svcp, xin_ocip->errp,
                            lcr, lcrtype, flag, OCI_DEFAULT) == OCI_ERROR)
    {
      ocierror(xin_ocip, (char *)"OCIXStreamInLCRSend failed");
    }

    /* If LCR has chunked columns (i.e, has LOB/Long/XMLType columns) */
    if (flag & OCI_XSTREAM_MORE_ROW_DATA)
    {
      /* receive and send chunked columns */
      get_chunks(xin_ocip, xout_ocip);
    }
  }

  if (status == OCI_ERROR)
    ocierror(xout_ocip, (char *)"OCIXStreamOutLCRReceive failed");

  /* clear the saved ping position if we just received some new lcrs */
  if (lcrcnt)
  {
    sv_pingpos_len = 0;
  }

  /* If no lcrs received during previous WHILE loop and got a new fetch
   * LWM then send a commit lcr to ping the inbound server with the new
   * fetch LWM position.
   */
  else if (fetchlwm_len > 0 && source_db_len > 0 &&
      (fetchlwm_len != sv_pingpos_len ||
       memcmp(sv_pingpos, fetchlwm, fetchlwm_len)))
  {
    /* To ensure we don't send multiple lcrs with duplicate position, send
     * a new ping only if we have saved the last ping position.
```

```
       */
      if (sv_pingpos_len > 0)
      {
        ping_svr(xin_ocip, commit_lcr, fetchlwm, fetchlwm_len,
                 source_db, source_db_len);
      }

      /* save the position just sent to inbound server */
      memcpy(sv_pingpos, fetchlwm, fetchlwm_len);
      sv_pingpos_len = fetchlwm_len;
    }

    /* flush inbound network to flush all lcrs to inbound server */
    OCICALL(xin_ocip,
            OCIXStreamInFlush(xin_ocip->svcp, xin_ocip->errp, OCI_DEFAULT));


    /* get processed LWM of inbound server */
    OCICALL(xin_ocip,
            OCIXStreamInProcessedLWMGet(xin_ocip->svcp, xin_ocip->errp,
                                        proclwm, &proclwm_len, OCI_DEFAULT));

    if (proclwm_len > 0)
    {
      /* Set processed LWM for outbound server */
      OCICALL(xout_ocip,
              OCIXStreamOutProcessedLWMSet(xout_ocip->svcp, xout_ocip->errp,
                                           proclwm, proclwm_len, OCI_DEFAULT));
    }
  }

  if (status != OCI_SUCCESS)
    ocierror(xout_ocip, (char *)"get_lcrs() encounters error");
}

/*----------------------------------------------------------------------
 * get_chunks - Get each chunk for the current LCR and send it to
 *              the inbound server.
 *---------------------------------------------------------------------*/
static void get_chunks(oci_t *xin_ocip, oci_t *xout_ocip)
{
  oratext *colname;
  ub2      colname_len;
  ub2      coldty;
  oraub8   col_flags;
  ub2      col_csid;
  ub4      chunk_len;
  ub1     *chunk_ptr;
  oraub8   row_flag;
  sword    err;
  sb4      rtncode;

  do
  {
    /* Get a chunk from outbound server */
    OCICALL(xout_ocip,
            OCIXStreamOutChunkReceive(xout_ocip->svcp, xout_ocip->errp,
                                      &colname, &colname_len, &coldty,
                                      &col_flags, &col_csid, &chunk_len,
                                      &chunk_ptr, &row_flag, OCI_DEFAULT));
    /* print chunked column info */
    printf(
      "  Chunked column name=%.*s DTY=%d  chunk len=%d csid=%d col_flag=0x%x\n",
      colname_len, colname, coldty, chunk_len, col_csid, col_flags);

    /* print chunk data */
    print_chunk(chunk_ptr, chunk_len, coldty);

    /* Send the chunk just received to inbound server */
    OCICALL(xin_ocip,
            OCIXStreamInChunkSend(xin_ocip->svcp, xin_ocip->errp, colname,
                                  colname_len, coldty, col_flags,
                                  col_csid, chunk_len, chunk_ptr,
                                  row_flag, OCI_DEFAULT));

  } while (row_flag & OCI_XSTREAM_MORE_ROW_DATA);
}

/*---------------------------------------------------------------------
 * print_chunk - Print chunked column information. Only print the first
 *               50 bytes for each chunk.
 *---------------------------------------------------------------------*/
static void print_chunk (ub1 *chunk_ptr, ub4 chunk_len, ub2 dty)
{
#define MAX_PRINT_BYTES     (50)          /* print max of 50 bytes per chunk */

  ub4  print_bytes;

  if (chunk_len == 0)
    return;

  print_bytes = chunk_len > MAX_PRINT_BYTES ? MAX_PRINT_BYTES : chunk_len;

  printf("  Data = ", chunk_len);
  if (dty == SQLT_CHR)
    printf("%.*s", print_bytes, chunk_ptr);
  else
  {
    ub2  idx;

    for (idx = 0; idx < print_bytes; idx++)
      printf("%02x", chunk_ptr[idx]);
  }
  printf("\n");
}

/*---------------------------------------------------------------------
 * print_lcr - Print header information of given lcr.
 *---------------------------------------------------------------------*/
static void print_lcr(oci_t *ocip, void *lcrp, ub1 lcrtype,
                      oratext **src_db_name, ub2  *src_db_namel)
{
  oratext     *cmd_type;
  ub2          cmd_type_len;
  oratext     *owner;
  ub2          ownerl;
  oratext     *oname;
  ub2          onamel;
  oratext     *txid;
  ub2          txidl;
  sword        ret;

  printf("\n ----------- %s LCR Header  -----------------\n",
         lcrtype == OCI_LCR_XDDL ? "DDL" : "ROW");

  /* Get LCR Header information */
  ret = OCILCRHeaderGet(ocip->svcp, ocip->errp,
                        src_db_name, src_db_namel,              /* source db */
                        &cmd_type, &cmd_type_len,            /* command type */
                        &owner, &ownerl,                       /* owner name */
                        &oname, &onamel,                      /* object name */
                        (ub1 **)0, (ub2 *)0,                      /* lcr tag */
                        &txid, &txidl, (OCIDate *)0,   /* txn id  & src time */
                        (ub2 *)0, (ub2 *)0,              /* OLD/NEW col cnts */
                        (ub1 **)0, (ub2 *)0,                 /* LCR position */
                        (oraub8*)0, lcrp, OCI_DEFAULT);

  if (ret != OCI_SUCCESS)
    ocierror(ocip, (char *)"OCILCRHeaderGet failed");
  else
  {
    printf("  src_db_name=%.*s\n  cmd_type=%.*s txid=%.*s\n",
           *src_db_namel, *src_db_name, cmd_type_len, cmd_type, txidl, txid );

    if (ownerl > 0)
      printf("  owner=%.*s oname=%.*s \n", ownerl, owner, onamel, oname);
  }
}

/*---------------------------------------------------------------------
 * detach - Detach from XStream server
 *---------------------------------------------------------------------*/
static void detach(oci_t * ocip)
{
  sword  err = OCI_SUCCESS;

  printf ("Detach from XStream %s server\n",
          ocip->outbound ? "outbound" : "inbound" );

  if (ocip->outbound)
  {
    OCICALL(ocip, OCIXStreamOutDetach(ocip->svcp, ocip->errp, OCI_DEFAULT));
  }
  else
  {
    OCICALL(ocip, OCIXStreamInDetach(ocip->svcp, ocip->errp,
                                     (ub1 *)0, (ub2 *)0,    /* processed LWM */
                                     OCI_DEFAULT));
  }
}

/*---------------------------------------------------------------------
 * disconnect_db  - Logoff from the database
 *---------------------------------------------------------------------*/
static void disconnect_db(oci_t * ocip)
{
  if (OCILogoff(ocip->svcp, ocip->errp))
  {
    ocierror(ocip, (char *)"OCILogoff() failed");
  }

  if (ocip->errp)
    OCIHandleFree((dvoid *) ocip->errp, (ub4) OCI_HTYPE_ERROR);

  if (ocip->envp)
    OCIHandleFree((dvoid *) ocip->envp, (ub4) OCI_HTYPE_ENV);
}

/*---------------------------------------------------------------------
 * ocierror - Print error status and exit program
 *---------------------------------------------------------------------*/
static void ocierror(oci_t * ocip, char * msg)
{
  sb4 errcode=0;
  text bufp[4096];

  if (ocip->errp)
  {
    OCIErrorGet((dvoid *) ocip->errp, (ub4) 1, (text *) NULL, &errcode,
                bufp, (ub4) 4096, (ub4) OCI_HTYPE_ERROR);
    printf("%s\n%s", msg, bufp);
  }
  else
    puts(msg);

  printf ("\n");
  exit(1);
}

/*--------------------------------------------------------------------
 * print_usage - Print command usage
 *---------------------------------------------------------------------*/
static void print_usage(int exitcode)
{
  puts("\nUsage: xio -ob_svr <outbound_svr> -ob_db <outbound_db>\n"
       "           -ob_usr <conn_user> -ob_pwd <conn_user_pwd>\n"
       "           -ib_svr <inbound_svr> -ib_db <inbound_db>\n"
       "           -ib_usr <apply_user> -ib_pwd <apply_user_pwd>\n");
  puts("  ob_svr  : outbound server name\n"
       "  ob_db   : database name of outbound server\n"
       "  ob_usr  : connect user to outbound server\n"
       "  ob_pwd  : password of outbound's connect user\n"
       "  ib_svr  : inbound server name\n"
       "  ib_db   : database name of inbound server\n"
       "  ib_usr  : apply user for inbound server\n"
       "  ib_pwd  : password of inbound's apply user\n");

  exit(exitcode);
}

/*--------------------------------------------------------------------
 * get_inputs - Get user inputs from command line
 *---------------------------------------------------------------------*/
static void get_inputs(conn_info_t *xout_params, conn_info_t *xin_params,
                       int argc, char ** argv)
{
  char * option;
  char * value;
```

```
memset (xout_params, 0, sizeof(*xout_params));
memset (xin_params, 0, sizeof(*xin_params));
while(--argc)
{
  /* get the option name */
  argv++;
  option = *argv;

  /* check that the option begins with a "-" */
  if (!strncmp(option, (char *)"-", 1))
  {
    option ++;
  }
  else
  {
    printf("Error: bad argument '%s'\n", option);
    print_usage(1);
  }

  /* get the value of the option */
  --argc;
  argv++;

  value = *argv;

  if (!strncmp(option, (char *)"ob_db", 5))
  {
    xout_params->dbname = (oratext *)value;
    xout_params->dbnamelen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ob_usr", 6))
  {
    xout_params->user = (oratext *)value;
    xout_params->userlen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ob_pwd", 6))
  {
    xout_params->passw = (oratext *)value;
    xout_params->passwlen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ob_svr", 6))
  {
    xout_params->svrnm = (oratext *)value;
    xout_params->svrnmlen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ib_db", 5))
  {
    xin_params->dbname = (oratext *)value;
    xin_params->dbnamelen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ib_usr", 6))
  {
    xin_params->user = (oratext *)value;
    xin_params->userlen = strlen(value);
  }
  else if (!strncmp(option, (char *)"ib_pwd", 6))
  {
    xin_params->passw = (oratext *)value;
    xin_params->passwlen = strlen(value);
  }
```

```
    else if (!strncmp(option, (char *)"ib_svr", 6))
    {
      xin_params->svrnm = (oratext *)value;
      xin_params->svrnmlen = strlen(value);
    }
    else
    {
      printf("Error: unknown option '%s'.\n", option);
      print_usage(1);
    }
  }

  /* print usage and exit if any argument is not specified */
  if (!xout_params->svrnmlen || !xout_params->passwlen ||
      !xout_params->userlen || !xout_params->dbnamelen ||
      !xin_params->svrnmlen || !xin_params->passwlen ||
      !xin_params->userlen || !xin_params->dbnamelen)
  {
    printf("Error: missing command arguments. \n");
    print_usage(1);
  }
}
```

Sample XStream Client Application for the Java API

To run the sample XStream client application for the Oracle Call Interface API, compile and link the application file, and enter the following on a command line:

```
java xio xsin_oraclesid xsin_host xsin_port xsin_username
xsin_passwd xin_servername xsout_oraclesid xsout_host xsout_port
xsout_username xsout_passwd xsout_servername
```

where:

- *xsin_oraclesid* is the name Oracle SID of the inbound server's database
- *xsin_host* is the host name of the computer system running the inbound server
- *xsin_port* is the port number of the listener for the inbound server's database
- *xsin_username* is the inbound server's apply user
- *xsin_passwd* is the password for the inbound server's apply user
- *xin_servername* is the name of the inbound server
- *xsout_oraclesid* is the name Oracle SID of the outbound server's database
- *xsout_host* is the host name of the computer system running the outbound server
- *xsout_port* is the port number of the listener for the outbound server's database
- *xsout_username* is the outbound server's connect user
- *xsout_passwd* is the password for the outbound server's connect user
- *xsout_servername* is the name of the outbound server

When the sample client application is running, it prints information attaching to the inbound server and outbound server, along with the last position for each server. The output looks similar to the following:

```
Attached to inbound server:xin
Inbound Server Last Position is:
0000000937e10000000100000001000000937e1000000010000000101
Attached to outbound server:xout
Last Position is: 0000000937e10000000100000001000000937e1000000010000000101
```

This demo is available in the following location:

```
$ORACLE_HOME/rdbms/demo/xstream/java
```

The file name for the demo is `xio.java`. See the `README.txt` file in the demo directory for more information about compiling and running the application.

The code for the sample application that uses the Java API follows:

```
import oracle.streams.*;
import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.*;
import oracle.sql.*;
import java.sql.*;
import java.util.*;

public class xio
{
  public static String xsinusername = null;
  public static String xsinpasswd = null;
  public static String xsinName = null;
  public static String xsoutusername = null;
  public static String xsoutpasswd = null;
  public static String xsoutName = null;
  public static String in_url = null;
  public static String out_url = null;
  public static Connection in_conn = null;
  public static Connection out_conn = null;
  public static XStreamIn xsIn = null;
  public static XStreamOut xsOut = null;
  public static byte[] lastPosition = null;
  public static byte[] processedLowPosition = null;

  public static void main(String args[])
  {
    // get connection url to inbound and outbound server
    in_url = parseXSInArguments(args);
    out_url = parseXSOutArguments(args);

    // create connection to inbound and outbound server
    in_conn = createConnection(in_url, xsinusername, xsinpasswd);
    out_conn = createConnection(out_url, xsoutusername, xsoutpasswd);

    // attach to inbound and outbound server
    xsIn = attachInbound(in_conn);
    xsOut = attachOutbound(out_conn);

    // main loop to get lcrs
    get_lcrs(xsIn, xsOut);

    // detach from inbound and outbound server
    detachInbound(xsIn);
    detachOutbound(xsOut);
  }
```

```
// parse the arguments to get the conncetion url to inbound db
public static String parseXSInArguments(String args[])
{
  String trace, pref;
  String orasid, host, port;

  if (args.length != 12)
  {
    printUsage();
    System.exit(0);
  }

  orasid = args[0];
  host = args[1];
  port = args[2];
  xsinusername = args[3];
  xsinpasswd = args[4];
  xsinName = args[5];

  System.out.println("xsin_host = "+host);
  System.out.println("xsin_port = "+port);
  System.out.println("xsin_ora_sid = "+orasid);

  String in_url = "jdbc:oracle:oci:@"+host+":"+port+":"+orasid;
  System.out.println("xsin connection url: "+ in_url);

  return in_url;
}

// parse the arguments to get the conncetion url to outbound db
public static String parseXSOutArguments(String args[])
{
  String trace, pref;
  String orasid, host, port;

  if (args.length != 12)
  {
    printUsage();
    System.exit(0);
  }

  orasid = args[6];
  host = args[7];
  port = args[8];
  xsoutusername = args[9];
  xsoutpasswd = args[10];
  xsoutName = args[11];


  System.out.println("xsout_host = "+host);
  System.out.println("xsout_port = "+port);
  System.out.println("xsout_ora_sid = "+orasid);

  String out_url = "jdbc:oracle:oci:@"+host+":"+port+":"+orasid;
  System.out.println("xsout connection url: "+ out_url);

  return out_url;
}

// print out sample program usage message
```

```
public static void printUsage()
{
  System.out.println("");
  System.out.println("Usage: java xio "+"<xsin_oraclesid> " + "<xsin_host> "
                                      + "<xsin_port> ");
  System.out.println("                "+"<xsin_username> " + "<xsin_passwd> "
                                      + "<xsin_servername> ");
  System.out.println("                "+"<xsout_oraclesid> " + "<xsout_host> "
                                      + "<xsout_port> ");
  System.out.println("                "+"<xsout_username> " + "<xsout_passwd> "
                                      + "<xsout_servername> ");
}

// create a connection to an Oracle Database
public static Connection createConnection(String url,
                                          String username,
                                          String passwd)
{
  try
  {
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
    return DriverManager.getConnection(url, username, passwd);
  }
  catch(Exception e)
  {
    System.out.println("fail to establish DB connection to: " +url);
    e.printStackTrace();
    return null;
  }
}

// attach to the XStream Inbound Server
public static XStreamIn attachInbound(Connection in_conn)
{
  XStreamIn xsIn = null;
  try
  {
    xsIn = XStreamIn.attach((OracleConnection)in_conn, xsinName,
                            "XSDEMOINCLIENT" , XStreamIn.DEFAULT_MODE);

    // use last position to decide where should we start sending LCRs
    lastPosition = xsIn.getLastPosition();
    System.out.println("Attached to inbound server:"+xsinName);
    System.out.print("Inbound Server Last Position is: ");
    if (null == lastPosition)
    {
      System.out.println("null");
    }
    else
    {
      printHex(lastPosition);
    }
    return xsIn;
  }
  catch(Exception e)
  {
    System.out.println("cannot attach to inbound server: "+xsinName);
    System.out.println(e.getMessage());
    e.printStackTrace();
    return null;
  }
}

// attach to the XStream Outbound Server
public static XStreamOut attachOutbound(Connection out_conn)
{
  XStreamOut xsOut = null;

  try
  {
    // when attach to an outbound server, client needs to tell outbound
    // server the last position.
    xsOut = XStreamOut.attach((OracleConnection)out_conn, xsoutName,
                              lastPosition, XStreamOut.DEFAULT_MODE);
    System.out.println("Attached to outbound server:"+xsoutName);
    System.out.print("Last Position is: ");
    if (lastPosition != null)
    {
      printHex(lastPosition);
    }
    else
    {
      System.out.println("NULL");
    }
    return xsOut;
  }
  catch(Exception e)
  {
    System.out.println("cannot attach to outbound server: "+xsoutName);
    System.out.println(e.getMessage());
    e.printStackTrace();
    return null;
  }
}

// detach from the XStream Inbound Server
public static void detachInbound(XStreamIn xsIn)
{
  byte[] processedLowPosition = null;
  try
  {
    processedLowPosition = xsIn.detach(XStreamIn.DEFAULT_MODE);
    System.out.print("Inbound server processed low Position is: ");
    if (processedLowPosition != null)
    {
      printHex(processedLowPosition);
    }
    else
    {
      System.out.println("NULL");
    }
  }
  catch(Exception e)
  {
    System.out.println("cannot detach from the inbound server: "+xsinName);
    System.out.println(e.getMessage());
    e.printStackTrace();
  }
}
```

```
// detach from the XStream Outbound Server
public static void detachOutbound(XStreamOut xsOut)
{
  try
  {
    xsOut.detach(XStreamOut.DEFAULT_MODE);
  }
  catch(Exception e)
  {
    System.out.println("cannot detach from the outbound server: "+xsoutName);
    System.out.println(e.getMessage());
    e.printStackTrace();
  }
}

public static void get_lcrs(XStreamIn xsIn, XStreamOut xsOut)
{
  byte[] ping_pos = null;
  byte[] fetchlwm = null;
  String src_db = null;

  if (null == xsIn)
  {
    System.out.println("xstreamIn is null");
    System.exit(0);
  }

  if (null == xsOut)
  {
    System.out.println("xstreamOut is null");
    System.exit(0);
  }

  try
  {
    while(true)
    {
      // receive an LCR from outbound server
      LCR alcr = xsOut.receiveLCR(XStreamOut.DEFAULT_MODE);
      fetchlwm = xsOut.getFetchLowWatermark();

      // save source db for ping lcr
      if (null != alcr)
        src_db = alcr.getSourceDatabaseName();

      if (xsOut.getBatchStatus() == XStreamOut.EXECUTING) // batch is active
      {
        assert alcr != null;
        // send the LCR to the inbound server
        xsIn.sendLCR(alcr, XStreamIn.DEFAULT_MODE);

        // also get chunk data for this LCR if any
        if (alcr instanceof RowLCR)
        {
          // receive chunk from outbound then send to inbound
          if (((RowLCR)alcr).hasChunkData())
          {
            ChunkColumnValue chunk = null;
            do
            {
              chunk = xsOut.receiveChunk(XStreamOut.DEFAULT_MODE);
              xsIn.sendChunk(chunk, XStreamIn.DEFAULT_MODE);
            } while (!chunk.isEndOfRow());
          }
        }
        processedLowPosition = alcr.getPosition();
        ping_pos = processedLowPosition;
      }
      else  // batch is end
      {
        assert alcr == null;

        // send ping lcr if we haven't received any lcr in the batch
        // but we got a new fetch lwm, then send a commit lcr to
        // ping the inbound server with the new fetch LWM position
        if (null != src_db && null != fetchlwm &&
            !samePos(fetchlwm,ping_pos))
        {
          xsIn.sendLCR(createPing(src_db, fetchlwm),
                       XStreamIn.DEFAULT_MODE);
          ping_pos = fetchlwm;
        }

        // flush the network
        xsIn.flush(XStreamIn.DEFAULT_MODE);
        // get the processed_low_position from inbound server
        processedLowPosition =
            xsIn.getProcessedLowWatermark();
        // update the processed_low_position at oubound server
        if (null != processedLowPosition)
          xsOut.setProcessedLowWatermark(processedLowPosition,
                                         XStreamOut.DEFAULT_MODE);
      }
    }
  }
  catch(Exception e)
  {
    System.out.println("exception when processing LCRs");
    System.out.println(e.getMessage());
    e.printStackTrace();
  }
}

public static void printHex(byte[] b)
{
  for (int i = 0; i < b.length; ++i)
  {
    System.out.print(
      Integer.toHexString((b[i]&0xFF) | 0x100).substring(1,3));
  }
  System.out.println("");
}

// ping lcr is used to bump up the inbound server watermark
private static RowLCR createPing(String src_db, byte[] pos)
{
  java.util.Date today = new java.util.Date();
  java.sql.Timestamp now = new java.sql.Timestamp(today.getTime());
  oracle.sql.DATE src_time = new oracle.sql.DATE(now);
```

```
    RowLCR alcr = new DefaultRowLCR();
    ((RowLCR)alcr).setSourceDatabaseName(src_db);
    ((RowLCR)alcr).setSourceTime(src_time);
    ((RowLCR)alcr).setPosition(pos);
    ((RowLCR)alcr).setCommandType(RowLCR.COMMIT);
    ((RowLCR)alcr).setTransactionId("Ping: " + src_time.toString());

    return alcr;
  }

  private static boolean samePos(byte[] pos1, byte[] pos2)
  {
    int   cmp_len;
    boolean    result;

    if (pos1.length != pos2.length)
      return false;

    for (int i = 0; i<pos1.length; i++)
    {
      if (pos1[i] != pos2[i])
        return false;
    }

    return true;
  }
}
```

4

Managing XStream

This chapter provides instructions for managing XStream.

This chapter contains these topics:

- Managing XStream Out
- Managing XStream In

See Also:

- Chapter 1, "XStream Concepts"
- Chapter 2, "XStream Use Cases"
- Chapter 3, "Configuring XStream"
- Chapter 5, "Monitoring XStream"
- Chapter 6, "Troubleshooting XStream"

Managing XStream Out

This section describes managing an XStream Out configuration.

This section contains these topics:

- Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM
- Managing Rules for an XStream Out Configuration
- Changing the Connect User for an Outbound Server
- Changing the Capture User of the Capture Process for an Outbound Server
- Dropping Oracle Streams Components in an XStream Out Configuration

---

Note: With XStream Out, an Oracle Streams apply process functions as an outbound server. Therefore, you can use the instructions for managing an apply process to manage an outbound server. See *Oracle Streams Concepts and Administration*.

See Also:

- "XStream Out" on page 1-3
- "Configuring XStream Out" on page 3-1
- "Monitoring XStream Out" on page 5-1

Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM

In some XStream Out configurations, you can use the DBMS_XSTREAM_ADM package to manage the capture process that captures changes for an outbound server. However, other configurations require that you use the DBMS_CAPTURE_ADM package or the DBMS_STREAMS_ADM package to manage the capture process.

Specifically, the DBMS_XSTREAM_ADM package can manage an outbound server's capture process in the following ways:

- Add rules to and remove rules from the capture process's rule sets
- Change the capture user for the capture process
- Drop the capture process

An outbound server's capture process can be managed by the DBMS_XSTREAM_ADM package in either of the following cases:

- The capture process was created by the CREATE_OUTBOUND procedure in the DBMS_XSTREAM_ADM package.
- The queue used by the capture process was created by the CREATE_OUTBOUND procedure.

To check whether an outbound server's capture process can be managed by the DBMS_XSTREAM_ADM package, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the following query:

```
COLUMN SERVER_NAME HEADING 'Outbound Server Name' FORMAT A30
COLUMN CAPTURE_NAME HEADING 'Capture Process Name' FORMAT A30

SELECT SERVER_NAME,
       CAPTURE_NAME
  FROM DBA_XSTREAM_OUTBOUND;
```

   Your output looks similar to the following:

```
Outbound Server Name           Capture Process Name
------------------------------ ------------------------------
XOUT                           CAP$_XOUT_4
```

   If the Capture Process Name for an outbound server is non-NULL, then the DBMS_XSTREAM_ADM package can manage the capture process. In this case, you can also manage the capture process using the DBMS_CAPTURE_ADM package or the DBMS_STREAMS_ADM package. However, it is usually better to manage the capture process for an outbound server using the DBMS_XSTREAM_ADM package when it is possible.

   If the Capture Process Name for an outbound server is NULL, then the DBMS_XSTREAM_ADM package cannot manage the capture process. In this case, you must manage the capture process using the DBMS_CAPTURE_ADM package or the DBMS_STREAMS_ADM package.

   See Also: *Oracle Streams Concepts and Administration* for information about managing a capture process using the DBMS_CAPTURE_ADM package or the DBMS_STREAMS_ADM package

Managing Rules for an XStream Out Configuration

This section describes managing rules for an XStream Out configuration. Rules control which database changes are streamed to the outbound server and which database changes the outbound server streams to the client application.

This section contains these topics:

- Adding Rules to an XStream Out Configuration
- Removing Rules from an XStream Out Configuration

See Also: *Oracle Streams Concepts and Administration*

Adding Rules to an XStream Out Configuration

This section describes adding schema rules, table rules, and subset rules to an XStream Out configuration.

This section contains these topics:

- Adding Schema Rules and Table Rules to an XStream Out Configuration
- Adding Subset Rules to an Outbound Server's Positive Rule Set

Adding Schema Rules and Table Rules to an XStream Out Configuration This section describes adding schema rules and table rules to an XStream Out configuration using the ALTER_OUTBOUND procedure in the DBMS_XSTREAM_ADM package. The ALTER_OUTBOUND procedure adds rules for both data manipulation language (DML) and data definition language (DDL) changes.

When you follow the instructions in this section, the ALTER_OUTBOUND procedure always adds rules for the specified schemas and tables to one of the outbound server's rule sets. If the DBMS_XSTREAM_ADM package can manage the outbound server's capture process, then the ALTER_OUTBOUND procedure also adds rules for the specified schemas and tables to one of the rule sets used by this capture process.

To determine whether the DBMS_XSTREAM_ADM package can manage the outbound server's capture process, see "Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM" on page 4-2. If the DBMS_XSTREAM_ADM package cannot manage the outbound server's capture process, then the ALTER_OUTBOUND procedure adds rules to the outbound server's rule set only. In this case, if rules for same schemas and tables should be added to the capture process's rule set as well, then see *Oracle Streams Concepts and Administration*.

To add schema rules and table rules to an XStream Out configuration, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the ALTER_OUTBOUND procedure, and specify the following parameters:

   - server_name - Specify the name of the outbound server.
   - table_names - Specify the tables for which to add rules, or specify NULL to add no table rules.
   - schema_name - Specify the schemas for which to add rules, or specify NULL to add no schema rules.

- add - Specify TRUE so that the rules are added. (Rules are removed if you specify FALSE.)
- inclusion_rule - Specify TRUE to add rules to the positive rule set of the outbound server, or specify FALSE to add rules to the negative rule set of the outbound server. If the DBMS_XSTREAM_ADM package can manage the outbound server's capture process, then rules are also added to this capture process's rule set.

The following examples add rules to the configuration of an outbound server named xout.

Example 4–1 Adding Rules for the hr Schema, oe.orders Table, and oe.order_items Table to the Positive Rule Set

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name    => 'xout',
    table_names    => 'oe.orders, oe.order_items',
    schema_names   => 'hr',
    add            => TRUE,
    inclusion_rule => TRUE);
END;
/
```

Example 4–2 Adding Rules for the hr Schema to the Negative Rule Set

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name    => 'xout',
    table_names    => NULL,
    schema_names   => 'hr',
    add            => TRUE,
    inclusion_rule => FALSE);
END;
/
```

Adding Subset Rules to an Outbound Server's Positive Rule Set This section describes adding subset rules to an outbound server's positive rule set using the ADD_SUBSET_OUTBOUND_RULES procedure in the DBMS_XSTREAM_ADM package. The ADD_SUBSET_OUTBOUND_RULES procedure only adds rules for data manipulation language (DML) changes to an outbound server's positive rule set. It does not add rules for data definition language (DDL) changes, and it does not add rules to a capture process's rule set.

To add subset rules to an outbound server's positive rule set, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the ADD_SUBSET_OUTBOUND_RULES procedure, and specify the following parameters:

   - server_name - Specify the name of the outbound server.
   - table_name - Specify the table for which you want to capture and stream a subset of data.

- condition - Specify the subset condition, which is similar to the WHERE clause in a SQL statement.
- column_list - Specify the subset of columns to keep or discard, or specify NULL to keep all of the columns.
- keep - Specify TRUE to keep the columns listed in the column_list parameter, or specify FALSE to discard the columns in the column_list parameter.

  When column_list is non-NULL and keep is set to TRUE, the procedure creates a keep columns declarative rule-based transformation for the columns listed in column_list.

  When column_list is non-NULL and keep is set to FALSE, the procedure creates a delete column declarative rule-based transformation for each column listed in column_list.

3. If subset rules should also be added to the rule set of a capture process or propagation that streams row logical change records (row LCRs) to the outbound server, then see *Oracle Streams Concepts and Administration* for information about adding rules to a rule set.

Example 4–3 Adding Rules That Stream Changes to a Subset of Rows in a Table

The following procedure creates rules that only evaluate to TRUE for row changes where the department_id value is 40 in the hr.employees table:

```
BEGIN
  DBMS_XSTREAM_ADM.ADD_SUBSET_OUTBOUND_RULES(
    server_name => 'xout',
    table_name  => 'hr.employees',
    condition   => 'department_id=40');
END;
/
```

Example 4–4 Adding Rules That Stream Changes to a Subset Rows and Columns in a Table

The following procedure creates rules that only evaluate to TRUE for row changes where the department_id value is 40 for the hr.employees table, and the procedure creates delete column declarative rule-based transformations for the salary and commission_pct columns:

```
BEGIN
  DBMS_XSTREAM_ADM.ADD_SUBSET_OUTBOUND_RULES(
    server_name => 'xout',
    table_name  => 'hr.employees',
    condition   => 'department_id=40',
    column_list => 'salary,commission_pct',
    keep        => FALSE);
END;
/
```

See Also: *Oracle Streams Concepts and Administration*

Removing Rules from an XStream Out Configuration

This section describes removing schema rules, table rules, and subset rules from an XStream Out configuration.

This section contains these topics:

- Removing Schema Rules and Table Rules From an XStream Out Configuration
- Removing Subset Rules From an Outbound Server's Positive Rule Set

Removing Schema Rules and Table Rules From an XStream Out Configuration This section describes removing schema rules and table rules from an XStream Out configuration using the `ALTER_OUTBOUND` procedure in the `DBMS_XSTREAM_ADM` package. The `ALTER_OUTBOUND` procedure removes rules for both data manipulation language (DML) and data definition language (DDL) changes.

When you follow the instructions in this section, the `ALTER_OUTBOUND` procedure always removes rules for the specified schemas and tables from one of the outbound server's rule sets. If the `DBMS_XSTREAM_ADM` package can manage the outbound server's capture process, then the `ALTER_OUTBOUND` procedure also removes rules for the specified schemas and tables from one of the rule sets used by this capture process.

To determine whether the `DBMS_XSTREAM_ADM` package can manage the outbound server's capture process, see "Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM" on page 4-2. If the `DBMS_XSTREAM_ADM` package cannot manage the outbound server's capture process, then the `ALTER_OUTBOUND` procedure removes rules from the outbound server's rule set only. In this case, if rules for same schemas and tables should be removed from the capture process's rule set as well, then see *Oracle Streams Concepts and Administration* for instructions.

To remove schema rules and table rules from an XStream Out configuration, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the `ALTER_OUTBOUND` procedure, and specify the following parameters:

   - `server_name` - Specify the name of the outbound server.
   - `table_names` - Specify the tables for which to remove rules, or specify `NULL` to remove no table rules.
   - `schema_name` - Specify the schemas for which to remove rules, or specify `NULL` to remove no schema rules.
   - `add` - Specify `FALSE` so that the rules are removed. (Rules are added if you specify `TRUE`.)
   - `inclusion_rule` - Specify `TRUE` to remove rules from the positive rule set of the outbound server, or specify `FALSE` to remove rules from the negative rule set of the outbound server. If the `DBMS_XSTREAM_ADM` package can manage the outbound server's capture process, then rules are also removed from this capture process's rule set.

The following examples remove rules from the configuration of an outbound server named `xout`.

Example 4–5 Removing Rules for the hr Schema, oe.orders Table, and oe.order_items Table from the Positive Rule Set

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name     => 'xout',
    table_names     => 'oe.orders, oe.order_items',
    schema_names   => 'hr',
    add            => FALSE,
    inclusion_rule => TRUE);
END;
/
```

Example 4–6 Removing Rules for the hr Schema from the Negative Rule Set

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name    => 'xout',
    table_names    => NULL,
    schema_names   => 'hr',
    add            => FALSE,
    inclusion_rule => FALSE);
END;
/
```

Removing Subset Rules From an Outbound Server's Positive Rule Set This section describes removing subset rules from an outbound server's positive rule set using the REMOVE_SUBSET_OUTBOUND_RULES procedure in the DBMS_XSTREAM_ADM package. The REMOVE_SUBSET_OUTBOUND_RULES procedure only removes rules for data manipulation language (DML) changes. It does not remove rules for data definition language (DDL) changes, and it does not remove rules from a capture process's rule set.

To remove subset rules from an outbound server's positive rule set, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Determine the rule names for the subset rules by running the following query:

   ```
   SELECT RULE_OWNER, SUBSETTING_OPERATION, RULE_NAME
      FROM DBA_XSTREAM_RULES
      WHERE SUBSETTING_OPERATION IS NOT NULL;
   ```

3. Run the REMOVE_SUBSET_OUTBOUND_RULES procedure, and specify the rules to remove from the list of rules displayed in Step 2.

   For example, assume that Step 2 returned the following results:

   ```
   RULE_OWNER                     SUBSET RULE_NAME
   ------------------------------ ------ ------------------------------
   STRMADMIN                      INSERT EMPLOYEES71
   STRMADMIN                      UPDATE EMPLOYEES72
   STRMADMIN                      DELETE EMPLOYEES73
   ```

   To remove these rules from the positive rule set of the xout outbound server, run the following procedure:

   ```
   BEGIN
     DBMS_XSTREAM_ADM.REMOVE_SUBSET_OUTBOUND_RULES(
       server_name      => 'xout',
       insert_rule_name => 'strmadmin.employees71',
       update_rule_name => 'strmadmin.employees72',
       delete_rule_name => 'strmadmin.employees73');
   END;
   /
   ```

4. If subset rules should also be removed from the rule set of a capture process or propagation that streams row logical change records (row LCRs) to the outbound server, then see *Oracle Streams Concepts and Administration* for information about removing rules.

Changing the Connect User for an Outbound Server

A client application can connect to an outbound server as the connect user. This section describes changing the connect user for an outbound server using the ALTER_OUTBOUND procedure in the DBMS_XSTREAM_ADM package.

To change the connect user for an outbound server, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   The Oracle Streams administrator must be granted the DBA role to change the connect user for an outbound server.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the ALTER_OUTBOUND procedure, and specify the new connect user in the connect_user parameter.

   For example, to change the connect user to hr for an outbound server named xout, run the following procedure:

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name  => 'xout',
    connect_user => 'hr');
END;
/
```

Changing the Capture User of the Capture Process for an Outbound Server

A capture user is the user in whose security domain a capture process captures changes from the redo log. This section describes changing the capture user for a capture process that captures changes for an outbound server using the ALTER_OUTBOUND procedure in the DBMS_XSTREAM_ADM package.

To change the connect user for an outbound server, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Determine whether the DBMS_XSTREAM_ADM package can manage the capture process. See "Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM" on page 4-2.

   If the capture process can be managed using the DBMS_XSTREAM_ADM package, then proceed to Step 3.

   If the capture process cannot be managed using the DBMS_XSTREAM_ADM package, then follow the instructions in *Oracle Streams Concepts and Administration*.

3. Run the ALTER_OUTBOUND procedure, and specify the new capture user in the capture_user parameter.

For example, to change the capture user to `hq_admin` for an outbound server named `xout`, run the following procedure:

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name  => 'xout',
    capture_user => 'hq_admin');
END;
/
```

Dropping Oracle Streams Components in an XStream Out Configuration

This section describes dropping an outbound server using the `DROP_OUTBOUND` procedure in the `DBMS_XSTREAM_ADM` package.

This procedure always drops the specified outbound server. This procedure also drops the queue used by the outbound server if both of the following conditions are met:

- The queue was created by the `ADD_OUTBOUND` or `CREATE_OUTBOUND` procedure in the `DBMS_XSTREAM_ADM` package.
- The outbound server is the only subscriber to the queue.

If either one of these conditions is not met, then the `DROP_OUTBOUND` procedure only drops the outbound server. It does not drop the capture process or the queue.

This procedure also drops the capture process for the outbound server if both of the following conditions are met:

- The procedure can drop the outbound server's queue.
- The `DBMS_XSTREAM_ADM` package can manage the outbound server's capture process. See "Checking Whether a Capture Process Can Be Managed Using DBMS_XSTREAM_ADM" on page 4-2.

If the procedure can drop the queue but cannot manage the capture process, then it drops the queue without dropping the capture process.

To drop an outbound server, complete the following steps:

1. Connect to the outbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the `DROP_OUTBOUND` procedure.

   For example, to drop an outbound server named `xout`, run the following procedure:

   ```
   exec DBMS_XSTREAM_ADM.DROP_OUTBOUND('xout');
   ```

See Also: *Oracle Streams Concepts and Administration* for information about dropping a queue or a capture process

Managing XStream In

This section describes managing an XStream inbound server configuration.

This section contains these topics:

- Changing the Apply User for an Inbound Server
- Dropping Oracle Streams Components in an XStream In Configuration

Note: With XStream In, an Oracle Streams apply process functions as an inbound server. Therefore, you can use the instructions for managing an apply process to manage an inbound server. See *Oracle Streams Concepts and Administration*.

See Also:

- "XStream In" on page 1-8
- "Configuring XStream In" on page 3-11
- "Monitoring XStream In" on page 5-4

Changing the Apply User for an Inbound Server

An inbound server applies messages in the security domain of its apply user, and the client application must attach to the inbound server as the apply user. This section describes changing the apply user for an inbound server using the ALTER_INBOUND procedure in the DBMS_XSTREAM_ADM package.

To change the apply user for an inbound server, complete the following steps:

1. Connect to the inbound server database as the Oracle Streams administrator.

   The Oracle Streams administrator must be granted the DBA role to change the apply user for an inbound server.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the ALTER_INBOUND procedure, and specify the new apply user in the apply_user parameter.

   For example, to change the apply user to hr for an inbound server named xin, run the following procedure:

```
BEGIN
  DBMS_XSTREAM_ADM.ALTER_INBOUND(
    server_name => 'xin',
    apply_user  => 'hr');
END;
/
```

See Also:

- "Security Model" on page 7-4 for information about the security requirements for configuring and managing XStream
- *Oracle Streams Concepts and Administration*

Dropping Oracle Streams Components in an XStream In Configuration

This section describes dropping an inbound server using the DROP_INBOUND procedure in the DBMS_XSTREAM_ADM package.

This procedure always drops the specified inbound server. This procedure also drops the queue for the inbound server if both of the following conditions are met:

- One call to the CREATE_INBOUND procedure created the queue.
- The inbound server is the only subscriber to the queue.

If either one of these conditions is not met, then the DROP_INBOUND procedure only drops the inbound server. It does not drop the queue.

To drop an inbound server, complete the following steps:

1. Connect to the inbound server database as the Oracle Streams administrator.

   See *Oracle Database Administrator's Guide* for information about connecting to a database in SQL*Plus.

2. Run the DROP_INBOUND procedure.

   For example, to drop an inbound server named xin, run the following procedure:

   ```
   exec DBMS_XSTREAM_ADM.DROP_INBOUND('xin');
   ```

5 Monitoring XStream

This chapter provides instructions for monitoring XStream.

This chapter contains these topics:

- Monitoring XStream Out
- Monitoring XStream In
- Monitoring XStream Rules
- XStream and the Oracle Streams Performance Advisor

See Also:

- Chapter 1, "XStream Concepts"
- Chapter 2, "XStream Use Cases"
- Chapter 3, "Configuring XStream"
- Chapter 4, "Managing XStream"
- Chapter 6, "Troubleshooting XStream"

Monitoring XStream Out

With XStream Out, an Oracle Streams apply process functions as an outbound server. Therefore, you can use the data dictionary views for apply processes to monitor outbound servers. See *Oracle Streams Concepts and Administration*.

This section provides additional sample queries that you can use to monitor XStream Out.

This section contains these topics:

- Displaying General Information About an Outbound Server
- Displaying Detailed Information About an Outbound Server
- Displaying the Processed Low Position for an Outbound Server
- Determining the Oracle Process ID and Operating System ID of an Outbound Server
- Monitoring XStream Out Components Using General Oracle Streams Views

Displaying General Information About an Outbound Server

You can display the following information for an outbound server by running the query in this section:

- The outbound server name
- The name of the connect user for the outbound server

  The connect user is the user who can attach to the outbound server to retrieve the LCR stream. The client application must attach to the outbound server as the specified connect user.
- The name of the capture user for the capture process that captures changes for the outbound server to process
- The name of the capture process that captures changes for the outbound server to process
- The name of the source database for the captured changes
- The owner of the queue used by the outbound server
- The name of the queue used by the outbound server

To display this general information about an outbound server, run the following query:

```
COLUMN SERVER_NAME HEADING 'Outbound|Server|Name' FORMAT A10
COLUMN CONNECT_USER HEADING 'Connect|User' FORMAT A10
COLUMN CAPTURE_USER HEADING 'Capture|User' FORMAT A10
COLUMN CAPTURE_NAME HEADING 'Capture|Process|Name' FORMAT A11
COLUMN SOURCE_DATABASE HEADING 'Source|Database' FORMAT A11
COLUMN QUEUE_OWNER HEADING 'Queue|Owner' FORMAT A10
COLUMN QUEUE_NAME HEADING 'Queue|Name' FORMAT A10

SELECT SERVER_NAME,
       CONNECT_USER,
       CAPTURE_USER,
       CAPTURE_NAME,
       SOURCE_DATABASE,
       QUEUE_OWNER,
       QUEUE_NAME
  FROM DBA_XSTREAM_OUTBOUND;
```

Your output looks similar to the following:

```
Outbound                         Capture
Server     Connect    Capture    Process     Source      Queue      Queue
Name       User       User       Name        Database    Owner      Name
---------- ---------- ---------- ----------- ----------- ---------- ----------
XOUT       STRMADMIN  STRMADMIN  CAP$_XOUT_1 DB.EXAMPLE. STRMADMIN  Q$_XOUT_2
                                             COM
```

The DBA_XSTREAM_OUTBOUND view contains information about the capture user, the capture process, and the source database in either of the following cases:

- The outbound server was created using the CREATE_OUTBOUND procedure in the DBMS_XSTREAM_ADM package.
- The outbound server was created using the ADD_OUTBOUND procedure in the DBMS_XSTREAM_ADM package, and the capture process for the outbound server runs on the same database as the outbound server.

If the outbound server was created using the ADD_OUTBOUND procedure in this package, and the capture process for the outbound server is on a different database, then the DBA_XSTREAM_OUTBOUND view does not contain information about the capture user, the capture process, or the source database.

See Also: *Oracle Streams Concepts and Administration*

Displaying Detailed Information About an Outbound Server

You can monitor an outbound server using the same queries as you use to monitor an Oracle Streams apply process. See *Oracle Streams Concepts and Administration* for instructions.

The ALL_APPLY and DBA_APPLY views show "XStream Out" in the PURPOSE column for an apply process that is functioning as an outbound server. For example, run the following query:

```
COLUMN APPLY_NAME HEADING 'Apply Name' FORMAT A10
COLUMN STATUS HEADING 'Status' FORMAT A8
COLUMN PURPOSE HEADING 'Purpose' FORMAT A20
COLUMN ERROR_NUMBER HEADING 'Error Number' FORMAT 9999999
COLUMN ERROR_MESSAGE HEADING 'Error Message' FORMAT A20

SELECT APPLY_NAME,
       STATUS,
       PURPOSE,
       ERROR_NUMBER,
       ERROR_MESSAGE
  FROM DBA_APPLY;
```

Your output looks similar to the following:

```
Apply Name Status   Purpose              Error Number Error Message
---------- -------- -------------------- ------------ --------------------
XOUT       ENABLED  XSTREAM OUT
```

This output shows that XOUT is an apply process that is functioning as an outbound server.

Displaying the Processed Low Position for an Outbound Server

For an outbound server, the processed low position is a position below which all transactions have been committed and logged by the client application. The processed low position is important when the outbound server or the client application is restarted.

You can display the following information about the processed low position for an outbound server by running the query in this section:

- The outbound server name
- The name of the source database for the captured changes
- The processed low position, which indicates the low watermark position processed by the client application
- Time when the processed low position was last updated by the outbound server

```
COLUMN SERVER_NAME HEADING 'Outbound|Server|Name' FORMAT A10
COLUMN SOURCE_DATABASE HEADING 'Source|Database' FORMAT A20
COLUMN PROCESSED_LOW_POSITION HEADING 'Processed|Low LCR|Position' FORMAT A30
COLUMN PROCESSED_LOW_TIME HEADING 'Processed|Low|Time' FORMAT A9
```

```
SELECT SERVER_NAME,
       SOURCE_DATABASE,
       PROCESSED_LOW_POSITION,
       TO_CHAR(PROCESSED_LOW_TIME,'HH24:MI:SS MM/DD/YY') PROCESSED_LOW_TIME
FROM DBA_XSTREAM_OUTBOUND_PROGRESS;
```

Your output looks similar to the following:

```
Outbound                        Processed                      Processed
Server     Source               Low LCR                        Low
Name       Database             Position                       Time
---------- -------------------- ------------------------------ ---------
XOUT       DB.EXAMPLE.COM       00000008F17A000000000000000000 13:39:01
                                000008F17A000000000000000001   07/15/09
```

See Also: "The Processed Low Position and Restartability for XStream Out" on page 1-11

Determining the Oracle Process ID and Operating System ID of an Outbound Server

An outbound server is an Oracle background process. When a client application attaches to an outbound server, the Oracle process ID and operating system process ID of the outbound server process is recorded in the alert log. A message similar to the following records these IDs:

```
XStream Outbound Server for XOUT attached with pid=30 OS id=12873
```

In this example, the Oracle process ID is `30` and the operating system process ID is `12873`.

Note: The `SID` and `SERIAL#` columns in the `V$STREAMS_APPLY_SERVER` view identify the client application that attaches to the outbound server.

Monitoring XStream Out Components Using General Oracle Streams Views

With XStream Out, an Oracle Streams apply process functions as an outbound server. Therefore, you can use the data dictionary views for apply processes to monitor outbound servers. In addition, an XStream Out environment includes capture processes and queues, and might include other Oracle Streams components, such as propagations, rules, and rule-based transformations.

See Also: *Oracle Streams Concepts and Administration*

Monitoring XStream In

With XStream In, an Oracle Streams apply process functions as an inbound server. Therefore, you can use the data dictionary views for apply processes to monitor inbound servers. See *Oracle Streams Concepts and Administration*.

This section provides additional sample queries that you can use to monitor XStream In.

This section contains these topics:

- Displaying General Information About an Inbound Server
- Displaying Detailed Information About an Inbound Server
- Displaying the Position Information for an Inbound Server
- Determining the Process ID and Operating System ID of a Propagation Receiver

Displaying General Information About an Inbound Server

You can display the following information for an inbound server by running the query in this section:

- The inbound server name
- The owner of the queue used by the inbound server
- The name of the queue used by the inbound server
- The apply user for the inbound server

To display this general information about an inbound server, run the following query:

```
COLUMN SERVER_NAME HEADING 'Inbound Server Name' FORMAT A20
COLUMN QUEUE_OWNER HEADING 'Queue Owner' FORMAT A15
COLUMN QUEUE_NAME HEADING 'Queue Name' FORMAT A15
COLUMN APPLY_USER HEADING 'Apply User' FORMAT A15

SELECT SERVER_NAME,
       QUEUE_OWNER,
       QUEUE_NAME,
       APPLY_USER
  FROM DBA_XSTREAM_INBOUND;
```

Your output looks similar to the following:

```
Inbound Server Name  Queue Owner     Queue Name      Apply User
-------------------- --------------- --------------- ---------------
XIN                  STRMADMIN       XQUEUE          STRMADMIN
```

Displaying Detailed Information About an Inbound Server

You can monitor an inbound server using the same queries as you use to monitor an Oracle Streams apply process. See *Oracle Streams Concepts and Administration* for instructions.

The `ALL_APPLY` and `DBA_APPLY` views show "XStream In" in the `PURPOSE` column for an apply process that is functioning as an inbound server. For example, run the following query:

```
COLUMN APPLY_NAME HEADING 'Apply Name' FORMAT A10
COLUMN STATUS HEADING 'Status' FORMAT A8
COLUMN PURPOSE HEADING 'Purpose' FORMAT A20
COLUMN ERROR_NUMBER HEADING 'Error Number' FORMAT 9999999
COLUMN ERROR_MESSAGE HEADING 'Error Message' FORMAT A20

SELECT APPLY_NAME,
       STATUS,
       PURPOSE,
       ERROR_NUMBER,
       ERROR_MESSAGE
  FROM DBA_APPLY;
```

Your output looks similar to the following:

```
Apply Name Status   Purpose              Error Number Error Message
---------- -------- -------------------- ------------ --------------------
XIN        ENABLED  XSTREAM IN
```

This output shows that `XIN` is an apply process that is functioning as an inbound server.

Displaying the Position Information for an Inbound Server

For an inbound server, you can view position information by querying the `DBA_XSTREAM_INBOUND_PROGRESS` view. Specifically, you can display the following position information by running the query in this section:

- The inbound server name
- The applied low position for the inbound server
- The spill position for the inbound server
- The applied high position for the inbound server
- The processed low position for the inbound server

Run the following query to view this information:

```
COLUMN SERVER_NAME HEADING 'Inbound|Server|Name' FORMAT A10
COLUMN APPLIED_LOW_POSITION HEADING 'Applied Low|Position' FORMAT A15
COLUMN SPILL_POSITION HEADING 'Spill Position' FORMAT A15
COLUMN APPLIED_HIGH_POSITION HEADING 'Applied High|Position' FORMAT A15
COLUMN PROCESSED_LOW_POSITION HEADING 'Processed Low|Position' FORMAT A15

SELECT SERVER_NAME,
       APPLIED_LOW_POSITION,
       SPILL_POSITION,
       APPLIED_HIGH_POSITION,
       PROCESSED_LOW_POSITION
  FROM DBA_XSTREAM_INBOUND_PROGRESS;
```

Your output looks similar to the following:

```
Inbound
Server     Applied Low                     Applied High    Processed Low
Name       Position        Spill Position  Position        Position
---------- --------------- --------------- --------------- ---------------
XIN        C10A            C11D            C10A            C11D
```

The values of the positions shown in the output were set by the client application that attaches to the inbound server. However, the inbound server determines which values are the current applied low position, spill position, applied high position, and processed low position.

See Also: "Position of LCRs and XStream In" on page 1-12

Determining the Process ID and Operating System ID of a Propagation Receiver

A propagation receiver gets the logical change records (LCRs) sent by a client application and passes the LCRs to an inbound server. When a client application attaches to an inbound server, the process ID and operating system ID of the propagation receiver is recorded in the alert log. A message similar to the following records the process ID and operating system ID:

```
Propagation Receiver (CCA) for XStream In From_XOUT and Apply XIN with pid=23,
OS id=1980, objnum=0 started.
```

In this example, the process ID is `23` and the operating system ID is `1980`.

Monitoring XStream Rules

The `ALL_XSTREAM_RULES` and `DBA_XSTREAM_RULES` views contain information about the rules used by outbound servers and inbound servers. If an outbound server was created using the `CREATE_OUTBOUND` procedure in the `DBMS_XSTREAM_ADM` package, then these views also contain information about the rules used by the capture process that sends changes to the outbound server. However, if an outbound server was created using the `ADD_OUTBOUND` procedure, then these views do not contain information about the capture process rules. Also, these views do not contain information about the rules used by any propagation in the stream from a capture process to an outbound server.

To view information about the rules used by all Oracle Streams components, including capture processes, propagations, apply processes, outbound servers, and inbound servers, you can query the `ALL_STREAMS_RULES` and `DBA_STREAMS_RULES` views. See *Oracle Streams Concepts and Administration* for sample queries that enable you to monitor rules.

Run the following query to display this information:

```
COLUMN STREAMS_NAME HEADING 'Oracle|Streams|Name' FORMAT A12
COLUMN STREAMS_TYPE HEADING 'Oracle|Streams|Type' FORMAT A11
COLUMN RULE_NAME HEADING 'Rule|Name' FORMAT A10
COLUMN RULE_SET_TYPE HEADING 'Rule Set|Type' FORMAT A8
COLUMN STREAMS_RULE_TYPE HEADING 'Oracle|Streams|Rule|Level' FORMAT A7
COLUMN SCHEMA_NAME HEADING 'Schema|Name' FORMAT A6
COLUMN OBJECT_NAME HEADING 'Object|Name' FORMAT A11
COLUMN RULE_TYPE HEADING 'Rule|Type' FORMAT A4

SELECT STREAMS_NAME,
       STREAMS_TYPE,
       RULE_NAME,
       RULE_SET_TYPE,
       STREAMS_RULE_TYPE,
       SCHEMA_NAME,
       OBJECT_NAME,
       RULE_TYPE
  FROM DBA_XSTREAM_RULES;
```

Your output looks similar to the following:

```
Oracle       Oracle                          Streams
Streams      Streams     Rule       Rule Set Rule    Schema Object      Rule
Name         Type        Name       Type     Level   Name   Name        Type
------------ ----------- ---------- -------- ------- ------ ----------- ----
CAP$_XOUT_49 CAPTURE     DB52       POSITIVE GLOBAL                     DML
CAP$_XOUT_49 CAPTURE     DB53       POSITIVE GLOBAL                     DDL
XOUT         APPLY       DB55       POSITIVE GLOBAL                     DML
XOUT         APPLY       DB56       POSITIVE GLOBAL                     DDL
```

Notice that the `STREAMS_TYPE` is `APPLY` even though the rules are in the positive rule set for the outbound server `xout`. You can determine the purpose of an apply process by querying the `PURPOSE` column in the `DBA_APPLY` view.

XStream and the Oracle Streams Performance Advisor

The Oracle Streams Performance Advisor consists of the `DBMS_STREAMS_ADVISOR_ADM` PL/SQL package and a collection of data dictionary views. The Oracle Streams Performance Advisor enables you to monitor the topology and performance of an XStream environment. The XStream topology includes information about the components in an XStream environment, the links between the components, and the way information flows from capture to consumption. The Oracle Streams Performance Advisor also provides information about how Oracle Streams components are performing.

Apply processes function as XStream outbound servers and inbound servers. In general, the Oracle Streams Performance Advisor works the same way for an Oracle Streams environment with apply processes and an XStream environment with outbound servers or inbound servers. This section describes important considerations about using the Oracle Streams Performance Advisor in an XStream environment.

> **See Also:** *Oracle Streams Concepts and Administration* for detailed information about using the Oracle Streams Performance Advisor

XStream Components

The Oracle Streams Performance Advisor tracks the following types of components in an XStream environment:

- `QUEUE`
- `CAPTURE`
- `PROPAGATION SENDER`
- `PROPAGATION RECEIVER`
- `APPLY`

The following types are the same in an Oracle Streams environment and an XStream environment: `QUEUE`, `CAPTURE`, `PROPAGATION SENDER`, and `PROPAGATION RECEIVER`.

The `APPLY` component type can be an XStream outbound server or inbound server. The following subcomponent types are possible for apply processes, outbound servers, and inbound servers:

- `PROPAGATION SENDER+RECEIVER` for sending LCRs from a capture process directly to an apply process in a combined capture and apply optimization
- `APPLY READER` for a reader server
- `APPLY COORDINATOR` for a coordinator process
- `APPLY SERVER` for a reader server

In addition, the Oracle Streams Performance Advisor identifies a bottleneck component as the busiest component or the component with the least amount of idle time. In an XStream configuration, the XStream client application might be the bottleneck when `EXTERNAL` appears in the `ACTION_NAME` column of the `DBA_STREAMS_TP_PATH_BOTTLENECK` view.

Topology and Stream Paths

In the Oracle Streams topology, a stream path is a flow of messages from a source to a destination. A stream path begins where a capture process, synchronous capture, or application enqueues messages into a queue. A stream path ends where an apply process, outbound server, or inbound server dequeues the messages. The stream path might flow through multiple queues and propagations before it reaches an apply process, outbound server, or inbound server. Therefore, a single stream path can consist of multiple source/destination component pairs before it reaches last component.

The Oracle Streams topology only gathers information about a stream path if the stream path ends with an apply process, an outbound server, or an inbound server. The Oracle Streams topology does not track stream paths that end when a messaging client or an application dequeues messages.

XStream and Component-Level Statistics

The Oracle Streams Performance Advisor tracks the following component-level statistics:

- The `MESSAGE APPLY RATE` is the average number of messages applied each second by the apply process, outbound server, or inbound server.
- The `TRANSACTION APPLY RATE` is the average number of transactions applied by the apply process each second. Transactions typically include multiple messages.

A logical change record (LCR) can be applied in one of the following ways:

- An apply process or inbound server makes the change encapsulated in the LCR to a database object.
- An apply process or inbound server passes the LCR to an apply handler.
- An outbound server passes the LCR to an XStream client application. If the LCR raises an error, then the outbound server also reports the error to the client application.
- If the LCR raises an error, then an apply process or inbound server sends the LCR to the error queue.

Also, the Oracle Streams Performance Advisor tracks the `LATENCY` component-level statistics. `LATENCY` is defined in the following ways:

- For apply processes, the `LATENCY` is the amount of time between when the message was created at a source database and when the message was applied by the apply process at the destination database.
- For outbound servers, the `LATENCY` is amount of time between when the message was created at a source database and when the message was sent to the client application.
- For inbound servers, the `LATENCY` is amount of time between when the message was created by the XStream client application and when the message was applied by the apply process at the destination database.

When a capture process creates an LCR, the message creation time is the time when the redo entry for the database change was recorded. When an XStream client application creates an LCR, the message creation time is the time when the LCR was constructed.

See Also: *Oracle Streams Concepts and Administration* for more information about component-level statistics

The UTL_SPADV Package

The `UTL_SPADV` package provides subprograms to collect and analyze statistics for the XStream components in a distributed database environment. The package uses the Oracle Streams Performance Advisor to gather statistics, and the output is formatted so that it can be imported into a spreadsheet easily and analyzed.

The `UTL_SPADV` package works the same way for an Oracle Streams environment with apply processes and an XStream environment with outbound servers or inbound servers. However, there are some differences in the output for the `SHOW_STATS` procedure.

The following sections describe the output for the `SHOW_STATS` procedure:

- Sample Output When an Outbound Server Is the Last Component in a Path
- Sample Output When an Inbound Server Is the Last Component in a Path

See Also: *Oracle Streams Concepts and Administration* for detailed information about using the `UTL_SPADV` package

Sample Output When an Outbound Server Is the Last Component in a Path

The following is sample output for when an outbound server is the last component in a path:

```
OUTPUT
PATH 1 RUN_ID 2 RUN_TIME 2009-MAY-15 12:20:55 CCA Y
|<C> CAP$_XOUT_1 2733 2730 3392 LMR 8.3% 91.7% 0% "" LMP (1) 8.3% 91.7% 0% ""
LMB 8.3% 91.7% 0% "" CAP 8.3% 91.7% 0% "" |<Q> "STRMADMIN"."Q$_XOUT_2" 2730 0.01
4109 |<A> XOUT 2329 2.73 0 -1 PS+PR 8.3% 91.7% 0% "" APR 8.3% 91.7% 0% "" APC
100% 0% 0% "" APS (1) 8.3% 83.3% 8.3% "" |<B> "EXTERNAL"
.
.
.
```

Note: This output is for illustrative purposes only. Actual performance characteristics vary depending on individual configurations and conditions.

In this output, the `A` component is the outbound server `XOUT`. The output for when an outbound server is the last component in a path is similar to the output for when an apply process is the last component in a path. However, the apply server (APS) is not the last component because the outbound server connects to a client application. Statistics are not collected for the client application.

In an XStream Out configuration, the output can indicate flow control for the network because "SQL*Net more data to client" for an apply server is considered as a flow control event. If the output indicates flow control for an apply server, then either the network or the client application is considered the bottleneck component. In the previous output, `EXTERNAL` indicates that either the network or the client application is the bottleneck.

Other than these differences, you can interpret the statistics in the same way that you would for a path that ends with an apply process. Use the legend and the abbreviations to determine the statistics in the output.

Sample Output When an Inbound Server Is the Last Component in a Path

The following is sample output for when an inbound server is the last component in a path:

```
OUTPUT
PATH 1 RUN_ID 2 RUN_TIME 2009-MAY-16 10:11:38 CCA N
|<PR> "clientcap"=> 75% 0% 8.3% "CPU + Wait for CPU" |<Q> "STRMADMIN"."QUEUE2"  467 0.01 1
|<A> XIN 476 4.71 0 APR 100% 0% 0% "" APC 100% 0% 0% "" APS (4) 366.7% 0% 33.3% "CPU + Wait for CPU"
|<B> "EXTERNAL"
.
.
.
```

Note: This output is for illustrative purposes only. Actual performance characteristics vary depending on individual configurations and conditions.

In this output, the `A` component is the inbound server `XIN`. When an inbound server is the last component in a path, the XStream client application connects to the inbound server, and the inbound server applies the changes in the LCRs. The client application is not shown in the output.

The propagation receiver receives the LCRs from the client application. So, the propagation receiver is the first component shown in the output. In the previous sample output, the propagation receiver is named `"clientcap"=>`. In this case, `clientcap` is the source name given by the client application when it attaches to the inbound server.

If the propagation receiver is idle for a significant percentage of time, then either the network or the client application is considered a bottleneck component. In the previous output, `EXTERNAL` indicates that either the network or the client application is the bottleneck.

Other than these differences, you can interpret the statistics in the same way that you would for a path that ends with an apply process. Use the legend and the abbreviations to determine the statistics in the output.

6 Troubleshooting XStream

With XStream, an Oracle Streams apply process can function as an outbound server or an inbound server. Therefore, you can troubleshoot outbound servers and inbound servers in the same way that you troubleshoot apply processes. In addition, an XStream Out environment includes capture processes and queues, and might include other Oracle Streams components, such as propagations, rules, and rule-based transformations. To troubleshoot these components, see the troubleshooting documentation in *Oracle Streams Concepts and Administration*.

The following topics describe troubleshooting problems that are related specifically to XStream:

- Is an OCI Client Application Unable to Attach to the Outbound Server?
- Are Changes Failing to Reach the Client Application in XStream Out?
- Are LCRs Streaming from an Outbound Server Missing Extra Attributes?
- Is the Client Application Unresponsive?
- Is XStream In Unable to Identify an Inbound Server?

See Also:

- Chapter 1, "XStream Concepts"
- Chapter 2, "XStream Use Cases"
- Chapter 3, "Configuring XStream"
- Chapter 4, "Managing XStream"
- Chapter 5, "Monitoring XStream"

Is an OCI Client Application Unable to Attach to the Outbound Server?

If a client application cannot attach to an outbound server using the Oracle Call Interface (OCI) `OCIXStreamOutAttach()` function, then the following conditions might be causing the problem:

- The client application is not connected to the outbound server's database as the outbound server's connect user. The client application must connect to the database as the connect user before attaching to the outbound server.
- The client application is not passing a service handle to the outbound server. Ensure that the client application is passing a service handle using the `OCISvcCtx` and not `OCIServer`.

See Also:

- "XStream and Security" on page 1-2
- "OCIXStreamOutAttach()" on page 12-56

Are Changes Failing to Reach the Client Application in XStream Out?

In an XStream Out configuration, if database changes that should be captured and streamed to the client application are not reaching the client application, then the capture process might have fallen behind, or there might be problems with rules or rule-based transformations.

Checking Whether the Capture Process Has Fallen Behind

To determine the time when the capture process last created a logical change record (LCR) and the time when the capture process last enqueued an LCR, run the following query:

```
COLUMN CAPTURE_NAME HEADING 'Capture|Name' FORMAT A15
COLUMN CREATE_MESSAGE HEADING 'Last LCR|Create Time'
COLUMN ENQUEUE_MESSAGE HEADING 'Last|Enqueue Time'

SELECT CAPTURE_NAME,
       TO_CHAR(CAPTURE_MESSAGE_CREATE_TIME, 'HH24:MI:SS MM/DD/YY') CREATE_MESSAGE,
       TO_CHAR(ENQUEUE_MESSAGE_CREATE_TIME, 'HH24:MI:SS MM/DD/YY') ENQUEUE_MESSAGE
  FROM V$STREAMS_CAPTURE;
```

If the times returned are before the time when the database changes were made, then the capture process must catch up and capture the changes. Normally, the capture process will catch up on its own without the need for intervention.

See Also: *Oracle Streams Replication Administrator's Guide*

Checking for Problems with Rules and Rule-Based Transformations

Rules determine which logical change records (LCRs) are captured by a capture process, sent from a source queue to a destination queue by a propagation, and sent to a client application by an outbound server. If the rules are not configured properly, then the client application might not receive the LCRs it should receive. The client application might also receive LCRs that it should not receive.

Rule-based transformations modify the contents of LCRs. Therefore, if the expected change data is not reaching the client application, it might be because a rule-based transformation modified the data or deleted the data. For example, a `DELETE_COLUMN` declarative rule-based transformation removes a column from an LCR.

Check the rules and rule-based transformations that are configured for each component in the stream from the capture process to the client application, and correct any problems.

See Also: *Oracle Streams Concepts and Administration*

Tracking LCRs Through a Stream

If the capture process has not fallen behind, and there are no problems with rules or rule-based transformations, then LCRs might be blocked in the stream for some other reason. For example, a propagation or outbound server might be disabled, a database link might be broken, or there might be another problem.

You can track an LCR through a stream using one of the following methods:

- Setting the `message_tracking_frequency` capture process parameter to `1` or another relatively low value
- Running the `SET_MESSAGE_TRACKING` procedure in the `DBMS_STREAMS_ADM` package

After using one of these methods, use the `V$STREAMS_MESSAGE_TRACKING` view to monitor the progress of LCRs through a stream. By tracking an LCR through the stream, you can determine where the LCR is blocked.

See Also:

- *Oracle Streams Replication Administrator's Guide* for more information about tracking LCRs through a stream
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `message_tracking_frequency` capture process parameter

Are LCRs Streaming from an Outbound Server Missing Extra Attributes?

Logical change records (LCRs) can contain the following extra attributes related to database changes:

- `row_id`
- `serial#`
- `session#`
- `thread#`
- `tx_name`
- `username`

By default, a capture process does not capture these extra attributes. If you want extra attributes to be included in LCRs streamed from an outbound server to a client application, but the LCRs do not contain values for extra attributes, then make sure the capture process that captures changes for the outbound server is configured to capture values for the extra attributes.

To configure a capture process to capture one or more of these extra attributes, use the `INCLUDE_EXTRA_ATTRIBUTE` procedure in the `DBMS_CAPTURE_ADM` package.

See Also: *Oracle Streams Concepts and Administration*

Is the Client Application Unresponsive?

If the client application is unresponsive, then the Streams pool size might be too small. Run the following query at the database that contains the outbound server or inbound server:

```
SELECT STATE FROM V$PROPAGATION_RECEIVER;
```

If the state is `WAITING FOR MEMORY`, then consider increasing the Streams pool size.

You can also run the following query:

```
SELECT TOTAL_MEMORY_ALLOCATED/CURRENT_SIZE FROM V$STREAMS_POOL_STATISTICS;
```

If the value returned is 0.90 or greater, then consider increasing the Streams pool size.

For an outbound server that receives changes from a capture process that is running on the same database, you can also run the following query:

```
SELECT STATE FROM V$STREAMS_CAPTURE;
```

If the state is `WAITING FOR BUFFER QUEUE TO SHRINK`, then increase the Streams pool size.

You can increase the Streams pool size by modifying the `STREAMS_POOL_SIZE` initialization parameter, or by modifying other initialization parameters related to memory.

Otherwise, if there is enough memory in the Streams pool, then check your client application for programming errors.

See Also: *Oracle Streams Replication Administrator's Guide*

Is XStream In Unable to Identify an Inbound Server?

If an XStream In configuration cannot identify an inbound server, then the following error is returned:

```
ORA-26840: STREAMS unable to identify an apply for the source database "%s"
```

This error indicates that there are multiple subscribers to the queue used by the inbound server. Subscribers can include inbound servers, outbound servers, apply processes, and propagations.

To determine whether there are multiple subscribers to the inbound server's queue, run the following query on the inbound server's database:

```
SELECT APPLY_NAME SUBSCRIBER, QUEUE_NAME FROM DBA_APPLY UNION
  SELECT PROPAGATION_NAME, SOURCE_QUEUE_NAME
  FROM DBA_PROPAGATION
  ORDER BY QUEUE_NAME;
```

You can add a `WHERE` clause to the query to limit the output to the inbound server's queue.

If the query returns more than one subscriber to the inbound server's queue, then reconfigure the subscribers so that the inbound server is the only subscriber.

See Also: Chapter 3, "Configuring XStream"

Part III

XStream PL/SQL Packages and Types Reference

This part contains the XStream PL/SQL packages and types reference. This part contains the following chapters:

7 DBMS_XSTREAM_ADM

This `DBMS_XSTREAM_ADM` package, one of a set of Oracle Streams packages, provides interfaces for streaming database changes between an Oracle database and another system. XStream is a programmatic interface to Oracle Streams. XStream enables applications to stream out or stream in database changes.

This chapter contains the following topic:

See Also:

- Chapter 1, "XStream Concepts"
- Part IV, "XStream OCI API Reference"
- *Oracle Database XStream Java API Reference*

Using DBMS_XSTREAM_ADM

This section contains topics which relate to using the `DBMS_XSTREAM_ADM` package.

- Overview
- Security Model
- Operational Notes

Overview

The package provides interfaces for configuring outbound servers that stream database changes from an Oracle database to another system. The package also provides interfaces for configuring inbound servers that stream database changes from another system to an Oracle database.

In both cases, the database changes are encapsulated in logical change records (LCRs). Also, the other system can be an Oracle system or a non-Oracle system, such as a non-Oracle database.

XStream outbound servers can stream out LCRs from an Oracle database programmatically using C or Java. After receiving the LCRs, the other system can save the contents of the LCRs to a file, send the LCRs to an Oracle database through an XStream inbound server, or generate SQL statements and execute them on any Oracle or non-Oracle databases.

XStream inbound servers accept LCRs from another system and either apply them to an Oracle database or process them in a customized way.

See Also: Chapter 1, "XStream Concepts"

Security Model

To ensure that the user who runs the subprograms in this package has the necessary privileges, configure an Oracle Streams administrator and connect as the Oracle Streams administrator when using this package.

An administrator must be granted the `DBA` role when the administrator is performing any of the following actions:

- Running the `ADD_OUTBOUND` procedure while connected as a user that is different than the configured connect user for an outbound server
- Running the `ALTER_OUTBOUND` procedure to change the capture user for a capture process or the connect user for an outbound server
- Running the `CREATE_OUTBOUND` procedure because this procedure creates a capture process
- Running the `ALTER_INBOUND` procedure to change the apply user for an inbound server
- Running the `ADD_INBOUND` procedure while connected as a user that is different than the configured apply user for an inbound server

When the administrator does not need to perform these tasks, `DBA` role is not required.

See Also:

- *Oracle Streams Replication Administrator's Guide* for information about configuring an Oracle Streams administrator
- "XStream and Security" on page 1-2 for more information about XStream and security

Operational Notes

Some subprograms in the DBMS_APPLY_ADM package can manage XStream outbound servers, and some subprograms in the DBMS_APPLY_ADM package can manage XStream inbound servers.

See Also: Chapter 8, "Addendum To DBMS_APPLY_ADM" for details about which subprograms can manage outbound servers and inbound servers

Summary of DBMS_XSTREAM_ADM Subprograms

Table 7–1 DBMS_XSTREAM_ADM Package Subprograms

Subprogram	Description
ADD_OUTBOUND Procedure on page 7-7	Creates an XStream outbound server that dequeues logical change records (LCRs) from the specified queue
ADD_SUBSET_OUTBOUND_RULES Procedure on page 7-10	Adds subset rules to an outbound server configuration
ALTER_INBOUND Procedure on page 7-12	Modifies an XStream inbound server
ALTER_OUTBOUND Procedure on page 7-13	Modifies an XStream outbound server
CREATE_INBOUND Procedure on page 7-16	Creates an XStream inbound server and its queue
CREATE_OUTBOUND Procedure on page 7-18	Creates an XStream outbound server, queue, and capture process to enable client applications to stream out Oracle database changes encapsulated in logical change records (LCRs)
DROP_INBOUND Procedure on page 7-22	Removes an inbound server configuration
DROP_OUTBOUND Procedure on page 7-23	Removes an outbound server configuration
REMOVE_SUBSET_OUTBOUND_RULES Procedure on page 7-24	Removes subset rules from an outbound server configuration

Note: All subprograms commit unless specified otherwise.

ADD_OUTBOUND Procedure

This procedure creates an XStream outbound server that dequeues logical change records (LCRs) from the specified queue. The outbound server streams out the LCRs to a client application.

This procedure creates neither a capture process nor a queue. To create an outbound server, a capture process, and a queue with one procedure call, use the CREATE_OUTBOUND Procedure.

To create the capture process individually, use one of the following packages:

- `DBMS_STREAM_ADM`
- `DBMS_CAPTURE_ADM`

To create a queue individually, use the `SET_UP_QUEUE` procedure in the `DBMS_STREAMS_ADM` package.

This procedure is overloaded. One `table_names` parameter is type `VARCHAR2` and the other `table_names` parameters is type `DBMS_UTILITY.UNCL_ARRAY`. Also, one `schema_names` parameter is type `VARCHAR2` and the other `schema_names` parameters is type `DBMS_UTILITY.UNCL_ARRAY`. These parameters enable you to enter the lists of tables and schemas in different ways and are mutually exclusive.

Note:

- Only one client application at a time can attach to an outbound server. See Part IV, "XStream OCI API Reference" and *Oracle Database XStream Java API Reference* for information about attaching to an outbound server.
- This procedure enables the outbound server that it creates.

Syntax

```
DBMS_XSTREAM_ADM.ADD_OUTBOUND(
   server_name     IN  VARCHAR2,
   queue_name      IN  VARCHAR2,
   source_database IN  VARCHAR2  DEFAULT NULL,
   table_names     IN  DBMS_UTILITY.UNCL_ARRAY,
   schema_names    IN  DBMS_UTILITY.UNCL_ARRAY,
   connect_user    IN  VARCHAR2  DEFAULT NULL,
   comment         IN  VARCHAR2  DEFAULT NULL);

DBMS_XSTREAM_ADM.ADD_OUTBOUND(
   server_name     IN VARCHAR2,
   queue_name      IN VARCHAR2,
   source_database IN VARCHAR2  DEFAULT NULL,
   table_names     IN VARCHAR2  DEFAULT NULL,
   schema_names    IN VARCHAR2  DEFAULT NULL,
   connect_user    IN VARCHAR2  DEFAULT NULL,
   comment         IN VARCHAR2  DEFAULT NULL);
```

Parameters

Table 7–2 ADD_OUTBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the outbound server being created. A `NULL` specification is not allowed. Do not specify an owner. The specified name must not match the name of an existing outbound server, inbound server, apply process, or messaging client. **Note:** The `server_name` setting cannot be altered after the outbound server is created.
`queue_name`	The name of the local queue from which the outbound server dequeues logical change records (LCRs), specified as `[`*`schema_name`*`.]`*`queue_name`*. The current database must contain the queue, and the queue must be `ANYDATA` type. For example, to specify a queue named `streams_queue` in the `strmadmin` schema, enter `strmadmin.streams_queue` for this parameter. If the schema is not specified, then the current user is the default.
`source_database`	The global name of the source database. The source database is where the changes being captured originated. If you do not include the domain name, then the procedure appends it to the database name automatically. For example, if you specify `DBS1` and the domain is `.EXAMPLE.COM`, then the procedure specifies `DBS1.EXAMPLE.COM` automatically. If `NULL`, then this procedure does not add a condition regarding the source database to the generated rules. Otherwise, a condition regarding the source database is added.
`table_names`	The tables for which data manipulation language (DML) and data definition language (DDL) changes are streamed out to the XStream Out client application. The tables can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a table. The first table should be in position 1. The last position must be `NULL`. Each table should be specified as `[`*`schema_name`*`.]`*`table_name`*. For example, `hr.employees`. If the schema is not specified, then the current user is the default. **See Also:** "Usage Notes" on page 7-9 for more information about this parameter.
`schema_names`	The schemas for which DML and DDL changes are streamed out to the XStream Out client application. The schemas can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a schema. The first schema should be in position 1. The last position must be `NULL`. **See Also:** "Usage Notes" on page 7-9 for more information about this parameter.

Table 7–2 (Cont.) ADD_OUTBOUND Procedure Parameters

Parameter	Description
`connect_user`	The user who can attach to the specified outbound server to retrieve the LCR stream. The client application must attach to the outbound server as the specified connect user. See "CREATE_OUTBOUND Procedure" on page 7-18 for information about the privileges required by a connect user. If `NULL`, then the current user is the default.
`comment`	An optional comment associated with the outbound server.

Usage Notes

The following list describes the behavior of the outbound server for various combinations of the `table_names` and `schema_names` parameters:

- If both the `table_names` and `schema_names` parameters are `NULL`, then the outbound server streams all DML and DDL changes to the client application.
- If both the `table_names` and `schema_names` parameters are specified, then the outbound server streams DML and DDL changes for the specified tables and schemas.
- If the `table_names` parameter is specified and the `schema_names` parameter is `NULL`, then the outbound server streams DML and DDL changes for the specified tables.
- If the `table_names` parameter is `NULL` and the `schema_names` parameter is specified, then the outbound server streams DML and DDL changes for the specified schema.

ADD_SUBSET_OUTBOUND_RULES Procedure

This procedure adds subset rules to an outbound server configuration. Subset rules instruct the outbound server to stream out a subset of the changes in the specified tables. Outbound servers can stream out a subset of both rows and columns.

This procedure is overloaded. One `column_list` parameter is type `VARCHAR2` and the other `column_list` parameters is type `DBMS_UTILITY.LNAME_ARRAY`. These parameters enable you to enter the list of columns in different ways and are mutually exclusive.

Note: This procedure does not add rules to the outbound server's capture process.

Syntax

```
DBMS_XSTREAM_ADM.ADD_SUBSET_OUTBOUND_RULES(
   server_name IN VARCHAR2,
   table_name  IN VARCHAR2,
   condition   IN VARCHAR2,
   column_list IN DBMS_UTILITY.LNAME_ARRAY,
   keep        IN BOOLEAN   DEFAULT TRUE);

DBMS_XSTREAM_ADM.ADD_SUBSET_OUTBOUND_RULES(
   server_name IN VARCHAR2,
   table_name  IN VARCHAR2,
   condition   IN VARCHAR2,
   column_list IN VARCHAR2  DEFAULT NULL,
   keep        IN BOOLEAN   DEFAULT TRUE);
```

Parameters

Table 7–3 ADD_SUBSET_OUTBOUND_RULES Procedure Parameters

Parameter	Description
`server_name`	The name of the outbound server to which rules are being added. Specify an existing outbound server. Do not specify an owner.
`table_name`	The name of the table specified as `[`*`schema_name`*`.]`*`object_name`*. For example, `hr.employees`. If the schema is not specified, then the current user is the default. If the outbound server configuration uses a local capture process, then the table must exist at the local source database. If the outbound server configuration uses a downstream capture process, then the table must exist at both the source database and at the local downstream capture database. The specified table cannot have any LOB, `LONG`, or `LONG RAW` columns currently or in the future.

Table 7–3 (Cont.) ADD_SUBSET_OUTBOUND_RULES Procedure Parameters

Parameter	Description
`condition`	The subset condition. Specify this condition similar to the way you specify conditions in a `WHERE` clause in SQL. For example, to specify rows in the `hr.employees` table where the `salary` is greater than `4000` and the `job_id` is `SA_MAN`, enter the following as the condition: `' salary > 4000 and job_id = ''SA_MAN'' '` If `NULL`, then the procedure raises an error. **Note:** The quotation marks in the preceding example are all single quotation marks.
`column_list`	The list of columns either to include in the outbound server configuration or to exclude from the outbound server configuration. Whether the columns are included or excluded depends on the setting for the `keep` parameter. The columns can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.LNAME_ARRAY`, where each element is the name of a column. The first column should be in position 1. The last position must be `NULL`. If `NULL`, then all of the columns are included.
`keep`	If `TRUE`, then the columns specified in the `column_list` parameter are kept as part of the outbound server configuration. Therefore, changes to these columns that satisfy the condition in the `condition` parameter are streamed to the outbound server client application. If `FALSE`, then the columns specified in the `column_list` parameter are excluded from the outbound server configuration. Therefore, changes to these columns are not streamed to the outbound server client application. **See Also:** "Usage Notes" on page 7-11

Usage Notes

When `column_list` is non-`NULL` and `keep` is set to `TRUE`, this procedure creates a keep columns declarative rule-based transformation for the columns listed in `column_list`.

When `column_list` is non-`NULL` and `keep` is set to `FALSE`, this procedure creates a delete column declarative rule-based transformation for each column listed in `column_list`.

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

ALTER_INBOUND Procedure

This procedure modifies an XStream inbound server.

Syntax

```
DBMS_XSTREAM_ADM.ALTER_INBOUND(
   server_name IN VARCHAR2,
   apply_user  IN VARCHAR2  DEFAULT NULL,
   comment     IN VARCHAR2  DEFAULT NULL);
```

Parameters

Table 7–4 ALTER_INBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the inbound server being altered. Specify an existing inbound server. Do not specify an owner.
`apply_user`	The user who applies all data manipulation language (DML) and data definition language (DDL) changes that satisfy the inbound server rule sets, who runs user-defined apply handlers, and who runs custom rule-based transformations configured for inbound server rules. The client application must attach to the inbound server as the apply user. Specify a user to change the apply user. In this case, the user who invokes the `ALTER_INBOUND` procedure must be granted `DBA` role. Only the `SYS` user can set the `connect_user` to `SYS`. If `NULL`, then the apply user is not changed. See "CREATE_INBOUND Procedure" on page 7-16 for information about the required privileges for an apply user.
`comment`	An optional comment associated with the inbound server. If non-`NULL`, then the specified comment replaces the existing comment. If `NULL`, then the existing comment is not changed.

ALTER_OUTBOUND Procedure

This procedure modifies an XStream outbound server configuration.

This procedure always alters the specified outbound server. This procedure can also alter the outbound server's capture process when either of the following conditions are met:

- The capture process was created by the CREATE_OUTBOUND procedure in this package
- The queue used by the capture process was created by the CREATE_OUTBOUND procedure.

To check whether this procedure can alter the outbound server's capture process, query the CAPTURE_NAME column in the DBA_XSTREAM_OUTBOUND view. When the name of the capture process appears in the CAPTURE_NAME column of this view, the ALTER_OUTBOUND procedure can manage the capture process's rules or change the capture user for the capture process.

This procedure is overloaded. One table_names parameter is type VARCHAR2 and the other table_names parameters is type DBMS_UTILITY.UNCL_ARRAY. Also, one schema_names parameter is type VARCHAR2 and the other schema_names parameters is type DBMS_UTILITY.UNCL_ARRAY. These parameters enable you to enter the list of tables and schemas in different ways and are mutually exclusive.

Syntax

```
DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
   server_name    IN VARCHAR2,
   table_names    IN DBMS_UTILITY.UNCL_ARRAY,
   schema_names   IN DBMS_UTILITY.UNCL_ARRAY,
   add            IN BOOLEAN   DEFAULT TRUE,
   capture_user   IN VARCHAR2  DEFAULT NULL,
   connect_user   IN VARCHR2   DEFAULT NULL,
   comment        IN VARCHAR2  DEFAULT NULL,
   inclusion_rule IN BOOLEAN   DEFAULT TRUE);

DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
   server_name    IN VARCHAR2,
   table_names    IN VARCHAR2  DEFAULT NULL,
   schema_names   IN VARCHAR2  DEFAULT NULL,
   add            IN BOOLEAN   DEFAULT TRUE,
   capture_user   IN VARCHAR2  DEFAULT NULL,
   connect_user   IN VARCHAR2  DEFAULT NULL,
   comment        IN VARCHAR2  DEFAULT NULL,
   inclusion_rule IN BOOLEAN   DEFAULT TRUE);
```

Parameters

Table 7–5 ALTER_OUTBOUND Procedure Parameters

Parameter	Description
server_name	The name of the outbound server being altered. Specify an existing outbound server. Do not specify an owner.

Table 7–5 (Cont.) ALTER_OUTBOUND Procedure Parameters

Parameter	Description
`table_names`	The tables that are either added to or removed from the XStream Out configuration. Whether the tables are added or removed depends on the setting for the `add` parameter. The tables can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a table. The first table should be in position 1. The last position must be `NULL`. Each table should be specified as `[`*`schema_name`*`.]`*`table_name`*. For example, `hr.employees`. If the schema is not specified, then the current user is the default.
`schema_names`	The schemas that are either added to or removed from the XStream Out configuration. Whether the schemas are added or removed depends on the setting for the `add` parameter. The schemas can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a schema. The first schema should be in position 1. The last position must be `NULL`.
`add`	If `TRUE`, then the procedure adds to the XStream Out configuration the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter. If `FALSE`, then the procedure removes from the XStream Out configuration the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter.
`capture_user`	The user in whose security domain a capture process captures changes that satisfy its rule sets and runs custom rule-based transformations configured for capture process rules. Specify a user to change the capture user. In this case, the user who invokes the `ALTER_OUTBOUND` procedure must be granted `DBA` role. Only the `SYS` user can set the `capture_user` to `SYS`. If `NULL`, then the capture user is not changed. If you change the capture user, then this procedure grants the new capture user enqueue privilege on the queue used by the capture process and configures the user as a secure queue user of the queue. Ensure that the capture user is granted the other required privileges. See "CREATE_OUTBOUND Procedure" on page 7-18 for information about the privileges required by a capture user. The capture process is stopped and restarted automatically when you change the value of this parameter. **Note:** If the specified user is dropped using `DROP USER... CASCADE`, then the `capture_user` setting for the capture process is set to `NULL` automatically. You must specify a capture user before the capture process can run.

Table 7–5 (Cont.) ALTER_OUTBOUND Procedure Parameters

Parameter	Description
`connect_user`	The user who can attach to the specified outbound server to retrieve the change stream. The client application must attach to the outbound server as the specified connect user. Specify a user to change the connect user. In this case, the user who invokes the `ALTER_OUTBOUND` procedure must be granted `DBA` role. Only the `SYS` user can set the `connect_user` to `SYS`. If `NULL`, then the connect user is not changed. If you change the connect user, then this procedure grants the new connect user dequeue privilege on the queue used by the outbound server and configures the user as a secure queue user of the queue. Ensure that the connect user is granted the other required privileges. See "CREATE_OUTBOUND Procedure" on page 7-18 for information about the privileges required by a connect user.
`comment`	An optional comment associated with the outbound server. If non-`NULL`, then the specified comment replaces the existing comment. If `NULL`, then the existing comment is not changed.
`inclusion_rule`	If `TRUE` and the `ADD` parameter is set to `TRUE`, then the procedure adds rules for the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter to the positive rule sets in the XStream Out configuration. When rules for tables and schemas are in positive rule sets, the XStream Out configuration streams data manipulation language (DML) and data definition language (DDL) changes to the tables out to the client application. If `TRUE` and the `ADD` parameter is set to `FALSE`, then the procedure removes rules for the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter from the positive rule sets in the XStream Out configuration. If `FALSE` and the `ADD` parameter is set to `TRUE`, then the procedure adds rules for the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter to the negative rule sets in the XStream Out configuration. When rules for tables and schemas are in negative rule sets, the XStream Out configuration does not stream changes to the tables out to the client application. If `FALSE` and the `ADD` parameter is set to `FALSE`, then the procedure removes rules for the tables specified in the `table_names` parameter and the schemas specified in the `schema_names` parameter from the negative rule sets in the XStream Out configuration.

CREATE_INBOUND Procedure

This procedure creates an XStream inbound server and its queue.

Note:

- A client application can attach to only one inbound server at a time. See Part IV, "XStream OCI API Reference" and *Oracle Database XStream Java API Reference* for information about attaching to an inbound server.
- This procedure enables the inbound server that it creates.

Syntax

```
DBMS_XSTREAM_ADM.CREATE_INBOUND(
   server_name IN VARCHAR2,
   queue_name  IN VARCHAR2,
   apply_user  IN VARCHAR2  DEFAULT NULL,
   comment     IN VARCHAR2  DEFAULT NULL);
```

Parameters

Table 7–6 CREATE_INBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the inbound server being created. A `NULL` specification is not allowed. Do not specify an owner. The specified name must not match the name of an existing outbound server, inbound server, apply process, or messaging client. **Note:** The `server_name` setting cannot be altered after the inbound server is created.
`queue_name`	The name of the local queue used by the inbound server, specified as `[`*`schema_name`*`.]`*`queue_name`*. If the specified queue exists, then it is used. If the specified queue does not exist, then the procedure creates it. For example, to specify a queue named `streams_queue` in the `strmadmin` schema, enter `strmadmin.streams_queue` for this parameter. If the schema is not specified, then the current user is the default. **Note:** An inbound server's queue is only used to store error transactions.

Table 7–6 (Cont.) CREATE_INBOUND Procedure Parameters

Parameter	Description
`apply_user`	The apply user. If `NULL`, then the current user is the default. The client application must attach to the inbound server as the apply user. The apply user is the user in whose security domain an inbound server evaluates whether LCRs satisfy its rule sets, applies data manipulation language (DML) and data definition language (DDL) changes directly to database objects, runs custom rule-based transformations configured for inbound server rules, and runs apply handlers configured for the inbound server. This user must have the necessary privileges to perform these actions. This procedure grants the apply user dequeue privilege on the queue used by the inbound server and configures the user as a secure queue user of the queue. In addition to the privileges granted by this procedure, you also should grant the following privileges to the apply user: ■ The necessary privileges to perform DML and DDL changes on the apply objects ■ `EXECUTE` privilege on the rule sets used by the apply process ■ `EXECUTE` privilege on all rule-based transformation functions used in the rule set ■ `EXECUTE` privilege on all apply handler procedures These privileges can be granted directly to the apply user, or they can be granted through roles. In addition, the apply user must be granted `EXECUTE` privilege on all packages, including Oracle-supplied packages, that are invoked in subprograms run by the apply process. These privileges must be granted directly to the apply user. They cannot be granted through roles. **Note:** If the specified user is dropped using `DROP USER... CASCADE`, then the `apply_user` setting for the inbound server is set to `NULL` automatically. You must specify an apply user before the inbound server can run.
`comment`	An optional comment associated with the inbound server.

Usage Notes

By default, an inbound server does not use rules or rule sets. Therefore, an inbound server applies all of the LCRs sent to it by a client application. However, to filter the LCRs sent to an inbound server, you can add rules and rule sets to an inbound server using the `DBMS_STREAMS_ADM` and `DBMS_RULE_ADM` packages.

CREATE_OUTBOUND Procedure

This procedure creates an XStream outbound server, queue, and capture process to enable client applications to stream out Oracle database changes.

This procedure is overloaded. One `table_names` parameter is type `VARCHAR2` and the other `table_names` parameters is type `DBMS_UTILITY.UNCL_ARRAY`. Also, one `schema_names` parameter is type `VARCHAR2` and the other `schema_names` parameters is type `DBMS_UTILITY.UNCL_ARRAY`. These parameters enable you to enter the list of tables and schemas in different ways and are mutually exclusive.

Note:

- Only one client application at a time can attach to an outbound server. See "OCIXStreamOutAttach()" on page 12-56 and *Oracle Database XStream Java API Reference* for information about attaching to an outbound server.
- This procedure automatically generates a name for the capture process and the queue that it creates.
- This procedure enables both the capture process and outbound server that it creates.

Syntax

```
DBMS_XSTREAM_ADM.CREATE_OUTBOUND(
   server_name     IN VARCHAR2,
   source_database IN VARCHAR2  DEFAULT NULL,
   table_names     IN DBMS_UTILITY.UNCL_ARRAY,
   schema_names    IN DBMS_UTILITY.UNCL_ARRAY,
   capture_user    IN VARCHAR2  DEFAULT NULL,
   connect_user    IN VARCHAR2  DEFAULT NULL,
   comment         IN VARCHAR2  DEFAULT NULL);

DBMS_XSTREAM_ADM.CREATE_OUTBOUND(
   server_name     IN VARCHAR2,
   source_database IN VARCHAR2  DEFAULT NULL,
   table_names     IN VARCHAR2  DEFAULT NULL,
   schema_names    IN VARCHAR2  DEFAULT NULL,
   capture_user    IN VARCHAR2  DEFAULT NULL,
   connect_user    IN VARCHAR2  DEFAULT NULL,
   comment         IN VARCHAR2  DEFAULT NULL);
```

Parameters

Table 7–7 CREATE_OUTBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the outbound server being created. A `NULL` specification is not allowed. Do not specify an owner. The specified name must not match the name of an existing outbound server, inbound server, apply process, or messaging client. **Note:** The `server_name` setting cannot be altered after the outbound server is created.

Table 7–7 (Cont.) CREATE_OUTBOUND Procedure Parameters

Parameter	Description
`source_database`	The global name of the source database. The source database is where the changes to be captured originated. If you do not include the domain name, then the procedure appends it to the database name automatically. For example, if you specify `DBS1` and the domain is `.EXAMPLE.COM`, then the procedure specifies `DBS1.EXAMPLE.COM` automatically. If `NULL`, or if the specified name is the same as the global name of the current database, then local capture is assumed. If non-`NULL` and the specified name is different from the global name of the current database, then downstream capture is assumed. In this case, configure the transmission of redo data from the source database to the downstream database before running the `CREATE_OUTBOUND` procedure. See *Oracle Streams Replication Administrator's Guide* for instructions.
`table_names`	The tables for which data manipulation language (DML) and data definition language (DDL) changes are streamed out to the XStream Out client application. The tables can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a table. The first table should be in position 1. The last position must be `NULL`. Each table should be specified as `[`*`schema_name`*`.]`*`table_name`*. For example, `hr.employees`. If the schema is not specified, then the current user is the default. **See Also:** "Usage Notes" on page 7-21 for more information about this parameter.
`schema_names`	The schemas for which DML and DDL changes are streamed out to the XStream Out client application. The schemas can be specified in the following ways: ■ Comma-delimited list of type `VARCHAR2` ■ A PL/SQL index-by table of type `DBMS_UTILITY.UNCL_ARRAY`, where each element is the name of a schema. The first schema should be in position 1. The last position must be `NULL`. **See Also:** "Usage Notes" on page 7-21 for more information about this parameter.

Table 7–7 (Cont.) CREATE_OUTBOUND Procedure Parameters

Parameter	Description
`capture_user`	The user in whose security domain a capture process captures changes that satisfy its rule sets and runs custom rule-based transformations configured for capture process rules. If `NULL`, then the current user is the default. This procedure grants the capture user enqueue privilege on the queue used by the capture process and configures the user as a secure queue user of the queue. In addition, ensure that the capture user has the following privileges: ■ `EXECUTE` privilege on the rule sets used by the capture process ■ `EXECUTE` privilege on all rule-based transformation functions used in the positive rule set These privileges can be granted directly to the capture user, or they can be granted through roles. In addition, the capture user must be granted `EXECUTE` privilege on all packages, including Oracle-supplied packages, that are invoked in rule-based transformations run by the capture process. These privileges must be granted directly to the capture user. They cannot be granted through roles. Only a user who is granted `DBA` role can set a capture user. Only the `SYS` user can set the `capture_user` to `SYS`. A capture user does not require privileges on a database object to capture changes to the database object. The capture process can pass these changes to a custom rule-based transformation function. Therefore, ensure that you consider security implications when you configure a capture process.
`connect_user`	The user who can attach to the specified outbound server to retrieve the change stream. The client application must attach to the outbound server as the specified connect user. If `NULL`, then the current user is the default. The connect user is the user in whose security domain an outbound server dequeues LCRs that satisfy its rule sets and runs custom rule-based transformations configured for outbound server rules. This user must have the necessary privileges to perform these actions. This procedure grants the connect user dequeue privilege on the queue used by the outbound server and configures the user as a secure queue user of the queue. In addition to the privileges granted by this procedure, you also should grant the following privileges to the apply user: ■ `EXECUTE` privilege on the rule sets used by the apply process ■ `EXECUTE` privilege on all rule-based transformation functions used in the rule set These privileges can be granted directly to the connect user, or they can be granted through roles. In addition, the connect user must be granted `EXECUTE` privilege on all packages, including Oracle-supplied packages, that are invoked in subprograms run by the apply process. These privileges must be granted directly to the apply user. They cannot be granted through roles.
`comment`	An optional comment associated with the outbound server.

Usage Notes

The following list describes the behavior of the outbound server for various combinations of the `table_names` and `schema_names` parameters:

- If both the `table_names` and `schema_names` parameters are `NULL`, then the outbound server streams all DML and DDL changes to the client application.
- If both the `table_names` and `schema_names` parameters are specified, then the outbound server streams DML and DDL changes for the specified tables and schemas.
- If the `table_names` parameter is specified and the `schema_names` parameter is `NULL`, then the outbound server streams DML and DDL changes for the specified tables.
- If the `table_names` parameter is `NULL` and the `schema_names` parameter is specified, then the outbound server streams DML and DDL changes for the specified schema.

DROP_INBOUND Procedure

This procedure removes an inbound server configuration.

This procedure always removes the specified inbound server. This procedure also removes the queue for the inbound server if all of the following conditions are met:

- One call to the `CREATE_INBOUND` procedure created the queue.
- The inbound server is the only subscriber to the queue.

See Also: "CREATE_INBOUND Procedure" on page 7-16

Syntax

```
DBMS_XSTREAM_ADM.DROP_INBOUND(
   server_name IN VARCHAR2);
```

Parameters

Table 7–8 DROP_INBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the inbound server being removed. Specify an existing inbound server. Do not specify an owner.

DROP_OUTBOUND Procedure

This procedure removes an outbound server configuration.

This procedure always drops the specified outbound server. This procedure also drops the queue used by the outbound server if both of the following conditions are met:

- The queue was created by the `CREATE_OUTBOUND` procedure in this package.
- The outbound server is the only subscriber to the queue.

If either one of these conditions is not met, then the `DROP_OUTBOUND` procedure only drops the outbound server. It does not drop the queue.

This procedure also drops the capture process for the outbound server if both of the following conditions are met:

- The procedure can drop the outbound server's queue.
- The capture process was created by the `CREATE_OUTBOUND` procedure.

If the procedure can drop the queue but cannot manage the capture process, then it drops the queue without dropping the capture process.

See Also:

- "ADD_OUTBOUND Procedure" on page 7-7
- "CREATE_OUTBOUND Procedure" on page 7-18

Syntax

```
DBMS_XSTREAM_ADM.DROP_OUTBOUND(
   server_name IN VARCHAR2);
```

Parameters

Table 7–9 DROP_OUTBOUND Procedure Parameters

Parameter	Description
`server_name`	The name of the outbound server being removed. Specify an existing outbound server. Do not specify an owner.

REMOVE_SUBSET_OUTBOUND_RULES Procedure

This procedure removes subset rules from an outbound server configuration.

The names of the specified insert, update, and delete rules must match those generated by the ADD_SUBSET_OUTBOUND_RULES procedure. To view the rule names for subset rules, run the following query:

```
SELECT RULE_OWNER, SUBSETTING_OPERATION, RULE_NAME
   FROM DBA_XSTREAM_RULES
   WHERE SUBSETTING_OPERATION IS NOT NULL;
```

Note:

- This procedure removes the declarative rule-based transformation associated with each rule it removes.
- This procedure does not remove rules from the outbound server's capture process.

See Also: "ADD_SUBSET_OUTBOUND_RULES Procedure" on page 7-10

Syntax

```
DBMS_XSTREAM_ADM.REMOVE_SUBSET_OUTBOUND_RULES(
   server_name      IN VARCHAR2,
   insert_rule_name IN VARCHAR2,
   update_rule_name IN VARCHAR2,
   delete_rule_name IN VARCHAR2);
```

Parameters

Table 7–10 REMOVE_SUBSET_OUTBOUND_RULES Procedure Parameters

Parameter	Description
`server_name`	The name of the outbound server from which rules are being removed. Specify an existing outbound server. Do not specify an owner.
`insert_rule_name`	The name of the insert rule being removed, specified as `[`*schema_name*`.]`*rule_name*. For example, to specify a rule in the `hr` schema named `prop_rule1`, enter `hr.prop_rule1`. If the schema is not specified, then the current user is the default. If `NULL`, then the procedure raises an error.
`update_rule_name`	The name of the update rule being removed, specified as `[`*schema_name*`.]`*rule_name*. If `NULL`, then the procedure raises an error.
`delete_rule_name`	The name of the delete rule being removed, specified as `[`*schema_name*`.]`*rule_name*. If `NULL`, then the procedure raises an error.

8 Addendum To DBMS_APPLY_ADM

The DBMS_APPLY_ADM package, one of a set of Oracle Streams packages, provides subprograms to configure and manage apply processes, outbound servers, and inbound servers.

All of the subprograms in this package can manage Oracle Streams apply processes. In an XStream configuration, an apply process can also function as an XStream outbound server or inbound server. A subset of the subprograms in this package can manage outbound servers and inbound servers.

The following sections describe how these subprograms work with outbound servers and inbound servers:

- DBMS_APPLY_ADM Subprograms and Outbound Servers
- DBMS_APPLY_ADM Subprograms and Inbound Servers

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- Chapter 7, "DBMS_XSTREAM_ADM"

DBMS_APPLY_ADM Subprograms and Outbound Servers

Table 8–1 shows whether a subprogram in this package can manage an outbound server. This table also provides notes about how the subprogram works with an outbound server. Also, when a subprogram can manage an outbound server, the parameters in the subprogram work the same way for an outbound server as they would for an apply process, unless the "Notes" column in the table describes different behavior for a parameter.

Table 8–1 DBMS_APPLY_ADM Subprograms and Outbound Servers

Subprogram	Purpose	Can Manage an Outbound Server?	Notes
`ADD_STMT_HANDLER` Procedure	Adds a statement DML handler for a specified operation on a specified database object to a single Oracle Streams apply process or to all apply processes in the database	No	Outbound servers ignore all apply handlers.
`ALTER_APPLY` Procedure	Alters an Oracle Streams outbound server	Yes	Outbound servers only process LCRs. Therefore, outbound servers ignore message handlers specified in the `message_handler` parameter. The `apply_user` parameter can change the connect user for an outbound server.
`COMPARE_OLD_VALUES` Procedure	Specifies whether to compare the old value of one or more columns in a row logical change record (row LCR) with the current value of the corresponding columns at the destination site during apply	No	This procedure cannot manage an outbound server.
`CREATE_APPLY` Procedure	Creates an apply process	No	This procedure cannot create an outbound server.
`CREATE_OBJECT_DEPENDENCY` Procedure	Creates an object dependency	No	This procedure cannot manage an outbound server.
`DELETE_ALL_ERRORS` Procedure	Deletes all the error transactions for the specified apply process	No	This procedure cannot manage an outbound server.
`DELETE_ERROR` Procedure	Deletes the specified error transaction	No	This procedure cannot manage an outbound server.
`DROP_APPLY` Procedure	Drops an outbound server	Yes	When the `DROP_APPLY` procedure is executed on an outbound server, it runs the `DROP_OUTBOUND` procedure in the `DBMS_XSTREAM_ADM` package. Therefore, it might also drop the outbound server's capture process and queue.
`DROP_OBJECT_DEPENDENCY` Procedure	Drops an object dependency	No	This procedure cannot manage an outbound server.
`EXECUTE_ALL_ERRORS` Procedure	Reexecutes the error transactions for the specified apply process.	No	Outbound servers do not enqueue error transactions into an error queue.
`EXECUTE_ERROR` Procedure	Reexecutes the specified error transaction	No	Outbound servers do not enqueue error transactions into an error queue.
`GET_ERROR_MESSAGE` Function	Returns the message payload from the error queue for the specified message number and transaction identifier	No	This procedure cannot manage an outbound server.

Table 8–1 (Cont.) DBMS_APPLY_ADM Subprograms and Outbound Servers

Subprogram	Purpose	Can Manage an Outbound Server?	Notes
`REMOVE_STMT_HANDLER` Procedure	Removes a statement DML handler for a specified operation on a specified database object from a single apply process or from all apply processes in the database	No	This procedure cannot manage an outbound server.
`SET_CHANGE_HANDLER` Procedure	Sets or unsets a statement DML handler that tracks changes for a specified operation on a specified database object for a single apply process	No	Outbound servers ignore all apply handlers.
`SET_DML_HANDLER` Procedure	Sets a user procedure as a procedure DML handler for a specified operation on a specified database object for a single apply process or for all apply processes in the database	No	Outbound servers ignore all apply handlers.
`SET_ENQUEUE_DESTINATION` Procedure	Sets the queue where the apply process automatically enqueues a message that satisfies the specified rule	No	This procedure cannot manage an outbound server.
`SET_EXECUTE` Procedure	Specifies whether a message that satisfies the specified rule is executed by an apply process	No	This procedure cannot manage an outbound server.
`SET_GLOBAL_INSTANTIATION_SCN` Procedure	Records the specified instantiation SCN for the specified source database and, optionally, for the schemas at the source database and the tables owned by these schemas	Yes	The `apply_database_link` parameter must be set to `NULL` or to the local database for this procedure to set an instantiation SCN for an outbound server. **See Also:** "Apply Process Features That Are Applicable to Outbound Servers" on page 1-4 for more information about outbound servers and instantiation SCNs
`SET_KEY_COLUMNS` Procedures	Records the set of columns to be used as the substitute primary key for local apply purposes and removes existing substitute primary key columns for the specified object if they exist	No	This procedure cannot manage an outbound server.

Table 8–1 (Cont.) DBMS_APPLY_ADM Subprograms and Outbound Servers

Subprogram	Purpose	Can Manage an Outbound Server?	Notes
SET_PARAMETER Procedure	Sets an apply parameter to the specified value	Yes	Outbound servers ignore the settings for the following apply parameters: ■ allow_duplicate_rows ■ commit_serialization ■ disable_on_error ■ parallelism ■ preserve_encryption ■ rtrim_on_implicit_conversion The commit_serialization parameter is always set to FULL for an outbound server, and the parallelism parameter is always set to 1 for an outbound server. You can use the other apply parameters with outbound servers.
SET_SCHEMA_INSTANTIATION_SCN Procedure	Records the specified instantiation SCN for the specified schema in the specified source database and, optionally, for the tables owned by the schema at the source database	Yes	The apply_database_link parameter must be set to NULL or to the local database for this procedure to set an instantiation SCN for an outbound server. **See Also:** "Apply Process Features That Are Applicable to Outbound Servers" on page 1-4 for more information about outbound servers and instantiation SCNs
SET_TABLE_INSTANTIATION_SCN Procedure	Records the specified instantiation SCN for the specified table in the specified source database	Yes	The apply_database_link parameter must be set to NULL or to the local database for this procedure to set an instantiation SCN for an outbound server. **See Also:** "Apply Process Features That Are Applicable to Outbound Servers" on page 1-4 for more information about outbound servers and instantiation SCNs
SET_UPDATE_CONFLICT_HANDLER Procedure	Adds, updates, or drops an update conflict handler for the specified object	No	This procedure cannot manage an outbound server.
SET_VALUE_DEPENDENCY Procedure	Sets or removes a value dependency	No	This procedure cannot manage an outbound server.
START_APPLY Procedure	Directs the outbound server to start applying messages	Yes	This procedure functions the same way for apply processes and outbound servers.
STOP_APPLY Procedure	Stops the outbound server from applying any messages and rolls back any unfinished transactions being applied	Yes	This procedure functions the same way for apply processes and outbound servers.

DBMS_APPLY_ADM Subprograms and Inbound Servers

Table 8–2 shows whether a subprogram in this package can manage an inbound server. This table also provides notes about how the subprogram works with an inbound server. Also, when a subprogram can manage an inbound server, the parameters in the subprogram work the same way for an inbound server as they would for an apply process, unless the "Notes" column in the table describes different behavior for a parameter.

Table 8–2 DBMS_APPLY_ADM Subprograms and Inbound Servers

Subprogram	Purpose	Can Manage an Inbound Server?	Notes
ADD_STMT_HANDLER Procedure	Adds a statement DML handler for a specified operation on a specified database object to a single inbound server or to all inbound servers in the database	Yes	Inbound servers can use statement DML handlers.
ALTER_APPLY Procedure	Alters an inbound server	Yes	The apply_user parameter can change the apply user for an inbound server.
COMPARE_OLD_VALUES Procedure	Specifies whether to compare the old value of one or more columns in a row logical change record (row LCR) with the current value of the corresponding columns at the destination site during apply	Yes	This procedure functions the same way for apply processes and inbound servers.
CREATE_APPLY Procedure	Creates an inbound server	Yes	The CREATE_APPLY procedure always creates an apply process. The apply process remains an apply process if it receives messages from a source other than an XStream client application, such as a capture process. The apply process can become an inbound server if an XStream client application attaches to it before it receives messages from any other source. After the initial contact, an apply process cannot be changed into an inbound server, and an inbound server cannot be changed into an apply process. When creating an inbound server using the CREATE_APPLY procedure, set the apply_captured parameter to TRUE. Inbound server can use apply handlers. However, inbound servers only process LCRs. Therefore, inbound servers ignore message handlers specified in the message_handler parameter.
CREATE_OBJECT_DEPENDENCY Procedure	Creates an object dependency	Yes	This procedure functions the same way for apply processes and inbound servers.

Table 8–2 (Cont.) DBMS_APPLY_ADM Subprograms and Inbound Servers

Subprogram	Purpose	Can Manage an Inbound Server?	Notes
`DELETE_ALL_ERRORS` Procedure	Deletes all the error transactions for the specified inbound server	Yes	This procedure functions the same way for apply processes and inbound servers.
`DELETE_ERROR` Procedure	Deletes the specified error transaction	Yes	This procedure functions the same way for apply processes and inbound servers.
`DROP_APPLY` Procedure	Drops an inbound server	Yes	When the `DROP_APPLY` procedure is executed on an inbound server, it runs the `DROP_INBOUND` procedure in the `DBMS_XSTREAM_ADM` package. Therefore, it might also drop the inbound server's queue.
`DROP_OBJECT_DEPENDENCY` Procedure	Drops an object dependency	Yes	This procedure functions the same way for apply processes and inbound servers.
`EXECUTE_ALL_ERRORS` Procedure	Reexecutes the error transactions for the specified inbound server	Yes	Inbound servers enqueue error transactions into an error queue the same way that apply processes enqueue error transactions into an error queue.
`EXECUTE_ERROR` Procedure	Reexecutes the specified error transaction	Yes	Inbound servers enqueue error transactions into an error queue the same way that apply processes enqueue error transactions into an error queue.
`GET_ERROR_MESSAGE` Function	Returns the message payload from the error queue for the specified message number and transaction identifier	Yes	This procedure functions the same way for apply processes and inbound servers.
`REMOVE_STMT_HANDLER` Procedure	Removes a statement DML handler for a specified operation on a specified database object from a single inbound server or from all inbound servers in the database	Yes	This procedure functions the same way for apply processes and inbound servers.
`SET_CHANGE_HANDLER` Procedure	Sets or unsets a statement DML handler that tracks changes for a specified operation on a specified database object for a single inbound server	Yes	Inbound servers can use change handlers.
`SET_DML_HANDLER` Procedure	Sets a user procedure as a procedure DML handler for a specified operation on a specified database object for a single inbound server or for all inbound servers in the database	Yes	Inbound servers can use procedure DML handlers.
`SET_ENQUEUE_DESTINATION` Procedure	Sets the queue where the inbound server automatically enqueues a message that satisfies the specified rule	Yes	This procedure functions the same way for apply processes and inbound servers.

Table 8–2 (Cont.) DBMS_APPLY_ADM Subprograms and Inbound Servers

Subprogram	Purpose	Can Manage an Inbound Server?	Notes
`SET_EXECUTE` Procedure	Specifies whether a message that satisfies the specified rule is executed by an inbound server	Yes	This procedure functions the same way for apply processes and inbound servers.
`SET_GLOBAL_ INSTANTIATION_SCN` Procedure	Records the specified instantiation SCN for the specified source database and, optionally, for the schemas at the source database and the tables owned by these schemas	No	Inbound servers ignore instantiation SCNs.
`SET_KEY_COLUMNS` Procedures	Records the set of columns to be used as the substitute primary key for local apply purposes and removes existing substitute primary key columns for the specified object if they exist	Yes	This procedure functions the same way for apply processes and inbound servers.
`SET_PARAMETER` Procedure	Sets an apply parameter to the specified value	Yes	Inbound servers ignore the setting for the `maximum_scn` apply process parameter. You can use all of the other apply process parameters with inbound servers. The default parallelism of an inbound server is 4.
`SET_SCHEMA_ INSTANTIATION_SCN` Procedure	Records the specified instantiation SCN for the specified schema in the specified source database and, optionally, for the tables owned by the schema at the source database	No	Inbound servers ignore instantiation SCNs.
`SET_TABLE_ INSTANTIATION_SCN` Procedure	Records the specified instantiation SCN for the specified table in the specified source database	No	Inbound servers ignore instantiation SCNs.
`SET_UPDATE_CONFLICT_ HANDLER` Procedure	Adds, updates, or drops an update conflict handler for the specified object	Yes	This procedure functions the same way for apply processes and inbound servers.
`SET_VALUE_DEPENDENCY` Procedure	Sets or removes a value dependency	Yes	This procedure functions the same way for apply processes and inbound servers.
`START_APPLY` Procedure	Directs the inbound server to start applying messages	Yes	This procedure functions the same way for apply processes and inbound servers.
`STOP_APPLY` Procedure	Stops the inbound server from applying any messages and rolls back any unfinished transactions being applied	Yes	This procedure functions the same way for apply processes and inbound servers.

9

Addendum To DBMS_STREAMS_ADM

The DBMS_STREAMS_ADM package, one of a set of Oracle Streams packages, provides subprograms to configure and manage Oracle Streams apply processes, outbound servers, and inbound servers.

This chapter contains the following topic:

- Inbound Server Creation and Apply User
- XStream Inbound Server Rules for LCRs
- XStream Outbound Server Rules for LCRs

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- Chapter 7, "DBMS_XSTREAM_ADM"

Inbound Server Creation and Apply User

The following procedures in the DBMS_STREAMS_ADM package can create an XStream inbound server:

- ADD_GLOBAL_RULES procedure
- ADD_SCHEMA_RULES procedure
- ADD_SUBSET_RULES procedure
- ADD_TABLE_RULES procedure

If the streams_name parameter in one of these procedures is set to NULL, if the streams_type parameter is set to apply, and if one relevant apply process, inbound server, or outbound server exists, then the procedure uses the relevant apply process, inbound server, or outbound server. The relevant apply process, inbound server, or outbound server is identified in one of the following ways:

- If one existing apply process or outbound server has the source database specified in the source_database parameter and uses the queue specified in the queue_name parameter, then the procedure uses this apply process or outbound server.
- If the source_database parameter is set to NULL and one existing apply process, inbound server, or outbound server is using the queue specified in the queue_name parameter, then the procedure uses this apply process, inbound server, or outbound server.
- If the streams_name parameter is set to NULL and no relevant apply process, inbound server, or outbound server exists, then the procedure creates an apply process automatically with a system-generated name.

The apply process remains an apply process if it receives captured LCRs from a capture process. The apply process can become an inbound server if an XStream client application attaches to it before it receives captured LCRs from a capture process. After the initial contact, an apply process cannot be changed into an inbound server, and an inbound server cannot be changed into an apply process.

- If the `streams_name` parameter is set to `NULL` and multiple relevant apply processes, inbound servers, or outbound servers exist, then the procedure raises an error.

If one of these procedures creates an inbound server, then it configures the current user as the apply user. The apply user is the user in whose security domain an XStream client application attaches to an Oracle database.

An apply user applies changes directly to database objects, runs custom rule-based transformations configured for inbound server rules, and runs apply handlers configured for the inbound server. This user must have the necessary privileges to apply changes. The procedure grants the apply user `DEQUEUE` privilege on the queue used by the inbound server and configures the user as a secure queue user of the queue.

Each inbound server must have a unique name. The name cannot be used by an apply process, outbound server, or messaging client in the same database, and the name cannot be used by another inbound server in the same database.

Note: These procedures cannot create an outbound server.

XStream Inbound Server Rules for LCRs

The following procedures add rules to a rule set of an XStream inbound server when you specify `apply` for the `streams_type` parameter and an inbound server for the `streams_name` parameter:

- The `ADD_GLOBAL_RULES` procedure adds rules whose rule condition evaluates to `TRUE` for all LCRs in the inbound server's queue.
- The `ADD_SCHEMA_RULES` procedure adds rules whose rule condition evaluates to `TRUE` for LCRs in the inbound server's queue containing changes made to a specified schema.
- The `ADD_SUBSET_RULES` procedure adds rules whose rule condition evaluates to `TRUE` for row LCRs in the inbound server's queue containing the results of DML changes made to a subset of rows in a specified table.
- The `ADD_TABLE_RULES` procedure adds rules whose rule condition evaluates to `TRUE` for LCRs in the inbound server's queue containing changes made to a specified table.

If one of these procedures adds rules to the positive rule set for the inbound server, then the rules specify that the inbound server applies LCRs sent to it by the XStream client application. If one of these procedures adds rules to the negative rule set for the inbound server, then the rules specify that the discards LCRs sent to it by the XStream client application. For inbound server rules, you should execute these procedures at the database to which the XStream client application attaches. If an inbound server has no rule sets, then it applies all of the messages sent to it by the XStream client application.

Changes applied by an inbound server created by one of these procedures generate tags in the redo log at the destination database with a value of `'00'` (double zero).

You can use the ALTER_APPLY procedure in the DBMS_APPLY_ADM package to alter the tag value after the inbound server is created, if necessary.

The rules in the XStream inbound server rule sets determine which messages are either applied or discarded after the messages are received from the XStream client application. An inbound server can only process LCRs sent from an XStream client application.

When one of these procedures creates rules for an inbound server, the procedure ignores the source_database parameter.

Note: If the name specified in the streams_name parameter does not exist, then these procedures always create an apply process. The apply process remains an apply process if it receives captured LCRs from a capture process. The apply process can become an inbound server if an XStream client application attaches to it before it receives captured LCRs from a capture process. After the initial contact, an apply process cannot be changed into an inbound server, and an inbound server cannot be changed into an apply process.

See Also:

- Chapter 7, "DBMS_XSTREAM_ADM"
- *Oracle Database PL/SQL Packages and Types Reference*

XStream Outbound Server Rules for LCRs

The following procedures add rules to a rule set of an XStream outbound server when you specify apply for the streams_type parameter and an outbound server for the streams_name parameter:

- The ADD_GLOBAL_RULES procedure adds rules whose rule condition evaluates to TRUE for all LCRs in the outbound server's queue.
- The ADD_SCHEMA_RULES procedure adds rules whose rule condition evaluates to TRUE for LCRs in the outbound server's queue containing changes made to a specified schema.
- The ADD_SUBSET_RULES procedure adds rules whose rule condition evaluates to TRUE for row LCRs in the outbound server's queue containing the results of DML changes made to a subset of rows in a specified table.
- The ADD_TABLE_RULES procedure adds rules whose rule condition evaluates to TRUE for LCRs in the outbound server's queue containing changes made to a specified table.

If one of these procedures adds rules to the positive rule set for the outbound server, then the rules specify that the outbound server sends LCRs in its queue to the XStream client application. If one of these procedures adds rules to the negative rule set for the outbound server, then the rules specify that the outbound server discards LCRs in its queue. For outbound server rules, you should execute these procedures at the database to which the XStream client application attaches.

An outbound server can process captured LCRs from only one source database. The source database is the database where the changes originated. If one of these procedures creates an outbound server, then specify the source database for the outbound server using the source_database parameter. If the source_database parameter is NULL, and one of these procedures creates an outbound server, then the source database name of the first LCR received by the outbound server is used for the source database.

The rules in the outbound server's rule sets determine which messages are dequeued by the outbound server. When you create rules with one of these procedures, and you specify a value for the `source_database` parameter, then the rules include conditions for the specified source database. If the outbound server dequeues an LCR with a source database that is different than the source database for the outbound server, then an error is raised. In addition, when adding rules to an existing outbound server, the database specified in the `source_database` parameter cannot be different than the source database for the outbound server. You can determine the source database for an outbound server by querying the `DBA_XSTREAM_OUTBOUND` data dictionary view.

Note: These procedures cannot create an XStream outbound server. You can use one of the procedures in the `DBMS_STREAMS_ADM` package to add rules to a rule set used by the outbound server after you create it.

10 Addendum To Logical Change Record Types

In Oracle Streams, logical change record (LCR) types are message payloads that contain information about changes to a database. These changes can include changes to the data, which are data manipulation language (DML) changes, and changes to database objects, which are data definition language (DDL) changes.

When you use Oracle Streams, the capture process captures changes in the form of LCRs and enqueues them into a queue. These LCRs can be propagated from a queue in one database to a queue in another database. Finally, the apply process can apply LCRs at a destination database. You also have the option of creating, enqueuing, and dequeuing LCRs manually.

This chapter describes the constructors for LCR types. This chapter also describes functions that are intended to be used with XStream outbound servers.

This chapter contains the following topics:

- Logical Change Record Constructors
- Functions Related to XStream

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `LCR$_DDL_RECORD` type and the `LCR$_ROW_RECORD` type
- "Position Order in an LCR Stream" on page 1-9
- Chapter 7, "DBMS_XSTREAM_ADM"

Logical Change Record Constructors

This section describes the following logical change record (LCR) constructors:

- DDL Logical Change Record Constructor
- Row Logical Change Record Constructor

DDL Logical Change Record Constructor

The DDL LCR constructor function creates a `SYS.LCR$_DDL_RECORD` object with the specified information.

```
STATIC FUNCTION CONSTRUCT(
   source_database_name  IN  VARCHAR2,
   command_type          IN  VARCHAR2,
   object_owner          IN  VARCHAR2,
   object_name           IN  VARCHAR2,
   object_type           IN  VARCHAR2,
   ddl_text              IN  CLOB,
   logon_user            IN  VARCHAR2,
   current_schema        IN  VARCHAR2,
   base_table_owner      IN  VARCHAR2,
   base_table_name       IN  VARCHAR2,
   tag                   IN  RAW       DEFAULT NULL,
   transaction_id        IN  VARCHAR2  DEFAULT NULL,
   scn                   IN  NUMBER    DEFAULT NULL,
   position              IN  RAW       DEFAULT NULL,
   edition_name          IN  VARCHAR2  DEFAULT NULL)
RETURN SYS.LCR$_DDL_RECORD;
```

Table 10–1 describes the parameters for the DDL LCR constructor function.

Table 10–1 Constructor Function Parameters for LCR$_DDL_RECORD

Parameter	Description
`source_database_name`	The database where the DDL statement occurred. If you do not include the domain name, then the function appends the local domain to the database name automatically. For example, if you specify `DBS1` and the local domain is `.NET`, then the function specifies `DBS1.NET` automatically. Set this parameter to a non-`NULL` value.

`command_type`	The type of command executed in the DDL statement. Set this parameter to a non-`NULL` value.
	
	See Also: The "SQL Command Codes" table in the *Oracle Call Interface Programmer's Guide* for a complete list of command types
	
	The following command types *are not supported* in DDL LCRs:
	
	`ALTER MATERIALIZED VIEW`
	`ALTER MATERIALIZED VIEW LOG`
	`ALTER SUMMARY`
	`CREATE SCHEMA`
	`CREATE MATERIALIZED VIEW`
	`CREATE MATERIALIZED VIEW LOG`
	`CREATE SUMMARY`
	`DROP MATERIALIZED VIEW`
	`DROP MATERIALIZED VIEW LOG`
	`DROP SUMMARY`
	`RENAME`
	
	The snapshot equivalents of the materialized view command types are also not supported.

`object_owner`	The user who owns the object on which the DDL statement was executed
`object_name`	The database object on which the DDL statement was executed

Table 10–1 (Cont.) Constructor Function Parameters for LCR$_DDL_RECORD

Parameter	Description

`object_type`	The type of object on which the DDL statement was executed.
	The following are valid object types:
	`CLUSTER`
	`FUNCTION`
	`INDEX`
	`LINK`
	`OUTLINE`
	`PACKAGE`
	`PACKAGE BODY`
	`PROCEDURE`
	`SEQUENCE`
	`SYNONYM`
	`TABLE`
	`TRIGGER`
	`TYPE`
	`USER`
	`VIEW`
	`LINK` represents a database link.
	`NULL` is also a valid object type. Specify `NULL` for all object types not listed. The `GET_OBJECT_TYPE` member procedure returns `NULL` for object types not listed.

`ddl_text`	The text of the DDL statement. Set this parameter to a non-`NULL` value.
`logon_user`	The user whose session executed the DDL statement
`current_schema`	The schema that is used if no schema is specified explicitly for the modified database objects in `ddl_text`. If a schema is specified in `ddl_text` that differs from the one specified for `current_schema`, then the function uses the schema specified in `ddl_text`. Set this parameter to a non-`NULL` value.
`base_table_owner`	If the DDL statement is a table related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_owner` specifies the owner of the table involved. Otherwise, `base_table_owner` is `NULL`.
`base_table_name`	If the DDL statement is a table related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_name` specifies the name of the table involved. Otherwise, `base_table_name` is `NULL`.
`tag`	A binary tag that enables tracking of the LCR. For example, this tag can be used to determine the original source database of the DDL statement if apply forwarding is used. **See Also:** *Oracle Streams Replication Administrator's Guide* for more information about tags
`transaction_id`	The identifier of the transaction
`scn`	The SCN at the time when the change record for a captured LCR was written to the redo log. The SCN value is meaningless for a user-created LCR.

Table 10–1 (Cont.) Constructor Function Parameters for LCR$_DDL_RECORD

Parameter	Description
`position`	The position of the LCR.
`edition_name`	The name of the edition in which the DDL statement was executed.

Row Logical Change Record Constructor

The row LCR constructor function creates a `SYS.LCR$_ROW_RECORD` object with the specified information.

```
STATIC FUNCTION CONSTRUCT(
   source_database_name  IN  VARCHAR2,
   command_type          IN  VARCHAR2,
   object_owner          IN  VARCHAR2,
   object_name           IN  VARCHAR2,
   tag                   IN  RAW                DEFAULT NULL,
   transaction_id        IN  VARCHAR2           DEFAULT NULL,
   scn                   IN  NUMBER             DEFAULT NULL,
   old_values            IN  SYS.LCR$_ROW_LIST  DEFAULT NULL,
   new_values            IN  SYS.LCR$_ROW_LIST  DEFAULT NULL,
   position              IN  RAW                DEFAULT NULL)
RETURN SYS.LCR$_ROW_RECORD;
```

Table 10–2 describes the parameters for the DDL LCR constructor function.

Table 10–2 Constructor Function Parameters for LCR$_ROW_RECORD

Parameter	Description
`source_database_name`	The database where the row change occurred. If you do not include the domain name, then the function appends the local domain to the database name automatically. For example, if you specify `DBS1` and the local domain is `.NET`, then the function specifies `DBS1.NET` automatically. Set this parameter to a non-`NULL` value.

`command_type`	The type of command executed in the DML statement. Set this parameter to a non-`NULL` value.
	
	Valid values are the following:
	
	`INSERT`
	`UPDATE`
	`DELETE`
	`LOB ERASE`
	`LOB WRITE`
	`LOB TRIM`
	
	If `INSERT`, then ensure that the LCR has a `new_values` collection that is not empty and an empty or `NULL` `old_values` collection.
	
	If `UPDATE`, then ensure that the LCR has a `new_values` collection that is not empty and an `old_values` collection that is not empty.
	
	If `DELETE`, then ensure that the LCR has a `NULL` or empty `new_values` collection and an `old_values` collection that is not empty.
	
	If `LOB ERASE`, `LOB WRITE`, or `LOB TRIM`, then ensure that the LCR has a `new_values` collection that is not empty and an empty or `NULL` `old_values` collection.

Table 10–2 (Cont.) Constructor Function Parameters for LCR$_ROW_RECORD

Parameter	Description
`object_owner`	The user who owns the table on which the row change occurred. Set this parameter to a non-`NULL` value.
`object_name`	The table on which the DML statement was executed. Set this parameter to a non-`NULL` value.
`tag`	A binary tag that enables tracking of the LCR. For example, this tag can be used to determine the original source database of the DML change when apply forwarding is used. **See Also:** *Oracle Streams Replication Administrator's Guide* for more information about tags
`transaction_id`	The identifier of the transaction
`scn`	The SCN at the time when the change record was written to the redo log. The SCN value is meaningless for a user-created LCR.
`old_values`	The column values for the row before the DML change. If the DML statement is an `UPDATE` or a `DELETE` statement, then this parameter contains the values of columns in the row before the DML statement. If the DML statement is an `INSERT` statement, then there are no old values.
`new_values`	The column values for the row after the DML change. If the DML statement is an `UPDATE` or an `INSERT` statement, then this parameter contains the values of columns in the row after the DML statement. If the DML statement is a `DELETE` statement, then there are no new values. If the LCR reflects a LOB operation, then this parameter contains the supplementally logged columns and any relevant LOB information.
`position`	The position of the LCR.

Functions Related to XStream

This section describes functions that are intended to be used with XStream outbound servers. These functions are common to both the `LCR$_DDL_RECORD` type and the `LCR$_ROW_RECORD` type.

This section contains the following topics:

- GET_COMMIT_SCN_FROM_POSITION Static Function
- GET_SCN_FROM_POSITION Static Function
- GET_POSITION Member Function

GET_COMMIT_SCN_FROM_POSITION Static Function

Gets the commit system change number (SCN) of a transaction from the input position, which is generated by an XStream outbound server.

Syntax

```
STATIC FUNCTION GET_COMMIT_SCN_FROM_POSITION(
   position  IN  RAW)
RETURN NUMBER;
```

Parameters

Table 10–3 GET_COMMIT_SCN_FROM_POSITION Function Parameter

Parameter	Description
`position`	The position. You can obtain the position by using the `GET_POSITION` member function or by querying the `DBA_XSTREAM_OUTBOUND_PROGRESS` data dictionary view.

GET_SCN_FROM_POSITION Static Function

Gets the system change number (SCN) from the input position, which is generated by an XStream outbound server.

Syntax

```
STATIC FUNCTION GET_SCN_FROM_POSITION(
   position  IN  RAW)
RETURN NUMBER;
```

Parameters

Table 10–4 GET_SCN_FROM_POSITION Function Parameter

Parameter	Description
`position`	The position. You can obtain the position by using the `GET_POSITION` member function or by querying the `DBA_XSTREAM_OUTBOUND_PROGRESS` data dictionary view.

GET_POSITION Member Function

Gets the position of the current LCR. The position uniquely identifies each LCR. The position strictly increases within each transaction and across transactions.

Syntax

```
MEMBER FUNCTION GET_POSITION()
RETURN RAW;
```

Part IV

XStream OCI API Reference

This part contains the XStream OCI API reference. This part contains the following chapters:

11 Introduction to the OCI Interface for XStream

The Oracle Call Interface (OCI) includes an interface for XStream. This chapter provides an introduction to the OCI interface for XStream.

This chapter contains these topics:

- Using the XStream Interface
- Handler and Descriptor Attributes

> **See Also:**
>
> - Chapter 12, "OCI XStream Functions"
> - Chapter 1, "XStream Concepts"
> - Chapter 3, "Configuring XStream"
> - Chapter 4, "Managing XStream"
> - Chapter 5, "Monitoring XStream"
> - Chapter 6, "Troubleshooting XStream"

Using the XStream Interface

Since Oracle Database 11g Release 2, Oracle Streams provides enhanced APIs, known as eXtended Streams (XStream) Out and XStream In, to enable high performance, near real-time information-sharing infrastructure between Oracle databases and non-Oracle databases, non-RDBMS Oracle products, file systems, third party software applications, and so on.

XStream is built on top of Streams infrastructure.

> **See Also:** Chapter 12, "OCI XStream Functions"

XStream Out

XStream Out allows a remote client to attach to an outbound server (a Streams apply process) and extract row changes in the form of Logical Change Records (LCRs). For the basics of LCRs:

> **See Also:** *Oracle Streams Concepts and Administration*

To use XStream Out, a capture and an apply process must be created similar to other Streams setup. All data types supported by Oracle Streams including LOB, `LONG`, and `XMLType` are supported by XStream. Such an apply process is called an outbound server. The capture and the outbound server may or may not be on the same database instance. After the capture and the outbound server have started, row changes will be captured and sent to the outbound server. An external client can then connect to this outbound server using OCI. After the connection is established, the client can loop waiting for LCRs from the outbound server. The client can register a client-side callback to be invoked each time an LCR is received. At anytime, the client can detach from the outbound server as needed. Upon restart, the outbound server knows where in the redo stream to start streaming LCRs to the client.

See Also: Chapter 1, "XStream Concepts" for more details of XStream concepts

LCR Streams

- An LCR stream must be repeatable.
- An LCR stream must contain a list of assembled and committed transactions.
- LCRs from one transaction are contiguous. There is no interleaving of transactions in the LCR stream.
- Each transaction within an LCR stream must have an ordered list of LCRs and a transaction ID.
- The last LCR in each transaction must be a commit LCR.
- Each LCR must have a unique position.
- The position of all LCRs within a single transaction and across transactions must be strictly increasing.

The Processed Low Position and Restart Considerations

If the outbound server or the client aborts abnormally, the connection between the two is automatically broken. The client must maintain the processed low position to properly recover after a restart.

The processed low position is a position below which all LCRs have been processed by the client. This position should be maintained by the client while applying each transaction. Periodically this position is sent to the server while the client executes XStream Out APIs. This position indicates to the server that the client has processed all LCRs below or equal to this position; thus, the server can purge redo logs that are no longer needed.

Upon restart, the client must re-attach to the outbound server. During the attach call, the client can notify the outbound server of the last position received by the client. The outbound server then sends LCRs with position greater than this last position. If the client does not specify the last position (that is, a `NULL` is specified), the outbound server will retrieve the processed low position from its system tables and derive the starting position to mine the redo logs. It will send to the client the LCRs with position greater than this processed low position.

XStream In

To replicate non-Oracle data into Oracle databases use XStream In which allows a remote client to attach to an inbound server (a Streams apply process) and send row and DDL changes in the form of LCRs.

An external client application connects to the inbound server using OCI. After the connection is established, the client application acts as the capture agent for the inbound server by streaming LCRs to it. A client application can attach to only one inbound server per database connection. Each inbound server only allows one client attaching to it.

XStream In uses the following features of Oracle Streams:

- High performance processing of DML changes using an apply process and, optionally, apply process parallelism.
- Apply process features such as SQL generation, conflict detection and resolution, error handling, and customized processing with apply handlers.
- Streaming network transmission of information with minimal network round trips.

XStream In supports all of the data types that are supported by Oracle Streams, including LOBs, `LONG`, `LONG RAW`, and `XMLType`. A client application sends LOB and `XMLType` data to the inbound server in chunks. Several chunks comprise a single column value of LOB or `XMLType`.

Processed Low Position and Restart Ability

The processed low position is the position below which the inbound server no longer requires any LCRs. This position corresponds with the oldest SCN for an Oracle Streams apply process that applies changes captured by a capture process.

The processed low position indicates that the LCRs less than or equal to this position have been processed by the inbound server. If the client re-attaches to the inbound server, it only needs to send LCRs greater than the processed low position because the inbound server discards any LCRs that are less than or equal to the processed low position.

If the client application aborts abnormally, then the connection between the client application and the inbound server is automatically broken. Upon restart, the client application retrieves the processed low position from the inbound server and instructs its capture agent to retrieve changes starting from this processed low position.

To limit the recovery time of a client application using the XStream In interface, the client application can send activity, such as empty transactions, periodically to the inbound server. When there are no LCRs to send to the server, the client can send a row LCR with a commit directive to advance the inbound server's processed low position. This activity acts as an acknowledgment so that the inbound server's processed low position can be advanced. The LCR stream sent to an inbound server must follow the LCR stream properties for XStream Out defined above.

Stream Position

Stream position refers to the position of an LCR in a given LCR stream.

For transactions captured outside Oracle databases the stream position must be encoded in certain format (for example, base-16 encoding) that supports byte comparison. The stream position is key to the total order of transaction stream sent by clients using the XStream In interface.

Security of XStream

XStream Out allows regular users to receive LCRs without requiring system level privileges. System level privileges, such as DBA role, are required to configure XStream Out. The user who configures XStream Out can specify a regular user as the connect user who can attach to an outbound server to receive LCRs.

See Also: Chapter 3, "Configuring XStream" for more about configuring XStream

XStream In allows regular users to update tables in its own schema without requiring system level privileges (for example, DBA) to configure XStream In.

XStream cannot assume that the connected user to the inbound or outbound server is trusted.

OCI clients must connect to an Oracle database before attaching to an XStream outbound or inbound server created on that database. The connected user must be the same as the `connect_user` configured for the attached outbound server or the `apply_user` configured for the attached inbound server; otherwise, an error is raised.

XStream and Character Sets

XStream Out implicitly converts character data in logical change records (LCRs) from the outbound database character set to the client application character set. XStream In implicitly converts character data in LCRs from the client application character set to the inbound database character set. To improve performance, analyze the LCR data flow from the source to the destination, and set the client character set of the OCI client application to the one that minimizes character conversion, incurs no data loss, and takes advantage of the implicit conversion done by XStream or the destination. For XStream Out, in general, setting the client application character set to the outbound database character set is the best practice.

Handler and Descriptor Attributes

This appendix describes the attributes for OCI handles and descriptors, which can be read with `OCIAttrGet()`, and modified with `OCIAttrSet()`.

Conventions

For each handle type, the attributes which can be read or changed are listed. Each attribute listing includes the following information:

Mode
The following modes are valid:

READ - the attribute can be read using `OCIAttrGet()`

WRITE - the attribute can be modified using `OCIAttrSet()`

READ/WRITE - the attribute can be read using `OCIAttrGet()`, and it can be modified using `OCIAttrSet()`.

Description
This is a description of the purpose of the attribute.

Attribute Data Type
This is the data type of the attribute. If necessary, a distinction is made between the data type for READ and WRITE modes.

Server Handle Attributes

The following server handle attributes are available:

- OCI_ATTR_XSTREAM_ACK_INTERVAL
- OCI_ATTR_XSTREAM_IDLE_TIMEOUT

OCI_ATTR_XSTREAM_ACK_INTERVAL

Mode
READ/WRITE

Description
For XStream Out, the ACK interval is the minimum interval in seconds that the outbound server receives the processed low position from the client. After each ACK interval, the outbound server ends any in-progress `OCIXStreamOutLCRReceive()` or `OCIXStreamOutLCRCallbackReceive()` call so that the processed low position cached at the client can be sent to the server.

For XStream In, the ACK interval is the minimum interval in seconds that the inbound server sends the processed low position to the client. After each ACK interval, any in-progress `OCIXStreamInLCRSend()` or `OCIXStreamInLCRCallbackSend()` call is terminated for the inbound server to send a new processed low position to the client.

The default value for `OCI_ATTR_XSTREAM_ACK_INTERVAL` is 30 seconds. This attribute is checked only during the `OCIXStreamOutAttach()` or `OCIXStreamInAttach()` calls. Thus, it must be set before invoking these APIs; otherwise, the default value is used.

Attribute Data Type
`ub4 */ub4`

OCI_ATTR_XSTREAM_IDLE_TIMEOUT

Mode
READ/WRITE

Description
The idle timeout is the number of seconds of idle the outbound server waits for an LCR before terminating the `OCIXStreamOutLCRReceive()` or `OCIXStreamOutLCRCallbackReceive()` call.

The default for `OCI_ATTR_XSTREAM_IDLE_TIMEOUT` is one second. This attribute is checked only during the `OCIXStreamOutAttach()` or `OCIXStreamInAttach()` call. Thus, it must be set before invoking these APIs; otherwise, the default value is used.

Attribute Data Type
`ub4 */ub4`

12 OCI XStream Functions

This chapter describes the XStream functions for OCI. XStream stands for eXtended Streams. LCR stands for Logical Change Record.

Row LCR is used to encapsulate each row change. It includes the schema name, table name, DML operation, and the column values. For update operations, both before and after column values are included. The column data is in the format specified by the "Program Variable" column in Table 12–3. Character columns are converted to the client's character set.

DDL LCR is used to encapsulate DDL changes. It includes the object name, the DDL text, and the DDL command, for example, `ALTER TABLE`, and `TRUNCATE TABLE` command. See *Oracle Call Interface Programmer's Guide* for a list of DDL command codes.

See Also: *Oracle Database Globalization Support Guide* for more information about NLS settings.

XStream example programs are found in `xstream/oci` under the demo directory.

Each LCR also has a transaction id and position. For transactions captured outside Oracle databases, any byte-comparable `RAW` array can be used as the LCR position, if the position of each LCR in the stream is strictly increasing.

This chapter contains the topic:

- Introduction to XStream Functions

Introduction to XStream Functions

The functions are described using the following conventions:

Conventions for OCI Functions

For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

The function declaration.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described in Table 12–1:

Table 12–1 Mode of a Parameter

Mode	Description
IN	A parameter that passes data to the OCI.
OUT	A parameter that receives data from the OCI on this call.
IN/OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Comments

More detailed information about the function (if available), which can include return values, restrictions on the use of the function, examples, or other information that can be useful when using the function in an application.

OCI XStream Functions

This section describes the OCI XStream functions.

Table 12–2 OCI XStream Functions

Function	Purpose
LCR	To get and set value of an LCR. **Note:** These calls do not require a server round trip.
"OCILCRAttributesGet()" on page 12-5	Returns existing extra attributes from the LCR.
"OCILCRAttributesSet()" on page 12-7	Sets extra attributes in a ROW or DDL LCR.
"OCILCRFree()" on page 12-9	Frees the LCR.
"OCILCRHeaderGet()" on page 12-12	Returns the common header fields for ROW/DDL LCR.
"OCILCRHeaderSet()" on page 12-27	Initializes the common header fields for ROW or DDL LCR.
"OCILCRDDLInfoGet()" on page 12-10	Retrieves specific fields in a DDL LCR.
"OCILCRDDLInfoSet()" on page 12-24	Populates DDL-specific fields in a DDL LCR.
"OCILCRLobInfoGet()" on page 12-30	Returns the LOB information for a piece-wise LOB LCR.
"OCILCRLobInfoSet()" on page 12-32	Sets the LOB information for a piece-wise LOB LCR.
"OCILCRNew()" on page 12-17	Constructs a new Streams LCR object of the specified type (ROW or DDL) for the given duration.
"OCILCRRowColumnInfoGet()" on page 12-18	Returns the column fields in a ROW LCR.
"OCILCRRowColumnInfoSet()" on page 12-21	Populates column fields in a ROW LCR.
"OCILCRRowStmtGet()" on page 12-14	Returns the generated SQL statement for the row LCR, with values in-lined.
"OCILCRRowStmtWithBindVarGet()" on page 12-15	Returns the generated SQL statement, which uses bind variables for column values.
"OCILCRSCNsFromPosition()" on page 12-34	Gets the SCN and commit SCN from a position value.
"OCILCRSCNToPosition()" on page 12-35	Converts SCN to position.
"OCILCRWhereClauseGet()" on page 12-36	Gets the `WHERE` clause statement for the given ROW LCR.
"OCILCRWhereClauseWithBindVarGet()" on page 12-38	Gets the `WHERE` clause statement with bind variables for the given ROW LCR.
XStream In	To send an LCR stream to an XStream inbound server.
"OCIXStreamInAttach()" on page 12-40	Attaches to an inbound server.
"OCIXStreamInChunkSend()" on page 12-52	Sends chunk data to the inbound server.
"OCIXStreamInDetach()" on page 12-42	Detaches from the inbound server.
"OCIXStreamInFlush()" on page 12-51	Flushes the network while attaching to an XStream inbound server.
"OCIXStreamInLCRCallbackSend()" on page 12-45	Sends the LCR stream to the attached inbound server using callbacks.
"OCIXStreamInLCRSend()" on page 12-43	Sends the LCR stream to the attached inbound server using callbacks.
"OCIXStreamInProcessedLWMGet()" on page 12-50	Gets the local processed low-position.

Table 12–2 (Cont.) OCI XStream Functions

Function	Purpose
XStream Out	To receive an LCR stream from an XStream outbound server.
"OCIXStreamOutAttach()" on page 12-56	Attaches to an outbound server.
"OCIXStreamOutChunkReceive()" on page 12-67	Retrieves data of each LOB or `LONG` or `XMLType` column one chunk at a time.
"OCIXStreamOutDetach()" on page 12-58	Detaches from the outbound server.
"OCIXStreamOutLCRCallbackReceive()" on page 12-61	Gets the LCR stream from the outbound server using callbacks.
"OCIXStreamOutLCRReceive()" on page 12-59	Receives an LCR stream from an outbound server without using callbacks.
"OCIXStreamOutProcessedLWMSet()" on page 12-66	Updates the local copy of the processed low-water mark.

OCILCRAttributesGet()

Purpose

This function gets extra attribute information in (ROW or DDL) LCR, and any non-first class attributes that are not populated through `OCILCRHeaderGet()`, `OCILCRDDLInfoGet()`, or `OCILCRRowColumnInfoGet()`, for example, edition name.

Syntax

```
sword OCILCRAttributesGet (      OCISvcCtx   *svchp,
                                 OCIError    *errhp,
                                 ub2         *num_attrs
                                 oratext     **attr_names,
                                 ub2         *attr_namesl,
                                 ub2         *attr_dtyp,
                                 void        **attr_valuesp,
                                 OCIInd      *attr_indp,
                                 ub2         *attr_alensp,
                                 void        *lcrp,
                                 ub2         array_size,
                                 ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

num_attrs (OUT)
Number of extra attributes.

attr_names (OUT)
An array of extra attribute name pointers.

attr_namesl (OUT)
An array of extra attribute name lengths.

attr_dtyp (OUT)
An array of extra attribute data types. Valid data types: see Comments.

attr_valuesp (OUT)
An array of extra attribute data value pointers.

attr_indp (OUT)
An indicator array. Each returned element is an `OCIInd` value (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

attr_alensp (OUT)
An array of actual extra attribute data lengths. Each element in `alensp` is the length in bytes.

lcrp (IN)
Pointer to ROW or DDL LCR.

array_size (IN)
Size of the array argument in the other parameters. If `array_size` is not large enough to accommodate the number of attributes in the requested attribute list then `OCI_ERROR` is returned. Parameter `num_attrs` returns the expected size.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

The valid data types for `attr_dtyp` are:

```
SQLT_CHR
SQLT_INT
SQLT_RDD
```

OCILCRAttributesSet()

Purpose

This function populates extra attribute information in ROW or DDL LCR, and any non-first class attributes that cannot be set through `OCILCRHeaderSet()`, `OCILCRDDLInfoSet()`, or `OCILCRRowColumnInfoSet()`, for example. edition name.

Syntax

```
sword OCILCRAttributesSet (      OCISvcCtx   *svchp,
                                 OCIError    *errhp,
                                 ub2         num_attrs
                                 oratext     **attr_names,
                                 ub2         *attr_names_lens,
                                 ub2         *attr_dtyp,
                                 void        **attr_valuesp,
                                 OCIInd      *attr_indp,
                                 ub2         *attr_alensp,
                                 void        *lcrp,
                                 ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

num_attrs (IN)
Number of extra attributes.

attr_names (IN)
Pointer to an array of extra attribute names. Attribute names must be canonicalized.

attr_names_lens (IN)
Pointer to an array of extra attribute name lengths.

attr_dtyp (IN)
Pointer to an array of extra attribute data types. See valid data types in Comments of "OCILCRRowColumnInfoSet()" on page 12-21.

attr_valuesp (IN)
Address of an array of extra attribute data values.

attr_indp (IN)
Pointer to an indicator array. For all data types, this is a pointer to an array of `OCIInd` values (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

attr_alensp (IN)
Pointer to an array of actual extra attribute data lengths. Each element in `attr_lensp` is the length in bytes.

lcrp (IN/OUT)
Pointer to a ROW or DDL LCR.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Valid attributes are:

```
#define OCI_LCR_ATTR_THREAD_NO             "THREAD#"
#define OCI_LCR_ATTR_ROW_ID                "ROW_ID"
#define OCI_LCR_ATTR_SESSION_NO            "SESSION#"
#define OCI_LCR_ATTR_SERIAL_NO             "SERIAL#"
#define OCI_LCR_ATTR_USERNAME              "USERNAME"
#define OCI_LCR_ATTR_TX_NAME               "TX_NAME"
#define OCI_LCR_ATTR_EDITION_NAME          "EDITION_NAME"
#define OCI_LCR_ATTR_MESSAGE_TRACKING_LABEL "MESSAGE_TRACKING_LABEL"
```

OCILCRFree()

Purpose

Frees the LCR.

Syntax

```
sword OCILCRFree ( OCISvcCtx   *svchp,
                   OCIError    *errhp,
                   void        *lcrp,
                   ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

lcrp (IN/OUT)
Streams LCR pointer.

mode (IN)
Specify `OCI_DEFAULT` for now.

OCILCRDDLInfoGet()

Purpose

Retrieves specific fields in a DDL LCR.

Syntax

```
sword OCILCRDDLInfoGet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     **object_type,
                         ub2         *object_type_len,
                         oratext     **ddl_text,
                         ub4         *ddl_text_len,
                         oratext     **logon_user,
                         ub2         *logon_user_len,
                         oratext     **current_schema,
                         ub2         *current_schema_len,
                         oratext     **base_table_owner,
                         ub2         *base_table_owner_len,
                         oratext     **base_table_name,
                         ub2         *base_table_name_len,
                         oraub8      *flag,
                         void        *ddl_lcrp,
                         ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

object_type (OUT)
The type of object on which the DDL statement was executed. (See `OCILCRDDLInfoSet()`). Optional -- if not `NULL` then both `object_type` and `object_type_len` arguments must not be `NULL`.

object_type_len (OUT)
Length of the `object_type` string without the `NULL` terminator.

ddl_text (OUT)
The text of the DDL statement. Optional. If not `NULL` then both `ddl_text` and `ddl_text_len` arguments must not be `NULL`.

ddl_text_len (OUT)
DDL text length in bytes without the `NULL` terminator.

logon_user (OUT)
Canonicalized (follows a rule or procedure) name of the user whose session executed the DDL statement. Optional. If not `NULL` then both `logon_user` and `logon_user_len` arguments must not be `NULL`.

logon_user_len (OUT)
Length of the logon_user string without the `NULL` terminator.

current_schema (OUT)
The canonicalized schema name that is used if no schema is specified explicitly for the modified database objects in `ddl_text`. Optional. If not `NULL` then both `current_schema` and `current_schema_len` arguments must not be `NULL`.

current_schema_len (OUT)
Length of the `current_schema` string without the `NULL` terminator.

base_table_owner (OUT)
If the DDL statement is a table-related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_owner` specifies the canonicalized owner of the table involved. Otherwise, `base_table_owner` is `NULL`. Optional -- if not `NULL` then both `base_table_owner` and `base_table_owner_len` arguments must not be `NULL`.

base_table_owner_len (OUT)
Length of the `base_table_owner` string without the `NULL` terminator.

base_table_name (OUT)
If the DDL statement is a table-related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_name` specifies the canonicalized name of the table involved. Otherwise, `base_table_name` is `NULL`. Optional -- if not `NULL` then both `base_table_name` and `base_table_name_len` arguments must not be `NULL`.

base_table_name_len (OUT)
Length of the `base_table_name` string without the `NULL` terminator.

flag (OUT)
DDL LCR flag. Optional. Data not returned if argument is `NULL`. Future extension not used currently.

ddl_lcrp (IN)
DDL LCR. Cannot be `NULL`.

mode (IN)
Specify `OCI_DEFAULT` for now.

OCILCRHeaderGet()

Purpose

Returns the common header fields for ROW or DDL LCR. All returned pointers point directly to the corresponding LCR fields.

Syntax

```
sword OCILCRHeaderGet ( OCISvcCtx   *svchp,
                        OCIError    *errhp,
                        oratext     **src_db_name,
                        ub2         *src_db_name_len,
                        oratext     **cmd_type,
                        ub2         *cmd_type_len,
                        oratext     **owner,
                        ub2         *owner_len,
                        oratext     **oname,
                        ub2         *oname_len,
                        ub1         **tag,
                        ub2         *tag_len,
                        oratext     **txid,
                        ub2         *txid_len,
                        OCIDate     *src_time,
                        ub2         *old_columns,
                        ub2         *new_columns,
                        ub1         **position,
                        ub2         *position_len,
                        oraub8      *flag,
                        void        *lcrp,
                        ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

src_db_name (OUT)
Canonicalized source database name. Must be non-`NULL`.

src_db_name_len (OUT)
Length of the `src_db_name` string in bytes excluding the `NULL` terminator.

cmd_type (OUT)
For ROW LCRs: One of the following values:

```
#define OCI_LCR_ROW_CMD_INSERT
#define OCI_LCR_ROW_CMD_DELETE
#define OCI_LCR_ROW_CMD_UPDATE
#define OCI_LCR_ROW_CMD_COMMIT
#define OCI_LCR_ROW_CMD_LOB_WRITE
#define OCI_LCR_ROW_CMD_LOB_TRIM
#define OCI_LCR_ROW_CMD_LOB_ERASE
```

For DDL LCRs: One of the command types in *Oracle Call Interface Programmer's Guide*.

cmd_type_len (OUT)
Length of the `cmd_type` string in bytes excluding the `NULL` terminator.

owner (OUT)
Canonicalized table owner name. Must be non-`NULL`.

owner_len (OUT)
Length of the `owner` string in bytes excluding the `NULL` terminator.

oname (OUT)
Canonicalized table name. Must be non-`NULL`

oname_len (OUT)
Length of the `oname` string in bytes excluding the `NULL` terminator.

tag (OUT)
A binary tag that enables tracking of the LCR. For example, this tag can be used to determine the original source database of the DML statement if apply forwarding is used.

tag_len (OUT)
Number of bytes in the tag.

txid (OUT)
Transaction ID. Must be non-`NULL`

txid_len (OUT)
Length of the string in bytes excluding the `NULL` terminator.

src_time (OUT)
The time when the change was generated in the redo log of the source database.

old_columns (OUT)
Number of columns in the OLD column list. Returns 0 if the input LCR is a DDL LCR. Optional.

new_columns (OUT)
Number of columns in the NEW column list. Returns 0 if the input LCR is a DDL LCR. Optional.

position (OUT)
Position for LCR.

position_len (OUT)
Length of `position`.

flag (OUT)
LCR flag. Future extension currently not used.

lcrp (IN)
`lcrp` cannot be `NULL`.

mode (IN)
`OCILCR_NEW_ONLY_MODE` - If this mode is specified then the `new_columns` returned is the count of the columns in the NEW column list only. Otherwise, the `new_columns` returned is the number of distinct columns present in either the NEW or the OLD column list of the given ROW LCR.

OCILCRRowStmtGet()

Purpose

Returns the generated SQL statement for the row LCR, with values in-lined. Users must preallocate the memory for `sql_stmt`, and `*sql_stmt_len` must be set to the size of the allocated buffer, when it is passed in. If *`sql_stmt_len` is not large enough to hold the generated SQL statement, then an error is raised.

Syntax

```
sword OCILCRRowStmtGet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     *row_stmt,
                         ub4         *row_stmt_len,
                         void        *row_lcrp,
                         ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

row_stmt (IN/OUT)
The generated SQL statement for the ROW LCR.

row_stmt_len (IN/OUT)
Set to the size of the allocated buffer for `row_stmt` when passed in; returns the length of `row_stmt`.

row_lcrp (IN)
Pointer to ROW LCR.

mode (IN)
Specify `OCI_DEFAULT` for now.

OCILCRRowStmtWithBindVarGet()

Purpose

Returns the generated SQL statement, which uses bind variables for column values. The values for the bind variables are returned separately in arrays. You must preallocate the memory for `sql_stmt` and the arrays, `*sql_stmt_len` must be set to the size of the allocated buffer, and `array_size` must be the length of the arrays. The actual column values in `bind_var_valuesp` will point to the values inside the LCR, so it is a shallow copy. If `array_size` is not large enough to hold all the variables, or `*sql_stmt_len` is not large enough to hold the generated SQL statement, then an error is raised.

Syntax

```
sword OCILCRRowStmtWithBindVarGet ( OCISvcCtx    *svchp,
                                    OCIError     *errhp,
                                    oratext      *row_stmt,
                                    ub4          *row_stmt_len,
                                    ub2          *num_bind_var,
                                    ub2          *bind_var_dtyp,
                                    void         **bind_var_valuesp,
                                    OCIInd       *bind_var_indp,
                                    ub2          *bind_var_alensp,
                                    ub1          *bind_var_csetidp,
                                    ub1          *bind_var_csetfp,
                                    void         *row_lcrp,
                                    oratext      **chunk_column_names,
                                    ub2          *chunk_column_namesl,
                                    oraub8       *chunk_column_flags,
                                    ub2          array_size,
                                    oratext      *bind_var_syntax,
                                    ub4          mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

row_stmt (IN/OUT)
The generated SQL statement for the ROW LCR.

row_stmt_len (IN/OUT)
Set to the size of the allocated buffer for `row_stmt` when passed in; returns the length of `row_stmt`.

num_bind_var (OUT)
The number of bind variables.

bind_var_dtyp (IN/OUT)
Array of data types for the bind variables.

bind_var_valuesp (IN/OUT)
Array of values for the bind variables.

bind_var_indp (IN/OUT)
Array of NULL indicators for the bind variables.

bind_var_alensp (IN/OUT)
Array of lengths for the bind variable values.

bind_var_csetidp (IN/OUT)
Array of character set ids for the bind variables.

bind_var_csetfp (IN/OUT)
Array of character set forms for the bind variables.

row_lcrp (IN)
Pointer to ROW LCR.

chunk_column_names (OUT)
Array of LOB column names in LCR.

chunk_column_namesl (OUT)
Array of LOB column name lengths.

chunk_column_flags (OUT)
Array of LOB column flags. Possible flags are listed in Comments.

array_size (IN)
Size of each of the parameter arrays.

bind_var_syntax (IN)
Either ":" (binds will be of the form :1, :2, and so on.) or "?" (binds will be of the form "?").

mode (IN)
Specify OCI_DEFAULT for now.

Comments

LCR column flags can be combined using bitwise OR operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */
/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

OCILCRNew()

Purpose

Constructs a new Streams LCR object of the specified type (ROW or DDL) for the given duration.

Syntax

```
sword OCILCRNew ( OCISvcCtx     *svchp,
                  OCIError      *errhp,
                  OCIDuration   duration,
                  ub1           lcrtype,
                  void          **lcrp,
                  ub4           mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

duration (IN)
Memory for the LCR is allocated for this specified duration.

lcrtype (IN)
LCR type. Values are:

```
#define OCI_LCR_XROW
#define OCI_LCR_XDDL
```

lcrp (IN/OUT)
If `*lcrp` is not `NULL`, an error will be raised.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Note:

- Once created, you will not be allowed to change the type of the LCR (ROW or DDL) or duration of the memory allocation.
- Use `OCILCRHeaderSet()` to populate common header fields for ROW or DDL LCR.
- After the LCR header is initialized, use `OCILCRRowColumnInfoSet()` or `OCILCRDDLInfoSet()` to populate operation specific elements. Use `OCILCRExtraAttributesSet()` to populate extra attribute information.
- Use `OCILCRFree()` to free the LCR created by this function.

OCILCRRowColumnInfoGet()

Purpose

Returns the column fields in a ROW LCR.

Syntax

```
sword OCILCRRowColumnInfoGet ( OCISvcCtx   *svchp,
                               OCIError    *errhp,
                               ub2         column_value_type,
                               ub2         *num_columns,
                               oratext     **column_names,
                               ub2         *column_name_lens,
                               ub2         *column_dtyp,
                               void        **column_valuesp,
                               OCIInd      *column_indp,
                               ub2         *column_alensp,
                               ub1         *column_csetfp,
                               oraub8      *column_flags,
                               ub2         *column_csid,
                               void        *row_lcrp,
                               ub2         array_size,
                               ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

column_value_type (IN)
ROW LCR column value type; either of:

```
#define OCI_LCR_ROW_COLVAL_OLD
#define OCI_LCR_ROW_COLVAL_NEW
```

num_columns (OUT)
Number of columns in the specified column array.

column_names (OUT)
An array of column name pointers.

column_name_lens (OUT)
An array of column name lengths.

column_dtyp (OUT)
An array of column data types. Optional. Data is not returned if `column_dtyp` is `NULL`.

column_valuesp (OUT)
An array of column data pointers.

column_indp (OUT)
An array of indicators.

column_alensp (OUT)
An array of column lengths. Each returned element is the length in bytes.

column_csetfp (OUT)
An array of character set forms for the columns. Optional. Data is not returned if the argument is `NULL`.

column_flags (OUT)
An array of column flags. Optional. Data is not returned if the argument is `NULL`. See Comments for the values.

column_csid (OUT)
An array of character set ids for the columns.

row_lcrp (IN)
`row_lcrp` cannot be `NULL`.

array_size (IN)
Size of each of the parameter arrays. An error is returned if `array_size` is less than the number of columns in the requested column list. The actual size of the requested column list is returned through the `num_columns` parameter.

mode (IN)
`OCILCR_NEW_ONLY_MODE` - If this mode is specified then the `new_columns` returned is the count of the columns in the NEW column list only. Otherwise, the `new_columns` returned is the number of distinct columns present in either the NEW or the OLD column list of the given ROW LCR.

Comments

- For `INSERT`, this function must only be called to get the NEW column values.
- For `DELETE`, this function must only be called to get the OLD column values.
- For `UPDATE`, this function can be called twice, once to get the NEW column values and once to get the OLD column values.
- This function must not be called for `COMMIT` operations.

LCR column flags. Can be combined using bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */
/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

The following chart lists the currently supported table column data types. For each data type, it lists the corresponding LCR column data type, the C program variable type to cast the LCR column value, and the OCI functions that can be used to manipulate the column values returned from `OCILCRRowColumnInfoGet()`.

Table 12–3 Table Column Data Types

Table Column Data Types	LCR Column Data Type	Program Variable	Conversion Function
VARCHAR, NVARCHAR2	SQLT_CHR	char *	
NUMBER	SQLT_VNU	OCINumber	OCINumberToInt(), OCINumberToReal(), OCINumberToText()
DATE	SQLT_ODT	OCIDate	OCIDateToText() Can access structure directly to get date and time fields.
RAW	SQLT_BIN	unsigned char *	
CHAR, NCHAR	SQLT_AFC	char *	
BINARY_FLOAT	SQLT_BFLOAT	float	
BINARY_DOUBLE	SQLT_BDOUBLE	double	

TIMESTAMP	SQLT_TIMESTAMP	OCIDateTime *	OCIDateTimeGetTime()
			OCIDateTimeGetDate()
			OCIDateTimeGetTimeZoneOffset()
			OCIDateTimeToText()

TIMESTAMP WITH TIME ZONE	SQLT_TIMESTAMP_TZ	OCIDateTime *	OCIDateTimeGetTime()
			OCIDateTimeGetDate()
			OCIDateTimeGetTimeZoneOffset()
			OCIDateTimeToText()

TIMESTAMP WITH LOCAL TIME ZONE	SQLT_TIMESTAMP_LTZ	OCIDateTime *	OCIDateTimeGetTime()
			OCIDateTimeGetDate()
			OCIDateTimeGetTimeZoneOffset()
			OCIDateTimeToText()

INTERVAL YEAR TO MONTH	SQLT_INTERVAL_YM	OCIInterval *	OCIIntervalToText()
			OCIIntervalGetYearMonth()

INTERVAL DAY TO SECOND	SQLT_INTERVAL_DS	OCIInterval *	OCIIntervalToText()
			OCIIntervalGetDaySecond()

UROWID	SQLT_RDD	OCIRowid *	OCIRowidToChar()
CLOB	SQLT_CHR or SQLT_BIN	unsigned char *	*
NCLOB	SQLT_BIN	unsigned char *	*
BLOB	SQLT_BIN	unsigned char *	*
LONG	SQLT_CHR	char *	*
LONG RAW	SQLT_BIN	unsigned char *	*

* Call OCIXStreamOutChunkReceive() to get column data.

OCILCRRowColumnInfoSet()

Purpose

Populates column fields in a ROW LCR.

Syntax

```
sword OCILCRRowColumnInfoSet ( OCISvcCtx   *svchp,
                               OCIError    *errhp,
                               ub2         column_value_type,
                               ub2         num_columns,
                               oratext     **column_names,
                               ub2         *column_name_lens,
                               ub2         *column_dtyp,
                               void        **column_valuesp,
                               OCIInd      *column_indp,
                               ub2         *column_alensp,
                               ub1         *column_csetfp,
                               oraub8      *column_flags,
                               ub2         *column_csid,
                               void        *row_lcrp,
                               ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

column_value_type (IN)
ROW LCR Column value types:

```
#define OCI_LCR_ROW_COLVAL_OLD
#define OCI_LCR_ROW_COLVAL_NEW
```

num_columns (IN)
Number of columns in each of the array parameters.

column_names (IN)
Pointer to an array of column names. Column names must be canonicalized. Column names must follow Oracle naming conventions and size limitations.

column_name_lens (IN)
Pointer to an array of column name lengths.

column_dtyp (IN)
Pointer to an array of column data types. See Comments for valid data types.

column_valuesp (IN)
Pointer to an array of column data pointers.

column_indp (IN)
Pointer to an indicator array. For all data types, this is a pointer to an array of `OCIInd` values (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

column_alensp (IN)
Pointer to an array of actual column lengths in bytes.

column_csetfp (IN)
Pointer to an array of character set forms for the columns. The default form is `SQLCS_IMPLICIT`. Setting this attribute will cause the database or national character set to be used on the client side. Set this attribute to `SQLCS_NCHAR` for the national character set or `SQLCS_IMPLICIT` for the database character set. Pass 0 for non-character columns.

column_flags (IN)
Pointer to an array of column flags. (See Comments for the list of valid LCR column flags.)

column_csid (IN)
Pointer to an array of character set ids for the columns.

row_lcrp (IN/OUT)
`row_lcrp` cannot be `NULL`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Note:

- For `INSERT`, this function must only be called to specify the NEW column values.
- For `DELETE`, this function must only be called to specify the OLD column values.
- For `UPDATE`, this function can be called twice, once to specify the NEW column values and once to specify the OLD column values.
- This function must not be called for `COMMIT` operations.

Here are the LCR column flags. They can be combined using the bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */
/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

Valid data types are:

```
SQLT_AFC          SQLT_TIMESTAMP
SQLT_DAT          SQLT_TIMESTAMP_TZ
SQLT_BFLOAT       SQLT_TIMESTAMP_LTZ
SQLT_BDOUBLE      SQLT_INTERVAL_YM
SQLT_NUM          SQLT_INTERVAL_DS
SQLT_VCS
SQLT_ODT
SQLT_INT
```

```
SQLT_BIN
SQLT_CHR
SQLT_RDD
SQLT_VST
SQLT_INT
SQLT_FLT
```

OCILCRDDLInfoSet()

Purpose

Populates DDL-specific fields in a DDL LCR.

Syntax

```
sword OCILCRDDLInfoSet ( OCISvcCtx    *svchp,
                         OCIError     *errhp,
                         oratext      *object_type,
                         ub2          object_type_len,
                         oratext      *ddl_text,
                         ub4          ddl_text_len,
                         oratext      *logon_user,
                         ub2          logon_user_len,
                         oratext      *current_schema,
                         ub2          current_schema_len,
                         oratext      *base_table_owner,
                         ub2          base_table_owner_len,
                         oratext      *base_table_name,
                         ub2          base_table_name_len,
                         oraub8       flag,
                         void         *ddl_lcrp,
                         ub4          mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

object_type (IN)
The type of object on which the DDL statement was executed. See Comments for the valid object types.

object_type_len (IN)
Length of the `object_type` string without the `NULL` terminator.

ddl_text (IN)
The text of the DDL statement. This parameter must be set to a non-`NULL` value. DDL text must be in Oracle DDL format.

ddl_text_len (IN)
DDL text length in bytes without the `NULL` terminator.

logon_user (IN)
Canonicalized name of the user whose session executed the DDL statement.

logon_user_len (IN)
Length of the `logon_user` string without the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

current_schema (IN)
The canonicalized schema name that is used if no schema is specified explicitly for the modified database objects in `ddl_text`. If a schema is specified in `ddl_text` that differs from the one specified for `current_schema`, then the function uses the schema specified in `ddl_text`.

This parameter must be set to a non-`NULL` value.

current_schema_len (IN)
Length of the `current_schema` string without the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

base_table_owner (IN)
If the DDL statement is a table-related DDL (such as `CREATE TABLE` or `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_owner` specifies the canonicalized owner of the table involved. Otherwise, `base_table_owner` is `NULL`.

base_table_owner_len (IN)
Length of the `base_table_owner` string without the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

base_table_name (IN)
If the DDL statement is a table-related DDL (such as `CREATE TABLE` or `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_name` specifies the canonicalized name of the table involved. Otherwise, `base_table_name` is `NULL`.

base_table_name_len (IN)
Length of the `base_table_name` without the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

flag (IN)
DDL LCR flag. (For future extension – specify `OCI_DEFAULT` for now)

ddl_lcrp (IN/OUT)
`ddl_lcrp` cannot be `NULL`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

The following are valid object types:

```
CLUSTER
FUNCTION
INDEX
OUTLINE
PACKAGE
PACKAGE BODY
PROCEDURE
SEQUENCE
SYNONYM
TABLE
TRIGGER
TYPE
```

```
USER
VIEW
```

`NULL` is also a valid object type. Specify `NULL` for all object types not listed.

OCILCRHeaderSet()

Purpose

Initializes the common header fields for ROW or DDL LCR.

Syntax

```
sword OCILCRHeaderSet ( OCISvcCtx  *svchp,
                        OCIError   *errhp,
                        oratext    *src_db_name,
                        ub2        src_db_name_len,
                        oratext    *cmd_type,
                        ub2        cmd_type_len,
                        oratext    *owner,
                        ub2        owner_len,
                        oratext    *oname,
                        ub2        oname_len,
                        ub1        *tag,
                        ub2        tag_len,
                        oratext    *txid,
                        ub2        txid_len,
                        OCIDate    *src_time,
                        ub1        *position,
                        ub2        position_len,
                        oraub8     flag,
                        void       *lcrp,
                        ub4        mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

src_db_name (IN)
Canonicalized source database name. Must be non-`NULL`.

src_db_name_len (IN)
Length of the `src_db_name` string in bytes excluding the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

cmd_type (IN)
For ROW LCRs: One of the following values:

```
#define OCI_LCR_ROW_CMD_INSERT
#define OCI_LCR_ROW_CMD_DELETE
#define OCI_LCR_ROW_CMD_UPDATE
#define OCI_LCR_ROW_CMD_COMMIT
#define OCI_LCR_ROW_CMD_LOB_WRITE
#define OCI_LCR_ROW_CMD_LOB_TRIM
#define OCI_LCR_ROW_CMD_LOB_ERASE
```

For DDL LCRs: One of the command types in *Oracle Call Interface Programmer's Guide*.

cmd_type_len (IN)
Length of `cmd_type`.

owner (IN)
Canonicalized table owner name. Owner is not required for `COMMIT` LCR.

owner_len (IN)
Length of the `owner` string in bytes excluding the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

oname (IN)
Canonicalized table name. Owner is not required for `COMMIT` LCR.

oname_len (IN)
Length of the `oname` string in bytes excluding the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

tag (IN)
A binary tag that enables tracking of the LCR. For example, this tag can be used to determine the original source database of the DML statement if apply forwarding is used.

tag_len (IN)
Number of bytes in the tag. Cannot exceed 2000 bytes.

txid (IN)
Transaction ID. Must be non-`NULL`.

txid_len (IN)
Length of the `txid` string in bytes, excluding the `NULL` terminator. Must follow Oracle naming conventions and size limitations.

src_time (IN)
The time when the change was generated in the redo log of the source database.

position (IN)
Position for LCR. Must be non-`NULL` and byte-comparable.

position_len (IN)
Length of position. Must be greater than zero.

flag (IN)
LCR flag. Future extension not used currently.

lcrp (IN/OUT)
`lcrp` cannot be `NULL`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Note:

- This function will set all internal fields of the LCR to `NULL` including extra attributes.
- This function *does not* deep copy the passed-in values. You must ensure data is valid for the duration of the LCR.

- For `COMMIT` LCRs, `owner` and `oname` information are not required. Provide valid values for `src_db_name`, `cmd_type`, `tag`, `txid`, and `position`.
- For ROW LCRs, use `OCILCRRowColumnInfoSet()` to populate ROW LCR specific-column information.
- For DDL LCRs, use `OCILCRDDLInfoSet()` to populate DDL operation specific information.
- For Row or DDL LCRs, use `OCILCRAttributesSet()` to populate extra attribute information.

OCILCRLobInfoGet()

Purpose

Returns the LOB information for a piece-wise LOB LCR generated from a DBMS_LOB or OCILob procedure.

Syntax

```
sword OCILCRLobInfoGet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     **column_name,
                         ub2         *column_name_len,
                         ub2         *column_dty,
                         oraub8      *column_flag,
                         ub4         *offset,
                         ub4         *size,
                         void        *row_lcrp,
                         ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

column_name (OUT)
LOB column name.

column_name_len (OUT)
Length of LOB column name.

column_dty (OUT)
Column data type (either `SQLT_CHR` or `SQLT_BIN`).

column_flag (OUT)
Column flag. See Comments in "OCILCRRowColumnInfoSet()" on page 12-21.

offset (OUT)
LOB operation offset in code points. Only returned for `LOB WRITE` and `LOB TRIM` operations. This is the same as the `offset` parameter for `OCILobErase()` or the `offset` parameter in `OCILobWrite()`.

size (OUT)
LOB operation size in code points. Only returned for `LOB TRIM` and `LOB ERASE` operations.This is the same as the `new_length` parameter in `OCILobTrim()` or the `amtp` parameter in `OCILobErase()`.

row_lcrp (IN)
Pointer to a ROW LCR.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Returns `OCI_SUCCESS` or `OCI_ERROR`.

OCILCRLobInfoSet()

Purpose

Sets the LOB information for a piece-wise LOB LCR. This call is valid when the input LCR is a LOB_WRITE, LOB_ERASE or LOB_TRIM; otherwise, an error is returned.

Syntax

```
sword OCILCRLobInfoSet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     *column_name,
                         ub2         column_name_len,
                         ub2         column_dty,
                         oraub8      column_flag,
                         ub4         offset,
                         ub4         size,
                         void        *row_lcrp,
                         ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

column_name (IN)
LOB column name.

column_name_len (IN)
Length of LOB column name.

column_dty (IN)
Column data type (either SQLT_CHR or SQLT_BIN).

column_flag (IN)
Column flag. See Comments in "OCILCRRowColumnInfoSet()" on page 12-21.

offset (IN)
LOB operation offset in code points. Only required for LOB WRITE and LOB TRIM operations. This is the same as the soffset parameter for OCILobErase() or the offset parameter in OCILobWrite().

size (IN)
LOB operation size in code points. Only required for LOB TRIM and LOB ERASE operations.This is the same as the new_length parameter in OCILobTrim() or the amtp parameter in OCILobErase().

row_lcrp (IN/OUT)
Pointer to a ROW LCR.

mode (IN)
Specify OCI_DEFAULT for now.

Comments

Returns `OCI_SUCCESS` or `OCI_ERROR`.

OCILCRSCNsFromPosition()

Purpose

Returns the SCN and the commit SCN from the position value. The input position must be one that is obtained from an XStream outbound server. An error is returned if the input position does not conform to the expected format.

Syntax

```
sword OCILCRSCNsFromPosition ( OCISvcCtx   *svchp,
                               OCIError    *errhp,
                               ub1         *position,
                               ub2         position_len,
                               OCINumber   *scn,
                               OCINumber   *commit_scn,
                               ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

position (IN)
LCR position value.

position_len (IN)
Length of LCR position value.

scn (OUT)
SCN number embedded in the given LCR position.

commit_scn (OUT)
The commit SCN embedded in the given position.

mode (IN)
Mode flags for future expansion. Specify `OCI_DEFAULT` for now.

OCILCRSCNToPosition()

Purpose

Converts an SCN to a position. The generated position can be passed as the `last_position` to `OCIXStreamOutAttach()` to filter the LCRs with commit SCN less than the given SCN and the LCR's SCN less than the given SCN. Therefore, the first LCR sent by the outbound server is either:

1. A commit LCR at the given SCN, or

2. The first LCR of the subsequent transaction with commit SCN greater than or equal to the given SCN.

Syntax

```
sword OCILCRSCNToPosition ( OCISvcCtx  *svchp,
                            OCIError   *errhp,
                            ub1        *position,
                            ub2        *position_len,
                            OCINumber  *scn,
                            ub4        mode );
```

Parameters

svchp (IN)
OCI service context.

errhp (IN)
OCI error handle.

position (OUT)
The resulting position. You must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes.

position_len (OUT)
Length of `position`.

scn (IN)
The SCN to be stored in `position`.

mode (IN)
Mode flags (for future extension).

Comments

Returns `OCI_SUCCESS` if the conversion succeeds, `OCI_ERROR` otherwise.

OCILCRWhereClauseGet()

Purpose

Gets the WHERE clause statement for the given ROW LCR.

Syntax

```
sword OCILCRWhereClauseGet ( OCISvcCtx  *svchp,
                             OCIError   *errhp,
                             oratext    *wc_stmt,
                             ub4        *wc_stmt_len,
                             void       *row_lcrp,
                             ub4        mode );
```

Parameters

svchp (IN/OUT)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

wc_stmt (OUT)
SQL statement equivalent to the LCR.

wc_stmt_len (IN/OUT)
Length of the wc_stmt buffer.

row_lcrp (IN)
Row LCR to be converted to SQL.

mode (IN)
Mode flags for future expansion. For now, specify OCI_DEFAULT.

Comments

WHERE clause generated for an INSERT LCR will have all the columns that are being inserted. This WHERE clause could be used to identify the inserted row after inserting. (Like "returning ROWID").

```
INSERT INTO TAB(COL1) VALUES (10) -> WHERE COL1=10
```

WHERE clause generated for UPDATE will have all the columns in the old column list. However the values of the columns will be that of the new value if it exists in the new column list of the UPDATE. If the column does not have a new value then the old column value will be used.

```
UPDATE TAB SET COL1 = 10 WHERE COL1 = 20 -> WHERE COL1 = 10
UPDATE TAB SET COL2 = 20 WHERE COL1 = 20 -> WHERE COL1 = 20
```

WHERE clause for DELETE will use the columns and values from the old column list.

LOB piecewise operations use the new columns and values for generating the WHERE clause.

Returns

`OCI_SUCCESS` or `OCI_ERROR`.

OCILCRWhereClauseWithBindVarGet()

Purpose

Gets the `WHERE` clause statement with bind variables for the given ROW LCR.

Syntax

```
sword OCILCRWhereClauseWithBindVarGet ( OCISvcCtx  *svchp,
                                        OCIError   *errhp,
                                        oratext    *wc_stmt,
                                        ub4        *wc_stmt_len,
                                        ub2        *num_bind_var,
                                        ub2        *bind_var_dtyp,
                                        void       **bind_var_valuesp,
                                        OCIInd     *bind_var_indp,
                                        ub2        *bind_var_alensp,
                                        ub2        *bind_var_csetidp,
                                        ub1        *bind_var_csetfp,
                                        void       *row_lcrp,
                                        ub2        array_size,
                                        oratext    *bind_var_syntax,
                                        ub4        mode );
```

Parameters

svchp (IN/OUT)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

wc_stmt (OUT)
SQL statement equivalent to the LCR.

wc_stmt_len (IN/OUT)
Length of the `wc_stmt` buffer.

num_bind_var (OUT)
Number of bind variables.

bind_var_dtyp (OUT)
Array of data types of bind variables.

bind_var_valuesp (OUT)
Array of values of bind variables.

bind_var_indp (OUT)
Array of `NULL` indicators of bind variables.

bind_var_alensp (OUT)
Array of lengths of bind values.

bind_var_csetidp (OUT)
Array of char set ids of binds.

bind_var_csetfp (OUT)
Array of char set forms of binds.

row_lcrp (IN)
Row LCR to be converted to SQL.

array_size (IN)
Size of the array of bind values.

bind_var_syntax (IN)
Native syntax to be used for binds.

mode (IN)
Mode flags for future expansion. For now, specify `OCI_DEFAULT`.

Comments

If `array_size` is not large enough to accommodate the number of columns in the requested column list then `OCI_ERROR` is returned. The expected `array_size` is returned through the `num_bind_var` parameter.

`bind_var_syntax` for Oracle Database should contain ":". This will generate positional binds such as :1, :2, :3, and so on. For non-Oracle databases input the string that needs to be used for binds.

The `WHERE` clause generated for `INSERT` LCR will have all the columns that are being inserted. This `WHERE` clause can be used to identify the inserted row after inserting. (like "returning `ROWID`").

```
INSERT INTO TAB(COL1) VALUES (10) -> WHERE COL1=10
```

The `WHERE` clause generated for `UPDATE` will have all the columns in the old column list. However the values of the columns will be that of the new column value of the column if it exists in the new values of the `UPDATE`. If the column appears only in the old column then the old column value will be used.

```
UPDATE TAB SET COL1 = 10 WHERE COL1 = 20 -> WHERE COL1 = 10
UPDATE TAB SET COL2 = 20 WHERE COL1 = 20 -> WHERE COL1 = 20
```

The `WHERE` clause for `DELETE` will use the columns and values from the old column list.

LOB piecewise operations will use the new columns and values for generating the `WHERE` clause.

Returns

`OCI_SUCCESS` or `OCI_ERROR`.

OCIXStreamInAttach()

Purpose

Attaches to an inbound server.

Syntax

```
sword OCIXStreamInAttach ( OCISvcCtx  *svchp,
                           OCIError   *errhp,
                           oratext    *server_name,
                           ub2        server_name_len,
                           oratext    *source_name,
                           ub2        source_name_len,
                           ub1        *last_position,
                           ub2        *last_position_len,
                           ub4        mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

server_name (IN)
XStream inbound server name.

server_name_len (IN)
Length of the XStream inbound server name.

source_name (IN)
Source name to identify the data source.

source_name_len (IN)
Source name length.

last_position (OUT)
Last position received by inbound server. Optional. If specified, then you must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes for the return value.

last_position_len (OUT)
Length of `last_position`. Must be non-`NULL` if `last_position` is non-`NULL`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

The name of the inbound server must be provided because multiple inbound servers can be configured in one Oracle instance. This function returns `OCI_ERROR` if any error encountered while attaching to the inbound server. Only one client can attach to an XStream inbound server at any time. An error is returned if multiple clients attempt to attach to the same inbound server or if the same client attempts to attach to multiple inbound servers concurrently.

This function returns the last position received by the inbound server. Having successfully attached to the server, the client should resume sending LCRs with positions greater than this `last_position` since the inbound server will discard all LCRs with positions less than or equal to the `last_position`.

Returns either `OCI_SUCCESS` or `OCI_ERROR` status code.

OCIXStreamInDetach()

Purpose

Detaches from the inbound server.

Syntax

```
sword OCIXStreamInDetach ( OCISvcCtx  *svchp,
                           OCIError   *errhp,
                           ub1        *processed_low_position,
                           ub2        *processed_low_position_len
                           ub4        mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

processed_low_position (OUT)
The server's processed low position.

processed_low_position (OUT)
Length of `processed_low_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

You must pass in a preallocated buffer for the position argument. The maximum length of this buffer is `OCI_LCR_MAX_POSITION_LEN`. This position is exposed in `DBA_XSTREAM_INBOUND_PROGRESS` view

This call returns the server's processed low-position. If this function is invoked while a `OCIXStreamInLCRSend()` call is in progress, then it immediately terminates that call before detaching from the inbound server.

Returns either `OCI_SUCCESS` or `OCI_ERROR` status code.

OCIXStreamInLCRSend()

Purpose

Sends an LCR stream from the client to the attached inbound server. To avoid a network round trip for every `OCIXStreamInLCRSend()` call, we will tie the connection to this call and terminate the call after ACK interval since we initiate the LCR stream to the server.

Syntax

```
sword OCIXStreamInLCRSend ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            void        *lcrp,
                            ub1         lcrtype,
                            oraub8      flag,
                            ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

lcrp (IN)
Pointer to the new LCR to send. It cannot be `NULL`.

lcrtype (IN)
LCR type. Either of:

```
#define OCI_LCR_XROW
#define OCI_LCR_XDDL
```

flag (IN)
If bit `OCI_XSTREAM_MORE_ROW_DATA` (0x01) is set, it means this LCR contains more chunk data. You must call `OCIXStreamInChunkSend()` before calling `OCIXStreamInLCRSend()` again.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Return codes are:

- `OCI_STILL_EXECUTING` means that the current call is still in progress. The connection associated with the specified service context handle is still tied to this call for streaming the LCRs to the server. An error is returned if you attempt to use the same connection to execute any OCI calls that require database round trip, for example, `OCIStmtExecute()`, `OCIStmtFetch()`, `OCILobRead()`, and so on. `OCILCR*` calls are local calls; thus, they are valid while this call is in progress.
- `OCI_SUCCESS` means the current call is completed. You can execute `OCIStmt*`, `OCILob*`, and so on from the same service context.

- `OCI_ERROR` means this call encounters some errors. Use `OCIErrorGet()` to obtain information about the error.

See Also: "Server Handle Attributes" on page 11-4

OCIXStreamInLCRCallbackSend()

Purpose

Sends an LCR stream to the attached inbound server. You must specify a callback to construct each LCR for streaming. If some LCRs contain chunk data, then a second callback must be provided to create each chunk data.

Syntax

```
sword OCIXStreamInLCRCallbackSend (
         OCISvcCtx                      *svchp,
         OCIError                       *errhp,
         OCICallbackXStreamInLCRCreate   createlcr_cb,
         OCICallbackXStreamInChunkCreate createchunk_cb,
         void                           *usrctxp,
         ub4                            mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

createlcr_cb (IN)
Client callback procedure to be invoked to generate an LCR for streaming. Cannot be `NULL`.

createchunk_cb (IN)
Client callback procedure to be invoked to create each chunk. Can be `NULL` if you do not need to send any LCR with LOB or `LONG` or `XMLType` columns. `OCI_ERROR` will be returned if this argument is `NULL` and you attempt to send an LCR with additional chunk data.

usrctxp (IN)
User context to pass to both callback functions.

mode (IN)
Specify `OCI_DEFAULT` fore now.

Comments

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

The `createlcr_cb` argument must be of type `OCICallbackXStreamInLCRCreate`:

```
typedef sb4  (*OCICallbackXStreamInLCRCreate)
              void  *usrctxp, void **lcrp, ub1 *lcrtyp, oraub8 *flag);
```

Parameters of `OCICallbackXStreamInLCRCreate()`:

usrctxp (IN/OUT)
Pointer to the user context.

lcrp (OUT)
Pointer to the LCR to be sent.

lcrtyp (OUT)
LCR type (OCI_LCR_XROW or OCI_LCR_XDDL).

flag (OUT)
If OCI_XSTREAM_MORE_ROW_DATA is set, then the current LCR has more chunk data.

The input parameter to the callback is the user context. The output parameters are the new LCR, its type, and a flag. If the generated LCR contains additional chunk data then this flag must have OCI_XSTREAM_MORE_ROW_DATA (0x01) bit set. The valid return codes from OCICallbackXStreamInLCRCreate() callback function are OCI_CONTINUE or OCI_SUCCESS. This callback function must return OCI_CONTINUE to continue processing OCIXStreamInLCRCallbackSend() call. Any return code other than OCI_CONTINUE signals that the client wants to terminate OCIXStreamInLCRCallbackSend() call immediately. In addition, a NULL LCR returned from OCICallbackXStreamInLCRCreate() callback function signals that the client wants to terminate the current call.

The createchunk_cb argument must be of type OCICallbackXStreamInChunkCreate:

```
typedef sb4 (*OCICallbackXStreamInChunkCreate)
void     *usrctxp,
oratext  **column_name,
ub2      *column_name_len,
ub2      *column_dty,
oraub8   *column_flag,
ub2      *column_csid,
ub4      *chunk_bytes,
ub1      **chunk_data,
oraub8   *flag);
```

The input parameters of the createchunk_cb() procedure are the user context and the information about the chunk.

Parameters of OCICallbackXStreamInChunkCreate():

usrctxp (IN/OUT)
Pointer to the user context.

column_name (OUT)
Column name of the current chunk.

column_name_len (OUT)
Length of the column name.

column_name_dty (OUT)
Chunk data type (SQLT_CHR or SQLT_BIN).

column_flag (OUT)
See Comments in "OCIXStreamInChunkSend()" on page 12-52.

column_csid (OUT)
Column character set id. Relevant only if the column is an XMLType column (that is, column_flag has OCI_LCR_COLUMN_XML_DATA bit set).

chunk_bytes (OUT)
Chunk data length in bytes.

chunk_data (OUT)
Chunk data pointer.

flag (OUT)
If `OCI_XSTREAM_MORE_ROW_DATA` is set, this means the current LCR has more chunk data.

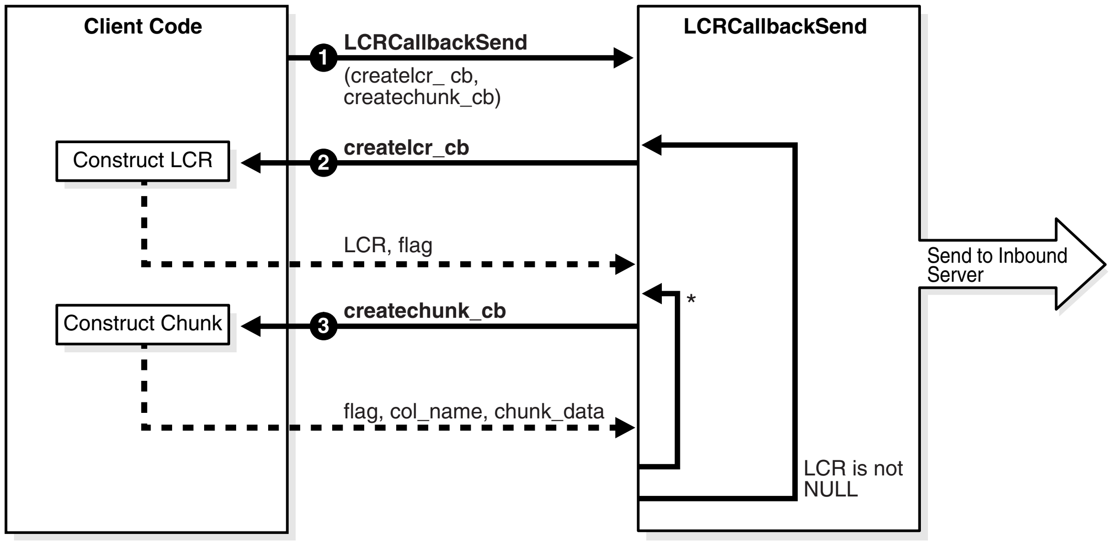
The `OCIXStreamInLCRCallbackSend()` function will invoke `createlcr_cb()` procedure to obtain each LCR to send to the server. If the return flag from the `createlcr_cb()` procedure has `OCI_XSTREAM_MORE_ROW_DATA` bit set, then it will invoke `createchunk_cb()` procedure to obtain each chunk. It repeatedly calls `createchunk_cb()` procedure while the flag returned from this callback has `OCI_XSTREAM_MORE_ROW_DATA` bit set. When this bit is not set, this function will cycle back to invoke `createlcr_cb()` procedure to get the next LCR. This cycle is repeated until the `createlcr_cb()` procedure returns a `NULL` LCR or when at the transaction boundary after an ACK interval has elapsed since the call begins.

The valid return codes from `OCICallbackXStreamInChunkCreate()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing `OCIXStreamInLCRCallbackSend()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate `OCIXStreamInLCRCallbackSend()` call immediately.

Since terminating the current call will flush the network and incur another network round trip in the next call, you must avoid returning a `NULL` LCR immediately when there is no LCR to send. Doing this can greatly reduce network throughput and affect performance. During short idle period, you can add some delays in the callback procedure instead of returning a `NULL` LCR immediately to avoid flushing the network too frequently.

The following figure shows the execution flow of the `OCIXStreamInLCRCallbackSend()` function:

Figure 12–1 Execution Flow of the OCIXStreamInLCRCallbackSend() Function



* `While OCI_XSTREAM_MORE_ROW_DATA` is set

Description of the figure:

- At 1, the user invokes `OCIXStreamInLCRCallbackSend()` providing two callbacks. This function initiates an LCR inbound stream to the server.
- At 2, this function invokes `createlcr_cb()` procedure to get an LCR from the callback to send to the server. If the return LCR is `NULL` then this function exits.
- If the flag from 2 indicates the current LCR has more data (that is, `OCI_XSTREAM_MORE_ROW_DATA` bit is set) then this function proceeds to 3; otherwise, it loops back to 2 to get the next LCR.
- At 3, this function invokes `createchunk_cb()` to get the chunk data to send to the server. If the flag from this callback has `OCI_XSTREAM_MORE_ROW_DATA` bit set, then it repeats 3; otherwise, it loops back to 2 to get the next LCR from the user. If any callback function returns any values other than `OCI_CONTINUE` then the `OCIXStreamInLCRCallbackSend()` call terminates.

  Following is a sample client pseudocode snippet for callback mode (error checking is not included for simplicity):

```
main
{
   /* Attach to inbound server */
   OCIXStreamInAttach();

   /* Get the server's processed low-position to determine
    * the position of the first LCR to generate.
    */
   OCIXStreamInProcessedLWMGet(&lwm);

   while (TRUE)
   {
      /* Initiate LCR inbound stream */
      OCIXStreamInLCRCallbackSend(createlcr_cb, createchunk_cb);

      OCIXStreamInProcessedLWMGet(&lwm);

      if (some terminating condition)
         break;
   }
   OCIXStreamInDetach(&lwm);
}


createlcr_cb (IN usrctx, OUT lcr, OUT flag)
{
   if (have more LCRs to send)
   {
      /* construct lcr */
      OCILCRHeaderSet(lcr);
      OCILCRRowColumnInfoSet(lcr);

      if (lcr has LOB | LONG | XMLType columns)
         Set OCI_XSTREAM_MORE_ROW_DATA flag;

      if (lcr is LOB_ERASE | LOB_TRIM | LOB_WRITE)
         OCILCRLobInfoSet(lcr);
   }
   else if (idle timeout expires)
   {
      lcr = null;
   }
}

createchunk_cb (IN usrctx, OUT chunk, OUT flag)
{
   /* set col_name, col_flag, chunk data, etc. */
   construct_chunk;

   if (last chunk of current column)
   {
      set col_flag |= OCI_LCR_COLUMN_LAST_CHUNK;

      if (last column)
         clear OCI_XSTREAM_MORE_ROW_DATA flag;
   }
}
```

OCIXStreamInProcessedLWMGet()

Purpose

Gets the local processed low-position that is cached at the client. It can be called anytime while the client is attached to an XStream inbound server. Clients, using the callback mode to stream LCRs to the server (see "OCIXStreamInLCRCallbackSend()" on page 12-45), can invoke this function while in the callback procedures.

Syntax

```
sword OCIXStreamInProcessedLWMGet ( OCISvcCtx  *svchp,
                                    OCIError   *errhp,
                                    ub1        *processed_low_position,
                                    ub2        *processed_low_position_len,
                                    ub4        mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

processed_low_position (OUT)
The processed low position maintained at the client.

processed_low_position_len (OUT)
Length of `processed_low_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

After attaching to an XStream inbound server, a local copy of the server's processed low-position (see "OCIXStreamOutProcessedLWMSet()" on page 12-66) is cached at the client. This local copy is refreshed with the server's low-position when each of the following calls returns `OCI_SUCCESS`:

- `OCIXStreamInAttach()`
- `OCIXStreamInLCRSend()`
- `OCIXStreamInLCRCallbackSend()`
- `OCIXStreamInFlush()`

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

You must pass in a preallocated buffer for the position argument. The maximum length of this buffer is `OCI_LCR_MAX_POSITION_LEN`. This position is exposed in the `DBA_XSTREAM_INBOUND_PROGRESS` view.

The client can use this position to periodically purge the logs used to generate the LCRs at or below this position.

OCIXStreamInFlush()

Purpose

Used to flush the network while attaching to an XStream inbound server. It terminates any in-progress `OCIXStreamInLCRSend()` call associated with the specified service context.

Syntax

```
sword OCIXStreamInFlush ( OCISvcCtx    *svchp,
                          OCIError     *errhp,
                          ub4          mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

Each call will incur a database round trip to get the server's processed low-position, which you can retrieve afterward using `OCIXStreamInProcessedLWMGet()`. This function should be called only when there is no LCR to send to the server and the client wants to know the progress of the attached inbound server.

This call returns `OCI_ERROR` if it is invoked from the callback functions of `OCIXStreamInLCRCallbackSend()`.

OCIXStreamInChunkSend()

Purpose

Sends a chunk to the inbound server. This function is valid during the execution of the `OCIXStreamInLCRSend()` call.

Syntax

```
sword OCIXStreamInChunkSend (  OCISvcCtx    *svchp,
                               OCIError     *errhp,
                               oratext      *column_name,
                               ub2          column_name_len,
                               ub2          column_dty,
                               oraub8       column_flag,
                               ub2          column_csid,
                               ub4          chunk_bytes,
                               ub1          *chunk_data,
                               oraub8       flag,
                               ub4          mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

column_name (IN)
Name of column associated with the given data. Column name must be canonicalized and must follow Oracle naming convention.

column_name_len (IN)
Length of column name.

column_dty (IN)
LCR chunk data type (must be `SQLT_CHR` or `SQLT_BIN`). See Table 12–5, " Storage of LOB or LONG Data in the LCR"

column_flag (IN)
Column flag. (See Comments for valid column flags). Must specify `OCI_LCR_COLUMN_LAST_CHUNK` for the last chunk of each LOB or `LONG` or `XMLType` column.

column_csid (IN)
Column character set id. This is required only if the `column_flag` has `OCI_LCR_COLUMN_XML_DATA` bit set.

chunk_bytes (IN)
Chunk data length in bytes.

chunk_data (IN)
Pointer to column data chunk. If the column is `NCLOB` or varying width `CLOB`, the input chunk data must be in `AL16UTF16` format. The chunk data must be in the character set defined in " Storage of LOB or LONG Data in the LCR" on page 12-68.

flag (IN)
If OCI_XSTREAM_MORE_ROW_DATA (0x01) bit is set, it means the current row change contains more data. You must clear this bit when sending the last chunk of the current LCR.

mode (IN)
Specify OCI_DEFAULT for now.

Comments

LCR column flags. Can be combined using bitwise OR operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */
```

In Streams, LOB, LONG, or XMLType column data is broken up into multiple chunks. For a row change containing columns of these data types, its associated LCR only contains data for the other column types. All LOB, LONG or XMLType columns are either represented in the LCR as NULL or not included in the LCR as defined in Table 12–4, " Required Column List in the First LCR".

OCILCRRowColumnInfoSet() is provided to generate a list of scalar columns in an LCR. For LOB, LONG, and XMLType columns, OCIXStreamInChunkSend() is provided to set the value of each chunk in a column. For a large column, this function can be invoked consecutively multiple times with smaller chunks of data. The XStream inbound server can assemble these chunks and apply the accumulated change to the designated column.

The LCR of a row change must contain all the scalar columns which can uniquely identify a row at the apply site. Table 12–4 describes the required column list in each LCR for each DML operation.

Table 12–4 Required Column List in the First LCR

Command Type of the First LCR of a Row Change	Columns Required in the First LCR
INSERT	The NEW column list must contain all non-NULL scalar columns. All LOB, XMLType, and LONG columns with chunk data must be included in this NEW column list. Each must have NULL value and OCI_LCR_COLUMN_EMPTY_LOB flag specified.
UPDATE	The OLD column list must contain the key columns. The NEW column list must contain all updated scalar columns. All LOB, XMLType, and LONG columns with chunk data must be included in this NEW column list. Each must have NULL value and OCI_LCR_COLUMN_EMPTY_LOB flag specified.
DELETE	The OLD column list must contain the key columns.
LOB_WRITE, LOB_TRIM, LOB_ERASE	The NEW column list must contain the key columns and the modified LOB column.

After constructing each LCR, you can call `OCIXStreamInLCRSend()` to send that LCR. Then `OCIXStreamInChunkSend()` can be called repeatedly to send the chunk data for each LOB or `LONG` or `XMLType` column in that LCR. Sending the chunk value for different columns cannot be interleaved. If a column contains multiple chunks, this function must be called consecutively using the same column name before proceeding to a new column. The ordering of the columns is irrelevant.

When invoking this function, you must pass `OCI_XSTREAM_MORE_ROW_DATA` as the flag argument if there is more data for the current LCR. When sending the last chunk of the current LCR, this flag must be cleared to signal the end of the current LCR.

This function is valid only for `INSERT`, `UPDATE`, and `LOB_WRITE` operations. Multiple LOB, `LONG`, or `XMLType` columns can be specified for `INSERT` and `UPDATE`, while only one LOB column is allowed for `LOB_WRITE` operation.

Following is a sample client pseudocode snippet for non-callback mode (error checking is not included for simplicity):

```
main
{
   /* Attach to inbound server */
   OCIXStreamInAttach();

   /* Get the server's processed low-position to determine
    * the position of the first LCR to generate.
    */
   OCIXStreamInProcessedLWMGet(&lwm);

   while (TRUE)
   {
      flag = 0;
      /* construct lcr */
      OCILCRHeaderSet(lcr);
      OCILCRRowColumnInfoSet(lcr);

      if (lcr has LOB | LONG | XMLType columns)
         set OCI_XSTREAM_MORE_ROW_DATA flag;

      status = OCIXStreamInLCRSend(lcr, flag);

      if (status == OCI_STILL_EXECUTING &&
            (OCI_XSTREAM_MORE_ROW_DATA flag set))
      {
         for each LOB/LONG/XMLType column in row change
         {
            for each chunk in column
            {
               /* set col_name, col_flag, chunk data */
               construct chunk;

               if (last chunk of current column)
                   col_flag |= OCI_LCR_COLUMN_LAST_CHUNK;

               if (last chunk of last column)
                  clear OCI_XSTREAM_MORE_ROW_DATA flag;

               OCIXStreamInChunkSend(chunk, col_flag, flag);
            }
         }
      }
      else if (status == OCI_SUCCESS)
      {
         /* get lwm when SendLCR call ends successfully. */
         OCIXStreamInProcessedLWMGet(&lwm);
      }

      if (some terminating_condition)
        break;
   }

   OCIXStreamInDetach();
}
```

OCIXStreamOutAttach()

Purpose

Attaches to an XStream outbound server.

Syntax

```
sword OCIXStreamOutAttach ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            oratext     *server_name,
                            ub2         server_name_len,
                            ub1         *last_position,
                            ub2         last_position_len,
                            ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

server_name (IN)
XStream outbound server name.

server_name_len (IN)
Length of XStream outbound server name.

last_position (IN)
Position to the last received LCR. Can be `NULL`.

last_position_len (IN)
Length of `last_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

The `OCIEnv` environment handle must be created with `OCI_OBJECT` mode, and the service context must be in connected state to issue this function. This function does not support nonblocking mode. It returns either `OCI_SUCCESS` or `OCI_ERROR` status code.

The name of the outbound server must be provided because multiple outbound servers can be configured in one Oracle instance. This function returns `OCI_ERROR` if it encounters any error while attaching to the outbound server. Only one client can attach to an XStream outbound server at any time. An error is returned if multiple clients attempt to attach to the same outbound server or if the same client attempts to attach to multiple outbound servers using the same service handle.

The `last_position` parameter is used to establish the starting point of the stream. This call returns OCI_ERROR if the specified position is non-`NULL` and less than the server's processed low-position (see "OCIXStreamOutProcessedLWMSet()" on page 12-66); otherwise, LCRs with positions greater than the specified `last_position` will be sent to the user.

If the `last_position` is `NULL` then the stream will start from the processed low-position maintained in the server.

OCIXStreamOutDetach()

Purpose

Detaches from the outbound server.

Syntax

```
sword OCIXStreamOutDetach ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            ub4         mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

This function sends the current local processed low-position to the server before detaching from the outbound server. The outbound server automatically restarts after this call. This function returns `OCI_ERROR` if it is invoked while a `OCIXStreamOutReceive()` call is in progress.

OCIXStreamOutLCRReceive()

Purpose

Receives an LCR from an outbound stream. If there is an LCR available, this function immediately returns that LCR. The duration of each LCR is limited to the interval between two successive `OCIXStreamOutLCRReceive()` calls. When there is no LCR available in the stream, this call returns a `NULL` LCR after an idle timeout.

Syntax

```
sword OCIXStreamOutLCRReceive ( OCISvcCtx     *svchp,
                                OCIError      *errhp,
                                void          **lcrp,
                                ub1           *lcrtype,
                                oraub8        *flag,
                                ub1           *fetch_low_position,
                                ub2           *fetch_low_position_len,
                                ub4           mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

lcrp (OUT)
Pointer to the LCR received from the stream. If there is an available LCR, that LCR is returned with status code `OCI_STILL_EXECUTING`. When the call ends, a `NULL` LCR is returned with status code `OCI_SUCCESS`.

lcrtype (OUT)
Type of the retrieved LCR. This value is valid only when `lcrp` is not `NULL`.

flag (OUT)
Return flag. If bit `OCI_XSTREAM_MORE_ROW_DATA` (0x01) is set, it means that this LCR has more data. You must use `OCIXStreamOutReceiveChunk()` function to get the remaining data.

fetch_low_position (OUT)
XStream outbound server's fetch low position. This value is returned only when the return code is `OCI_SUCCESS`. Optional. If non-`NULL`, then you must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes for the return value.

fetch_low_position_len (OUT)
Length of `fetch_low_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

To avoid network round trip for every `OCIXStreamOutLCRReceive()` call, we will tie the connection to this call and let the server fill up the network buffer with LCRs so subsequent calls can quickly receive the LCRs from the network. The server will end each call at the transaction boundary after ACK interval elapses since the call began. When there is no LCR in the stream, the server will end the call after the idle timeout elapses.

Return codes:

- OCI_STILL_EXECUTING means that the current call is still in progress. The connection associated with the specified service context handle is still tied to this call for streaming the LCRs from the server. An error is returned if you attempt to use the same connection to execute any OCI calls that require database round trip, for example, OCIStmtExecute(), OCIStmtFetch(), OCILobRead(), and so on. OCILCR* calls do not require round trips; thus, they are valid while the call is in progress.
- OCI_SUCCESS means that the current call is completed. You are free to execute OCIStmt*, OCILob*, and so on, from the same service context
- OCI_ERROR means the current call encounters some errors. Use OCIErrorGet() to obtain information about the error

This call always returns a NULL LCR when the return code is OCI_SUCCESS. In addition, it returns the fetch low position to denote that the outbound server has received all transactions with commit position lower than or equal to this value.

See Also:

- "Server Handle Attributes" on page 11-4
- "OCIXStreamOutChunkReceive()" on page 12-67 for non-callback pseudocode in the Comments section

OCIXStreamOutLCRCallbackReceive()

Purpose

Used to get the LCR stream from the outbound server using callbacks. You must supply a callback procedure to be invoked for each LCR received. If some LCRs in the stream may contain `LOB` or `LONG` or `XMLType` columns, then a second callback must be supplied to process each chunk (see "OCIXStreamOutChunkReceive()" on page 12-67).

Syntax

```
sword OCIXStreamOutLCRCallbackReceive (
          OCISvcCtx                         *svchp,
          OCIError                          *errhp,
          OCICallbackXStreamOutLCRProcess   processlcr_cb,
          OCICallbackXStreamOutChunkProcess processchunk_cb,
          void                              *usrctxp,
          ub1                               *fetch_low_position,
          ub2                               *fetch_low_position_len,
          ub4                               mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

processlcr_cb (IN)
Callback function to process each LCR received by the client. Cannot be `NULL`.

processchunk_cb (IN)
Callback function to process each chunk in the received LCR. Can be `NULL` if you do not expect to receive any LCRs with additional chunk data.

usrctxp (IN)
User context to pass to both callback procedures.

fetch_low_position (OUT)
XStream outbound server's fetch low position (see "OCIXStreamOutLCRReceive()" on page 12-59). Optional.

fetch_low_position_len (OUT)
Length of `fetch_low_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

Return code: `OCI_SUCCESS` or `OCI_ERROR`.

The `processlcr_cb` argument must be of type `OCICallbackXStreamOutLCRProcess`:

```
typedef sb4  (*OCICallbackXStreamOutLCRProcess)
               (void  *usrctxp, void *lcrp, ub1 lcrtyp, oraub8 flag);
```

Parameters of `OCICallbackXStreamOutLCRProcess()`

usrctxp (IN/OUT)
Pointer to the user context.

lcrp (IN)
Pointer to the LCR just received.

lcrtyp (IN)
LCR type (`OCI_LCR_XROW` or `OCI_LCR_XDDL`).

flag (IN)
If `OCI_XSTREAM_MORE_ROW_DATA` is set, then the current LCR has more chunk data.

The input parameters of the `processlcr_cb()` procedure are the user context, the LCR just received, its type, and a flag to indicate whether the LCR contains more data. If there is an LCR available, this callback is invoked immediately. If there is no LCR in the stream, after an idle timeout, this call will end with `OCI_SUCCESS` return code. The valid return codes from `OCICallbackXStreamOutLCRProcess()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing `OCIXStreamOutLCRCallbackReceive()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate `OCIXStreamOutLCRCallbackReceive()` immediately.

See Also: "Server Handle Attributes" on page 11-4

The `processchunk_cb` argument must be of type `OCICallbackXStreamOutChunkProcess`:

```
typedef sb4  (*OCICallbackXStreamOutChunkProcess)
(void        *usrctxp,
oratext      *column_name,
ub2          column_name_len,
ub2          column_dty,
oraub8       column_flag,
ub2          column_csid,
ub4          chunk_bytes,
ub1          *chunk_data,
oraub8       flag );
```

Parameters of `OCICallbackXStreamOutChunkProcess()`:

usrctxp (IN/OUT)
Pointer to the user context.

column_name (IN)
Column name of the current chunk.

column_name_len (IN)
Length of the column name.

column_name_dty (IN)
Chunk data type (`SQLT_CHR` or `SQLT_BIN`).

column_flag (IN)
See Comments in "OCIXStreamInChunkSend()" on page 12-52.

column_csid (IN)
Column character set id. Relevant only if the column is an `XMLType` column (that is, `column_flag` has `OCI_LCR_COLUMN_XML_DATA` bit set).

chunk_bytes (IN)
Chunk data length in bytes.

chunk_data (IN)
Chunk data pointer.

flag (IN)
If `OCI_XSTREAM_MORE_ROW_DATA` is set, this means the current LCR has more chunk data.

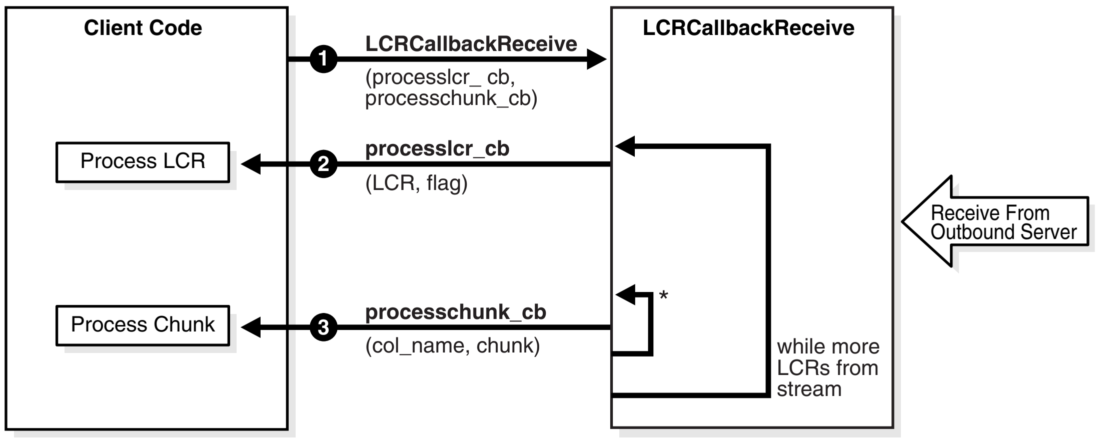
The input parameters of the `processchunk_cb()` procedure are the user context, the information about the chunk, and a flag. When the `flag` argument has `OCI_XSTREAM_MORE_ROW_DATA` (0x01) bit set, then it means there is more data for the current LCR. The valid return codes from `OCICallbackXStreamOutChunkProcess()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing `OCIXStreamOutLCRCallbackReceive()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate `OCIXStreamOutLCRCallbackReceive()` immediately.

OCI calls are provided to access each field in the LCR. If the LCR contains only scalar column(s), the duration of that LCR is limited only to the `processlcr_cb()` procedure. If the LCR contains some chunk data then the duration of the LCR is extended until all the chunks have been processed. If you want to access the LCR data at a later time, a copy of the LCR must be made before it is freed.

As for `OCIXStreamOutLCRReceive()`, the server will end each call at the transaction boundary after each ACK interval since the call begins, or after each idle timeout. The default ACK interval is 30 seconds, and the default idle timeout is one second. See "Server Handle Attributes" on page 11-4 to tune these values. This function also returns the fetch low position when the call ends.

The following figure shows the execution flow of the `OCIXStreamOutLCRCallbackReceive()` function.

Figure 12–2 Execution Flow of the OCIXStreamOutLCRCallbackReceive() Function



* While `OCI_XSTREAM_MORE_ROW_DATA` is set.

Description of the figure:

- At 1, the client invokes `OCIXStreamOutLCRCallbackReceive()` providing two callbacks. This function initiates an LCR outbound stream from the server.
- At 2, this function receives an LCR from the stream and invokes `processlcr_cb()` procedure with the LCR just received. It passes `OCI_XSTREAM_MORE_ROW_DATA` flag to `processlcr_cb()` if the current LCR has additional data.
- If the current LCR has no additional chunk, then this function repeats 2 for the next LCR in the stream.
- At 3, if the current LCR contains additional chunk data, then this function invokes `processchunk_cb()` for each chunk received with `OCI_XSTREAM_MORE_ROW_DATA` flag. This flag is cleared when the callback is invoked on the last chunk of the current LCR.
- If there is more LCR in the stream, then it loops back to 2. This process continues until the end of the current call, or when there is no LCR in the stream for one second, or if a callback function returns any value other than `OCI_CONTINUE`.

Here is sample pseudocode for callback mode:

```
main
{
   /* Attach to outbound server specifying last position */
   OCIXStreamOutAttach(last_pos);

   /* Update the local processed low-position */
   OCIXStreamOutProcessedLWMSet(lwm);

   while (TRUE)
   {
      OCIXStreamOutLCRCallbackReceive(processlcr_cb,
                                      processchunk_cb, fwm);

      /* Use fetch low-position(fwm)
       * to update processed lwm if applied.
       */

      /* Update the local lwm so it will be sent to
       * server during next call.
       */
      OCIXStreamOutProcessedLWMSet(lwm);
      if (some terminating_condition)
         break;
   }
   OCIXStreamOutDetach();
}

processlcr_cb (IN lcr, IN flag)
{
   /* Process the LCR just received */
   OCILCRHeaderGet(lcr);
   OCILCRRowColumnInfoGet(lcr);

   if (lcr is LOB_WRITE | LOB_TRIM | LOB_ERASE)
      OCILCRLobInfoGet(lcr);
   if (OCI_XSTREAM_MORE_ROW_DATA flag set)
      prepare_for_chunk_data;
   else
      process_end_of_row;
}

processchunk_cb (IN chunk, IN flag)
{
   process_chunk;

   if (OCI_XSTREAM_MORE_ROW_DATA flag not set)
      process_end_of_row;
}
```

OCIXStreamOutProcessedLWMSet()

Purpose

Updates the local copy of the processed low-position. It can be called anytime between `OCIXStreamOutAttach()` and `OCIXStreamOutDetach()` calls. Clients, using the callback mechanism to stream LCRs from the server (see "OCIXStreamOutLCRCallbackReceive()" on page 12-61), can invoke this function while in the callback procedures.

Syntax

```
sword OCIXStreamOutProcessedLWMSet ( OCISvcCtx  *svchp,
                                     OCIError   *errhp,
                                     ub1        *processed_low_position,
                                     ub2        processed_low_position_len,
                                     ub4        mode );
```

Parameters

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

processed_low_position (IN)
The processed low position maintained at the client.

processed_low_position_len (IN)
Length of `processed_low_position`.

mode (IN)
Specify `OCI_DEFAULT` for now.

Comments

The processed low-position denotes all LCRs at or below it have been processed. After successfully attaching to an XStream outbound server, a local copy of the processed low-position is maintained at the client. Periodically, this position is sent to the server so that archived logs containing already processed transactions can be purged.

Return code: `OCI_SUCCESS` or `OCI_ERROR`.

Clients using `XStreamOut` functions must keep track of the processed low-position based on what they have processed and call this function whenever their processed low-position has changed, so that a more current value is sent to the server during the next update, which occurs at the beginning of `OCIXStreamOutLCRCallbackReceive()` and `OCIXStreamDetach()` calls. For `OCIXStreamOutLCRReceive()` call, the processed low-position is sent to the server when it initiates a request to start the outbound stream. It is not sent while the stream is in progress.

You can query `DBA_XSTREAM_OUTBOUND_PROGRESS` view to confirm that the processed low-position has been saved in the server.

OCIXStreamOutChunkReceive()

Purpose

Allows the client to retrieve the data of each LOB or LONG or XMLType column one chunk at a time.

Syntax

```
sword OCIXStreamOutChunkReceive ( OCISvcCtx   *svchp,
                                  OCIError    *errhp,
                                  oratext     **column_name,
                                  ub2         *column_name_len,
                                  ub2         *column_dty,
                                  oraub8      *column_flag,
                                  ub2         *column_csid,
                                  ub4         *chunk_bytes,
                                  ub1         **chunk_data,
                                  oraub8      *flag,
                                  ub4         mode );
```

Syntax

svchp (IN)
Service handle context.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

column_name (OUT)
Name of the column that has data.

column_name_len (OUT)
Length of the column name.

column_dty (OUT)
Column chunk data type (either SQLT_CHR or SQLT_BIN).

column_flag (OUT)
Column flag. See Comments for valid flags.

column_csid (OUT)
Column character set id. This is returned only for XMLType column, that is, column_flag has OCI_LCR_COLUMN_XML_DATA bit set.

chunk_bytes (OUT)
Number of bytes in the returned chunk.

chunk_data (OUT)
Pointer to the chunk data in the LCR. The client must not de-allocate this buffer since the LCR and its contents are maintained by this function

flag (OUT)
If OCI_XSTREAM_MORE_ROW_DATA (0x01) is set, it means the current LCR has more chunks coming.

mode (IN)
Specify OCI_DEFAULT for now.

Comments

In Streams, LOB or LONG or XMLType column data is broken up into multiple LCRs based on how they are stored in the redo logs. Thus, for a row change containing these columns multiple LCRs may be constructed. The first LCR of a row change contains the column data for all the scalar columns. All LOB or LONG or XMLType columns in the first LCR are set to NULL since their data are sent in subsequent LCRs for that row change. These column data are stored in the LCR as either RAW (SQLT_BIN) or VARCHAR2 (SQLT_CHR) chunks as shown in the table Table 12–5:

Table 12–5 Storage of LOB or LONG Data in the LCR

Source Column Data Type	Streams LCR Data Type	Streams LCR Character Set
BLOB	RAW	N/A
Fixed-width CLOB	VARCHAR2	Client Character Set
Varying-width CLOB	RAW	AL16UTF16
NCLOB	RAW	AL16UTF16
XMLType	RAW	column csid obtained from the chunk

In Streams, LOB, LONG, or XMLType column, data is broken up into multiple chunks based on how they are stored in the redo logs. For a row change containing columns of these data types, its associated LCR only contains data for the other scalar columns. All LOB, LONG or XMLType columns are either represented in the LCR as NULL or not included in the LCR. The actual data for these columns are sent following each LCR as RAW (SQLT_BIN) or VARCHAR2 (SQLT_CHR) chunks as shown in Table 12–5, " Storage of LOB or LONG Data in the LCR".

LCR column flags can be combined using the bitwise OR operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */
```

Return code: OCI_ERROR or OCI_SUCCESS.

This call returns a NULL column name and NULL chunk data if it is invoked when the current LCR does not contain LOB, LONG, or XMLType columns. This function is valid only when an OCIXStreamOutLCRReceive() call is in progress. An error will be returned if it is called during other times.

If the return flag from OCIXStreamOutLCRReceive() has OCI_XSTREAM_MORE_ROW_DATA bit set, then you must iteratively call OCIXStreamOutChunkReceive() to retrieve all the chunks belonging to that row change before getting the next row change (that is, before making the next OCIXStreamOutLCRReceive() call); otherwise, an error will be returned.

Here is sample pseudocode for non-callback mode:

```
main
{
   /* Attach to outbound server specifying last position */
   OCIXStreamOutAttach(last_pos);

   /* Update the local processed low-position */
   OCIXStreamOutProcessedLWMSet(lwm);

   while (TRUE)
   {
      status = OCIXStreamOutLCRReceive(lcr, flag, fwm);

      if (status == OCI_STILL_EXECUTING)
      {
         /* Process LCR just received */
         OCILCRHeaderGet(lcr);
         OCILCRRowColumnInfoGet(lcr);

         while (OCI_XSTREAM_MORE_ROW_DATA flag set)
         {
            OCIXStreamReceiveChunk(chunk, flag, );

            process_chunk;
         }
         process_end_of_row;
      }
      else if (status == OCI_SUCCESS)
      {
         /* Use fetch low-position(fwm)
          * to update processed lwm if applied.
          */

         /* Update the local lwm so it will be sent to
          * server during next call.
          */
         OCIXStreamOutProcessedLWMSet(lwm);

          if (some terminating_condition)
            break;
       }
   }
   OCIXStreamOutDetach();
}
```

Part V

XStream Data Dictionary Views

This part contains descriptions of the data dictionary views related to XStream. This part contains the following chapters:

13 XStream Static Data Dictionary Views

This chapter describes the static data dictionary views related to XStream.

This chapter contains these topics:

See Also: *Oracle Database Reference*

ALL_APPLY

`ALL_APPLY` displays information about the apply processes that dequeue events from queues accessible to the current user.

Related View

`DBA_APPLY` displays information about all apply processes in the database.

Column	Data Type	NULL	Description
`APPLY_NAME`	`VARCHAR2(30)`	`NOT NULL`	Name of the apply process
`QUEUE_NAME`	`VARCHAR2(30)`	`NOT NULL`	Name of the queue from which the apply process dequeues

Column	Data Type	NULL	Description
QUEUE_OWNER	VARCHAR2(30)	NOT NULL	Owner of the queue from which the apply process dequeues
APPLY_CAPTURED	VARCHAR2(3)		Indicates whether the apply process applies captured events (YES) or user-enqueued events (NO)
RULE_SET_NAME	VARCHAR2(30)		Name of the positive rule set used by the apply process for filtering
RULE_SET_OWNER	VARCHAR2(30)		Owner of the positive rule set used by the apply process for filtering
APPLY_USER	VARCHAR2(30)		User who is applying events
APPLY_DATABASE_LINK	VARCHAR2(128)		Database link to which changes are applied. If null, then changes are applied to the local database.
APPLY_TAG	RAW(2000)		Tag associated with redo log records that are generated when changes are made by the apply process
DDL_HANDLER	VARCHAR2(98)		Name of the user-specified DDL handler, which handles DDL logical change records
PRECOMMIT_HANDLER	VARCHAR2(98)		Name of the user-specified pre-commit handler
MESSAGE_HANDLER	VARCHAR2(98)		Name of the user-specified procedure that handles dequeued events other than logical change records
STATUS	VARCHAR2(8)		Status of the apply process: ■ DISABLED ■ ENABLED ■ ABORTED
MAX_APPLIED_MESSAGE_NUMBER	NUMBER		System change number (SCN) corresponding to the apply process high-watermark for the last time the apply process was stopped using the DBMS_APPLY_ADM.STOP_APPLY procedure with the force parameter set to false. The apply process high-watermark is the SCN beyond which no events have been applied.
NEGATIVE_RULE_SET_NAME	VARCHAR2(30)		Name of the negative rule set used by the apply process for filtering
NEGATIVE_RULE_SET_OWNER	VARCHAR2(30)		Owner of the negative rule set used by the apply process for filtering
STATUS_CHANGE_TIME	DATE		Time that the STATUS of the apply process was changed
ERROR_NUMBER	NUMBER		Error number if the apply process was aborted
ERROR_MESSAGE	VARCHAR2(4000)		Error message if the apply process was aborted
MESSAGE_DELIVERY_MODE	VARCHAR2(10)		Reserved for internal use
PURPOSE	VARCHAR2(19)		Purpose of the apply process: ■ STREAMS APPLY for an apply process ■ XSTREAM OUT for an XStream outbound server ■ XSTREAM IN for an XStream inbound server

See Also: "DBA_APPLY" on page 13-6

ALL_APPLY_ERROR

ALL_APPLY_ERROR displays information about the error transactions generated by the apply processes that dequeue events from queues accessible to the current user.

Related View

DBA_APPLY_ERROR displays information about the error transactions generated by all apply processes in the database.

Column	Data Type	NULL	Description
APPLY_NAME	VARCHAR2(30)		Name of the apply process at the local database which processed the transaction
QUEUE_NAME	VARCHAR2(30)		Name of the queue at the local database from which the transaction was dequeued
QUEUE_OWNER	VARCHAR2(30)		Owner of the queue at the local database from which the transaction was dequeued
LOCAL_TRANSACTION_ID	VARCHAR2(22)		Local transaction ID for the error transaction
SOURCE_DATABASE	VARCHAR2(128)		Database where the transaction originated
SOURCE_TRANSACTION_ID	VARCHAR2(128)		Original transaction ID at the source database
SOURCE_COMMIT_SCN	NUMBER		Original commit system change number (SCN) for the transaction at the source database
MESSAGE_NUMBER	NUMBER		Identifier for the event in the transaction that raised an error
ERROR_NUMBER	NUMBER		Error number of the error raised by the transaction
ERROR_MESSAGE	VARCHAR2(4000)		Error message of the error raised by the transaction
RECIPIENT_ID	NUMBER		User ID of the original user that applied the transaction
RECIPIENT_NAME	VARCHAR2(30)		Name of the original user that applied the transaction
MESSAGE_COUNT	NUMBER		Total number of events inside the error transaction
ERROR_CREATION_TIME	DATE		Time that the error was created
SOURCE_COMMIT_POSITION	RAW(64)		Original commit position for the transaction

See Also: "DBA_APPLY_ERROR" on page 13-6

ALL_XSTREAM_INBOUND

ALL_XSTREAM_INBOUND displays information about the XStream inbound servers accessible to the current user.

Related View

DBA_XSTREAM_INBOUND displays information about all XStream inbound servers in the database.

Column	Data Type	NULL	Description
SERVER_NAME	VARCHAR2(30)	NOT NULL	Name of the inbound server
QUEUE_OWNER	VARCHAR2(30)	NOT NULL	Owner of the queue associated with the inbound server
QUEUE_NAME	VARCHAR2(30)	NOT NULL	Name of the queue associated with the inbound server
APPLY_USER	VARCHAR2(30)		Name of the user who is applying the messages
USER_COMMENT	VARCHAR2(4000)		User comment
CREATE_DATE	TIMESTAMP(6)		Date when the inbound server was created

See Also: "DBA_XSTREAM_INBOUND" on page 13-7

ALL_XSTREAM_INBOUND_PROGRESS

ALL_XSTREAM_INBOUND_PROGRESS displays information about the progress made by the XStream inbound servers accessible to the current user.

Related View

DBA_XSTREAM_INBOUND_PROGRESS displays information about the progress made by all XStream inbound servers in the database.

Column	Data Type	NULL	Description
SERVER_NAME	VARCHAR2(30)	NOT NULL	Name of the inbound server
PROCESSED_LOW_POSITION	RAW(64)		Position of the processed low transaction
APPLIED_LOW_POSITION	RAW(64)		All messages with a commit position less than this value have been applied
APPLIED_HIGH_POSITION	RAW(64)		Highest commit position of a transaction that has been applied
SPILL_POSITION	RAW(64)		Position of the spill low-watermark

See Also: "DBA_XSTREAM_INBOUND_PROGRESS" on page 13-7

ALL_XSTREAM_OUTBOUND

ALL_XSTREAM_OUTBOUND displays information about the XStream outbound servers accessible to the current user.

Related View

DBA_XSTREAM_OUTBOUND displays information about all XStream outbound servers in the database.

Column	Data Type	NULL	Description
SERVER_NAME	VARCHAR2(30)	NOT NULL	Name of the outbound server
CONNECT_USER	VARCHAR2(30)		Name of the user who can process the outbound LCRs
CAPTURE_NAME	VARCHAR2(30)		Name of the Streams capture process
SOURCE_DATABASE	VARCHAR2(128)		Database where the transaction originated
CAPTURE_USER	VARCHAR2(30)		Current user who is enqueuing captured messages
QUEUE_OWNER	VARCHAR2(30)	NOT NULL	Owner of the queue associated with the outbound server
QUEUE_NAME	VARCHAR2(30)	NOT NULL	Name of the queue associated with the outbound server
USER_COMMENT	VARCHAR2(4000)		User comment
CREATE_DATE	TIMESTAMP(6)		Date when the outbound server was created

See Also: "DBA_XSTREAM_OUTBOUND" on page 13-7

ALL_XSTREAM_OUTBOUND_PROGRESS

ALL_XSTREAM_OUTBOUND_PROGRESS displays information about the progress made by the XStream outbound servers accessible to the current user.

Related View

DBA_XSTREAM_OUTBOUND_PROGRESS displays information about the progress made by all XStream outbound servers in the database.

Column	Data Type	NULL	Description
SERVER_NAME	VARCHAR2(30)	NOT NULL	Name of the outbound server
SOURCE_DATABASE	VARCHAR2(128)		Database where the transaction originated
PROCESSED_LOW_POSITION	RAW(64)		Position of the low-watermark transaction processed by the client
PROCESSED_LOW_TIME	DATE		Time when the processed low position was last updated by the outbound server

See Also: "DBA_XSTREAM_OUTBOUND_PROGRESS" on page 13-7

ALL_XSTREAM_RULES

ALL_XSTREAM_RULES displays information about the XStream server rules accessible to the current user.

Related View

DBA_XSTREAM_RULES displays information about all XStream server rules in the database.

Column	Data Type	NULL	Description
STREAMS_NAME	VARCHAR2(30)		Name of the Streams process
STREAMS_TYPE	VARCHAR2(12)		Type of the Streams process: ■ CAPTURE ■ APPLY
STREAMS_RULE_TYPE	VARCHAR2(6)		For global, schema, or table rules, the type of the rule: ■ TABLE ■ SCHEMA ■ GLOBAL
RULE_SET_OWNER	VARCHAR2(30)		Owner of the rule set
RULE_SET_NAME	VARCHAR2(30)		Name of the rule set
RULE_SET_TYPE	CHAR(8)		Type of the rule set: ■ POSITIVE ■ NEGATIVE
RULE_OWNER	VARCHAR2(30)	NOT NULL	Owner of the rule
RULE_NAME	VARCHAR2(30)	NOT NULL	Name of the rule
RULE_TYPE	VARCHAR2(3)		For global, schema, or table rules, the type of the rule: ■ DML ■ DDL
RULE_CONDITION	CLOB		Current rule condition
SCHEMA_NAME	VARCHAR2(30)		For table and schema rules, the schema name
OBJECT_NAME	VARCHAR2(30)		For table rules, the table name
INCLUDE_TAGGED_LCR	VARCHAR2(3)		For global, schema, or table rules, indicates whether to include tagged logical change records (LCRs) (YES) or not (NO)

Column	Data Type	NULL	Description
SUBSETTING_OPERATION	VARCHAR2(6)		For subset rules, the type of operation: ■ INSERT ■ UPDATE ■ DELETE
DML_CONDITION	VARCHAR2(4000)		For subset rules, the row subsetting condition
SOURCE_DATABASE	VARCHAR2(128)		For global, schema, or table rules, the name of the database where the LCRs originated
ORIGINAL_RULE_CONDITION	VARCHAR2(4000)		For rules created by Streams administrative APIs, the original rule condition when the rule was created
SAME_RULE_CONDITION	VARCHAR2(3)		For rules created by Streams administrative APIs, indicates whether the current rule condition is the same as the original rule condition (YES) or not (NO)

See Also: "DBA_XSTREAM_RULES" on page 13-7

DBA_APPLY

DBA_APPLY displays information about all apply processes in the database. Its columns are the same as those in ALL_APPLY.

See Also: "ALL_APPLY" on page 13-1

DBA_APPLY_ERROR

DBA_APPLY_ERROR displays information about the error transactions generated by all apply processes in the database. Its columns are the same as those in ALL_APPLY_ERROR.

See Also: "ALL_APPLY_ERROR" on page 13-2

DBA_APPLY_SPILL_TXN

DBA_APPLY_SPILL_TXN displays information about the transactions spilled from memory to hard disk by all apply processes in the database.

Column	Data Type	NULL	Description
APPLY_NAME	VARCHAR2(30)	NOT NULL	Name of the apply process that spilled one or more transactions
XIDUSN	NUMBER	NOT NULL	Transaction ID undo segment number
XIDSLT	NUMBER	NOT NULL	Transaction ID slot number
XIDSQN	NUMBER	NOT NULL	Transaction ID sequence number
FIRST_SCN	NUMBER	NOT NULL	SCN of the first message in the transaction
MESSAGE_COUNT	NUMBER		Number of messages spilled for the transaction
FIRST_MESSAGE_CREATE_TIME	DATE		Source creation time of the first message in the transaction
SPILL_CREATION_TIME	DATE		Time the first message was spilled
FIRST_POSITION	RAW(64)		Position of the first message in this transaction
TRANSACTION_ID	VARCHAR2(128)		Transaction ID of the spilled transaction

DBA_XSTREAM_INBOUND

`DBA_XSTREAM_INBOUND` displays information about all XStream inbound servers in the database. Its columns are the same as those in `ALL_XSTREAM_INBOUND`.

See Also: "ALL_XSTREAM_INBOUND" on page 13-3

DBA_XSTREAM_INBOUND_PROGRESS

`DBA_XSTREAM_INBOUND_PROGRESS` displays information about the progress made by all XStream inbound servers in the database. Its columns are the same as those in `ALL_XSTREAM_INBOUND_PROGRESS`.

See Also: "ALL_XSTREAM_INBOUND_PROGRESS" on page 13-4

DBA_XSTREAM_OUTBOUND

`DBA_XSTREAM_OUTBOUND` displays information about all XStream outbound servers in the database. Its columns are the same as those in `ALL_XSTREAM_OUTBOUND`.

See Also: "ALL_XSTREAM_OUTBOUND" on page 13-4

DBA_XSTREAM_OUTBOUND_PROGRESS

`DBA_XSTREAM_OUTBOUND_PROGRESS` displays information about the progress made by all XStream outbound servers in the database. Its columns are the same as those in `ALL_XSTREAM_OUTBOUND_PROGRESS`.

See Also: "ALL_XSTREAM_OUTBOUND_PROGRESS" on page 13-7

DBA_XSTREAM_RULES

`DBA_XSTREAM_RULES` displays information about all XStream server rules in the database. Its columns are the same as those in `ALL_XSTREAM_RULES`.

See Also: "ALL_XSTREAM_RULES" on page 13-5

14

XStream Dynamic Performance (V$) Views

This chapter describes the dynamic performance (V$) views related to XStream.

This chapter contains these topics:

- V$STREAMS_APPLY_READER
- V$STREAMS_APPLY_SERVER
- V$STREAMS_CAPTURE
- V$STREAMS_MESSAGE_TRACKING
- V$STREAMS_TRANSACTION

See Also: *Oracle Database Reference*

V$STREAMS_APPLY_READER

`V$STREAMS_APPLY_READER` displays information about each apply reader. The apply reader for an apply process is a process which reads (dequeues) messages from the queue, computes message dependencies, builds transactions, and passes the transactions on to the apply process coordinator in commit order for assignment to the apply servers.

Column	Data Type	Description
`SID`	`NUMBER`	Session ID of the reader's session
`SERIAL#`	`NUMBER`	Serial number of the reader's session
`APPLY#`	`NUMBER`	Apply process number. An apply process is an Oracle background process, prefixed by `ap`.
`APPLY_NAME`	`VARCHAR2(30)`	Name of the apply process that spilled one or more transactions
`STATE`	`VARCHAR2(36)`	State of the reader: ■ `INITIALIZING` - Starting up ■ `IDLE` - Performing no work ■ `DEQUEUE MESSAGES` - Dequeuing messages from the apply process queue ■ `SCHEDULE MESSAGES` - Computing dependencies between messages and assembling messages into transactions ■ `SPILLING` - Spilling unapplied messages from memory to hard disk ■ `PAUSED - WAITING FOR DDL TO COMPLETE` - Waiting for a DDL LCR to be applied
`TOTAL_MESSAGES_DEQUEUED`	`NUMBER`	Total number of messages dequeued since the apply process was last started

Column	Data Type	Description
TOTAL_MESSAGES_ SPILLED	NUMBER	Number of messages spilled by the reader since the apply process was last started
DEQUEUE_TIME	DATE	Time when the last message was received
DEQUEUED_MESSAGE_ NUMBER	NUMBER	Number of the last message received
DEQUEUED_MESSAGE_ CREATE_TIME	DATE	For captured messages, creation time at the source database of the last message received. For user-enqueued messages, time when the message was enqueued into the queue at the local database.
SGA_USED	NUMBER	Amount (in bytes) of SGA memory used by the apply process since it was last started
ELAPSED_DEQUEUE_TIME	NUMBER	Time elapsed (in hundredths of a second) dequeuing messages since the apply process was last started
ELAPSED_SCHEDULE_TIME	NUMBER	Time elapsed (in hundredths of a second) scheduling messages since the apply process was last started. Scheduling includes computing dependencies between messages and assembling messages into transactions.
ELAPSED_SPILL_TIME	NUMBER	Elapsed time spent spilling messages (in hundredths of a second) since the apply process was last started
LAST_BROWSE_NUM	NUMBER	Last browse SCN
OLDEST_SCN_NUM	NUMBER	Oldest SCN
LAST_BROWSE_SEQ	NUMBER	Last browse sequence number
LAST_DEQ_SEQ	NUMBER	Last dequeue sequence number
OLDEST_XIDUSN	NUMBER	Transaction ID undo segment number of the oldest transaction to be applied/being applied
OLDEST_XIDSLT	NUMBER	Transaction ID slot number of the oldest transaction to be applied/being applied
OLDEST_XIDSQN	NUMBER	Transaction ID sequence number of the oldest transaction to be applied/ being applied
SPILL_LWM_SCN	NUMBER	Spill low-watermark SCN
PROXY_SID	NUMBER	When the apply process uses combined capture and apply, the session ID of the propagation receiver that is responsible for direct communication between capture and apply. If the apply process does not use combined capture and apply, then this column is 0.
PROXY_SERIAL	NUMBER	When the apply process uses combined capture and apply, the serial number of the propagation receiver that is responsible for direct communication between capture and apply. If the apply process does not use combined capture and apply, then this column is 0.
PROXY_SPID	VARCHAR2(12)	When the apply process uses combined capture and apply, the process identification number of the propagation receiver that is responsible for direct communication between capture and apply. If the apply process does not use combined capture and apply, then this column is 0.
CAPTURE_BYTES_ RECEIVED	NUMBER	When the apply process uses combined capture and apply, the number of bytes received by the apply process from the capture process since the apply process last started. If the apply process does not use combined capture and apply, then this column is not populated.
DEQUEUED_POSITION	RAW(64)	Dequeued position (for an apply process that is functioning as an XStream inbound server)
LAST_BROWSE_POSITION	RAW(64)	Last browse position (for an apply process that is functioning as an XStream inbound server)
OLDEST_POSITION	RAW(64)	Oldest position (for an apply process that is functioning as an XStream inbound server)
SPILL_LWM_POSITION	RAW(64)	Spill low watermark position (for an apply process that is functioning as an XStream inbound server)
OLDEST_TRANSACTION_ID	VARCHAR2(128)	Oldest transaction ID (for an apply process that is functioning as an XStream inbound server)

Note: The ELAPSED_DEQUEUE_TIME and ELAPSED_SCHEDULE_TIME columns are only populated if the TIMED_STATISTICS initialization parameter is set to true, or if the STATISTICS_LEVEL initialization parameter is set to TYPICAL or ALL.

V$STREAMS_APPLY_SERVER

V$STREAMS_APPLY_SERVER displays information about each apply server and its activities. An apply server receives events from the apply coordinator for an apply process. For each event received, an apply server either applies the event or sends the event to the appropriate apply handler.

Column	Data Type	Description
SID	NUMBER	Session ID of the apply server's session
SERIAL#	NUMBER	Serial number of the apply server's session
APPLY#	NUMBER	Apply process number. An apply process is an Oracle background process, prefixed by ap.
APPLY_NAME	VARCHAR2(30)	Name of the apply process
SERVER_ID	NUMBER	Parallel execution server number of the apply server
STATE	VARCHAR2(20)	State of the apply server: ▪ IDLE - Performing no work ▪ RECORD LOW-WATERMARK - Performing an administrative job that maintains information about the apply progress, which is used in the ALL_APPLY_PROGRESS and DBA_APPLY_PROGRESS data dictionary views ▪ ADD PARTITION - Performing an administrative job that adds a partition that is used for recording information about in-progress transactions ▪ DROP PARTITION - Performing an administrative job that purges rows that were used to record information about in-progress transactions ▪ EXECUTE TRANSACTION - Applying a transaction ▪ WAIT COMMIT - Waiting to commit a transaction until all other transactions with a lower commit SCN are applied. This state is possible only if the COMMIT_SERIALIZATION apply process parameter is set to a value other than DEPENDENT_TRANSACTIONS and the PARALLELISM apply process parameter is set to a value greater than 1. ▪ WAIT DEPENDENCY - Waiting to apply a logical change record (LCR) in a transaction until another transaction, on which it has a dependency, is applied. This state is possible only if the PARALLELISM apply process parameter is set to a value greater than 1. ▪ ROLLBACK TRANSACTION - Rolling back a transaction ▪ TRANSACTION CLEANUP - Cleaning up an applied transaction, which includes removing LCRs from the apply process's queue ▪ INITIALIZING - Starting up ▪ WAIT FOR CLIENT - Waiting for an XStream In client application to request more logical change records (LCRs)
XIDUSN	NUMBER	Transaction ID undo segment number of the transaction currently being applied
XIDSLT	NUMBER	Transaction ID slot number of the transaction currently being applied
XIDSQN	NUMBER	Transaction ID sequence number of the transaction currently being applied

Column	Data Type	Description
COMMITSCN	NUMBER	Commit system change number (SCN) of the transaction currently being applied
DEP_XIDUSN	NUMBER	Transaction ID undo segment number of a transaction on which the transaction being applied by this apply server depends
DEP_XIDSLT	NUMBER	Transaction ID slot number of a transaction on which the transaction being applied by this apply server depends
DEP_XIDSQN	NUMBER	Transaction ID sequence number of a transaction on which the transaction being applied by this apply server depends
DEP_COMMITSCN	NUMBER	Commit system change number (SCN) of the transaction on which this apply server depends
MESSAGE_SEQUENCE	NUMBER	Number of the current message being applied by the apply server. This value is reset to `1` at the beginning of each transaction.
TOTAL_ASSIGNED	NUMBER	Total number of transactions assigned to the apply server since the apply process was last started
TOTAL_ADMIN	NUMBER	Total number of administrative jobs done by the apply server since the apply process was last started. See the `STATE` information in this view for the types of administrative jobs.
TOTAL_ROLLBACKS	NUMBER	Number of transactions assigned to this server which were rolled back
TOTAL_MESSAGES_ APPLIED	NUMBER	Total number of messages applied by this apply server since the apply process was last started
APPLY_TIME	DATE	Time the last message was applied
APPLIED_MESSAGE_ NUMBER	NUMBER	Number of the last message applied
APPLIED_MESSAGE_ CREATE_TIME	DATE	Creation time at the source database of the last captured message applied. No information about user-enqueued messages is recorded in this column.
ELAPSED_DEQUEUE_TIME	NUMBER	Time elapsed (in hundredths of a second) dequeuing messages since the apply process was last started
ELAPSED_APPLY_TIME	NUMBER	Time elapsed (in hundredths of a second) applying messages since the apply process was last started
COMMIT_POSITION	RAW(64)	Commit position of the transaction (for an apply process that is functioning as an XStream inbound server)
DEP_COMMIT_POSITION	RAW(64)	Commit position of the transaction the slave depends on (for an apply process that is functioning as an XStream inbound server)
LAST_APPLY_POSITION	RAW(64)	Position of the last message applied (for an apply process that is functioning as an XStream inbound server)
TRANSACTION_ID	VARCHAR2(128)	Transaction ID that the slave is applying (for an apply process that is functioning as an XStream inbound server)
DEP_TRANSACTION_ID	VARCHAR2(128)	Transaction ID of the transaction the slave depends on (for an apply process that is functioning as an XStream inbound server)

Note: The `ELAPSED_DEQUEUE_TIME` and `ELAPSED_APPLY_TIME` columns are only populated if the `TIMED_STATISTICS` initialization parameter is set to `true`, or if the `STATISTICS_LEVEL` initialization parameter is set to `TYPICAL` or `ALL`.

V$STREAMS_CAPTURE

`V$STREAMS_CAPTURE` displays information about each capture process.

Column	Data Type	Description
SID	NUMBER	Session identifier of the capture process
SERIAL#	NUMBER	Session serial number of the capture process session
CAPTURE#	NUMBER	Capture process number. A capture process is an Oracle background process, prefixed by cp.
CAPTURE_NAME	VARCHAR2(30)	Name of the capture process
LOGMINER_ID	NUMBER	Session ID of the LogMiner session associated with the capture process
STARTUP_TIME	DATE	Time when the capture process was last started
STATE	VARCHAR2(551)	State of the capture process: ■ INITIALIZING ■ CAPTURING CHANGES ■ EVALUATING RULE ■ ENQUEUING MESSAGE ■ SHUTTING DOWN ■ ABORTING ■ CREATING LCR ■ WAITING FOR DICTIONARY REDO ■ WAITING FOR REDO ■ PAUSED FOR FLOW CONTROL ■ DICTIONARY INITIALIZATION ■ WAITING FOR A SUBSCRIBER TO BE ADDED ■ WAITING FOR BUFFERED QUEUE TO SHRINK ■ SUSPENDED FOR AUTO SPLIT/MERGE
TOTAL_PREFILTER_DISCARDED	NUMBER	Total number of prefiltered messages discarded
TOTAL_PREFILTER_KEPT	NUMBER	Total number of prefiltered messages kept
TOTAL_PREFILTER_EVALUATIONS	NUMBER	Total number of prefilter evaluations
TOTAL_MESSAGES_CAPTURED	NUMBER	Total number of redo entries passed by LogMiner to the capture process for detailed rule evaluation since the capture process last started. A capture process converts a redo entry into a message and performs detailed rule evaluation on the message when capture process prefiltering cannot discard the change.
CAPTURE_TIME	DATE	Time when the most recent message was captured
CAPTURE_MESSAGE_NUMBER	NUMBER	Number of the most recently captured message
CAPTURE_MESSAGE_CREATE_TIME	DATE	Creation time of the most recently captured message
TOTAL_MESSAGES_CREATED	NUMBER	Count associated with ELAPSED_LCR_TIME to calculate rate
TOTAL_FULL_EVALUATIONS	NUMBER	Count associated with ELAPSED_RULE_TIME to calculate rate
TOTAL_MESSAGES_ENQUEUED	NUMBER	Total number of messages enqueued since the capture process was last started
ENQUEUE_TIME	DATE	Time when the last message was enqueued
ENQUEUE_MESSAGE_NUMBER	NUMBER	Number of the last enqueued message
ENQUEUE_MESSAGE_CREATE_TIME	DATE	Creation time of the last enqueued message
AVAILABLE_MESSAGE_NUMBER	NUMBER	For local capture, the last redo SCN flushed to the log files. For downstream capture, the last SCN added to LogMiner through the archive logs.

Column	Data Type	Description
AVAILABLE_MESSAGE_CREATE_TIME	DATE	For local capture, the time the SCN was written to the log file. For downstream capture, the time the most recent archive log (containing the most recent SCN) was added to LogMiner.
ELAPSED_CAPTURE_TIME	NUMBER	Elapsed time (in hundredths of a second) scanning for changes in the redo log since the capture process was last started
ELAPSED_RULE_TIME	NUMBER	Elapsed time (in hundredths of a second) evaluating rules since the capture process was last started
ELAPSED_ENQUEUE_TIME	NUMBER	Elapsed time (in hundredths of a second) enqueuing messages since the capture process was last started
ELAPSED_LCR_TIME	NUMBER	Elapsed time (in hundredths of a second) creating logical change records (LCRs) since the capture process was last started
ELAPSED_REDO_WAIT_TIME	NUMBER	Elapsed time (in hundredths of a second) spent by the capture process in the WAITING FOR REDO state
ELAPSED_PAUSE_TIME	NUMBER	Elapsed flow control pause time (in hundredths of a second)
STATE_CHANGED_TIME	DATE	Time at which the state of the capture process changed
APPLY_NAME	VARCHAR2(30)	Reserved for internal use
APPLY_DBLINK	VARCHAR2(128)	Reserved for internal use
APPLY_MESSAGES_SENT	NUMBER	Reserved for internal use
APPLY_BYTES_SENT	NUMBER	Reserved for internal use
OPTIMIZATION	NUMBER	Indicates whether a capture process uses combined capture and apply (greater than zero) or a capture process does not use combined capture and apply (0)

Note: The ELAPSED_CAPTURE_TIME, ELAPSED_RULE_TIME, ELAPSED_ENQUEUE_TIME, ELAPSED_LCR_TIME, and ELAPSED_REDO_WAIT_TIME columns are only populated if the TIMED_STATISTICS initialization parameter is set to true, or if the STATISTICS_LEVEL initialization parameter is set to TYPICAL or ALL.

V$STREAMS_MESSAGE_TRACKING

V$STREAMS_MESSAGE_TRACKING displays information about LCRs tracked through the stream that are processed by each Streams client. Use the DBMS_STREAMS_ADM.SET_MESSAGE_TRACKING procedure to specify a tracking label that becomes part of each LCR generated by the current session.

If the actions parameter in the DBMS_STREAMS_ADM.SET_MESSAGE_TRACKING procedure is set to DBMS_STREAMS_ADM.ACTION_MEMORY, then information about the LCRs is tracked in memory and this view is populated with information about the LCRs. If the actions parameter is set to DBMS_STREAMS_ADM.ACTION_TRACE, then this view is not populated and information about the LCRs is sent to the trace files at each database.

Column	Data Type	Description
TRACKING_LABEL	VARCHAR2(30)	User-specified tracking label
TAG	RAW(30)	First 30 bytes of the tag of the LCR
COMPONENT_NAME	VARCHAR2(30)	Name of the component that processed the LCR
COMPONENT_TYPE	VARCHAR2(30)	Type of the component that processed the LCR
ACTION	VARCHAR2(50)	Action performed on the LCR

Column	Data Type	Description
ACTION_DETAILS	VARCHAR2(100)	Details of the action
TIMESTAMP	TIMESTAMP(9) WITH TIME ZONE	Time when the action was performed
MESSAGE_CREATION_TIME	DATE	Time when the message was created
MESSAGE_NUMBER	NUMBER	SCN of the message
TRACKING_ID	RAW(16)	Globally unique OID of the LCR
SOURCE_DATABASE_NAME	VARCHAR2(128)	Name of the source database
OBJECT_OWNER	VARCHAR2(30)	Owner of the object
OBJECT_NAME	VARCHAR2(30)	Name of the object
XID	VARCHAR2(128)	Transaction ID
COMMAND_TYPE	VARCHAR2(30)	Command type of the LCR
MESSAGE_POSITION	RAW(64)	Position of the message (for an apply process that is functioning as an XStream inbound server)

V$STREAMS_TRANSACTION

V$STREAMS_TRANSACTION displays information about transactions that are being processed by capture processes or apply processes. This view can be used to identify long running transactions and to determine how many logical change records (LCRs) are being processed in each transaction. This view only contains information about captured LCRs. It does not contain information about user-enqueued LCRs or user messages.

This view only shows information about LCRs that are being processed because they satisfied the rule sets for the Streams process at the time of the query. For capture processes, this view only shows information about changes in transactions that the capture process has converted to LCRs. It does not show information about all the active transactions present in the redo log. For apply processes, this view only shows information about LCRs that the apply process has dequeued. It does not show information about LCRs in the apply process's queue.

Information about a transaction remains in the view until the transaction commits or until the entire transaction is rolled back.

Column	Data Type	Description
STREAMS_NAME	VARCHAR2(30)	Name of the Streams process
STREAMS_TYPE	VARCHAR2(10)	Type of the Streams process: ■ CAPTURE ■ APPLY ■ PROPAGATION_SENDER
XIDUSN	NUMBER	Transaction ID undo segment number of the transaction
XIDSLT	NUMBER	Transaction ID slot number of the transaction
XIDSQN	NUMBER	Transaction ID sequence number of the transaction
CUMULATIVE_MESSAGE_COUNT	NUMBER	Number of LCRs processed in the transaction. If the Streams process is restarted while the transaction is being processed, then this column shows the number of LCRs processed in the transaction since the Streams process was started.
TOTAL_MESSAGE_COUNT	NUMBER	Total Number of LCRs processed in the transaction by an apply process. This column does not pertain to capture processes.

Column	Data Type	Description
FIRST_MESSAGE_TIME	DATE	Timestamp of the first LCR processed in the transaction. If a capture process is restarted while the transaction is being processed, then this column shows the timestamp of the first LCR processed after the capture process was started.
FIRST_MESSAGE_NUMBER	NUMBER	System change number (SCN) of the first message in the transaction. If a capture process is restarted while the transaction is being processed, then this column shows the SCN of the first message processed after the capture process was started.
LAST_MESSAGE_TIME	DATE	Timestamp of the last LCR processed in the transaction
LAST_MESSAGE_NUMBER	NUMBER	SCN of the most recent message encountered for the transaction
FIRST_MESSAGE_POSITION	RAW(64)	Position of the first message seen (for an apply process that is functioning as an XStream inbound server)
LAST_MESSAGE_POSITION	RAW(64)	Position of the last message seen (for an apply process that is functioning as an XStream inbound server)
TRANSACTION_ID	VARCHAR2(128)	Transaction ID (for an apply process that is functioning as an XStream inbound server)

Index