Oracle7™ Server Distributed Systems, Volume II: Replicated Data

Release 7.3

February 1996

Part No. A32545–2

ORACLE®

Oracle7™ Server Distributed Systems, Volume II: Replicated Data, Release 7.3

Part No. A32545–2

Primary Author: Maria Pratt
Contributing Authors: Jason Durbin, Thomas Albert
Contributors: Sukanaya Balaraman, Dean Daniels, Dominic Delmolino, Lip Doo, Alan Downing, Ashish Gupta, Gary Hallmark, Sandeep Jain, Sue Jang, Bob Jenkins, Larry Mix, Valarie Moore, Huyen Nguyen, Brian Oki, Javier Seen, Gordon Smith, Benny Souder, Jim Stamos, Harry Sun, Wendy Unwin

Preface

This manual describes Oracle7 Server replication capabilities. To use the synchronous replication facility, you must have installed Oracle's advanced replication option. Basic replication (read–only and updatable snapshots) is standard Oracle distributed functionality. Procedural replication requires PL/SQL and the advanced replication option.

Information in this manual applies to the Oracle7 Server running on all operating systems.

Topics include the following:

- read–only snapshots
- symmetric replication facility
  - updatable snapshots
  - multi–master replication
  - snapshot site replication
  - conflict resolution
- administering a replicated environment
- advanced techniques
- troubleshooting
- using job queues
- deferred transactions

Audience

This manual is written for application developers and database administrators who develop and maintain advanced Oracle7 distributed systems.

Knowledge Assumed of the Reader

This manual assumes you are familiar with relational database concepts, distributed database administration, PL/SQL (if using procedural replication), and the operating system under which you run will run an Oracle replicated environment.

This manual also assumes that you have read and understand the information in the following documents:

- *Oracle7 Server Concepts,*
- The *Oracle7 Server Administrator's Guide,*
- *Oracle7 Server Distributed Systems, Volume I,*
- and, if you plan to use procedural replication, the *PL/SQL User's Guide and Reference.*

How *Oracle7 Server Distributed Systems, Volume II* Is Organized

This manual contains 13 chapters and two appendices, as described below.

Chapter 1: Understanding Replication
This chapter provides an overview of the options available for data replication with the Oracle Server.

Chapter 2: Designing a Replicated Environment
This chapter introduces many of the issues that you will need to consider when designing your replicated environment.

Chapter 3: Read–Only Snapshots
This chapter explains how to use read–only snapshots to perform basic primary site replication.

Chapter 4: Multi–Master Replication
This chapter describes how to create and maintain a multi–master replicated environment using the Oracle symmetric replication facility.

Chapter 5: Snapshot Replication
This chapter describes how to create and maintain a snapshot site using the Oracle symmetric replication facility. This chapter also describes updatable snapshots.

Chapter 6: Conflict Resolution
This chapter describes how to use Oracle–supplied conflict resolution methods to resolve conflicts resulting from dynamic or shared ownership of data in a replicated environment.

Chapter 7: Administering a Replicated Environment
This chapter describes how to detect and resolve unresolved replication errors, as well as how to monitor successful conflict resolution.

Chapter 8: Advanced Techniques
This chapter describes advanced replication techniques, including how to do the following: create your own conflict resolution routines; implement a fail–over site; implement token passing; use procedural replication; and handle deletes.

Chapter 9: Troubleshooting
This chapter describes how to solve several common problems that you may encounter in setting up your replicated environment.

Chapter 10: Using Job Queues
This chapter describes how to use job queues to schedule routines to be executed periodically.

Chapter 11: Using Deferred Transactions
This chapter describes how to build transactions for deferred execution at remote locations.

Chapter 12: Reference
This chapter describes the parameters for the packaged procedures used to implement a replicated environment, as well as any exceptions that might be raised by these procedures.

Chapter 13: Data Dictionary Views
This chapter describes the views that may be of interest to users of deferred transactions, read–only snapshots, and the symmetric replication facility.

Appendix A: Compatibility
This appendix discusses the compatibility issues between release 7.3 of the advanced replication option and previous releases.

Appendix B: Operating System–Specific Information
This appendix is a summary of all the operating system–specific references contained within this manual.

Conventions Used in This Manual

This manual uses different fonts to represent different types of information.

Special Icons

Special icons alert you to particular information within the body of this manual:

Suggestion: The lightbulb highlights suggestions and practical tips that could save time, make procedures easier, and so on.

Attention: This icon draws your attention to items of particular importance.

Warning: The warning symbol highlights text that warns you of actions that could be particularly damaging or fatal to your operations.

OSDoc

Additional Information: The OSDoc icon refers you to the Oracle operating system–specific documentation for additional information.

Text of the Manual

The following sections describe the conventions used in the text of this manual.

UPPERCASE Characters

Uppercase text is used to call attention to command keywords, object names, parameters, filenames, and so on.

For example, "If you create a private rollback segment, the name must be included in the ROLLBACK_SEGMENTS parameter of the parameter file."

Italicized Characters

Italicized words within text indicate the definition of a word, book titles, or emphasized words.

An example of a definition is the following: "A *database* is a collection of data to be treated as a unit. The general purpose of a database is to store and retrieve related information, as needed."

An example of a reference to another book is the following: "For more information, see *Oracle7 Server Tuning.*"

An example of an emphasized word is the following: "You *must* back up your database regularly."

Code Examples

SQL, Server Manager line mode, and SQL*Plus commands/statements appear separated from the text of paragraphs in a monospaced font. For example:

```
INSERT INTO emp (empno, ename) VALUES (1000, 'SMITH');
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

Example statements may include punctuation, such as commas or quotation marks. All punctuation in example statements is required. All example statements terminate with a semicolon (;). Depending on the application, a semicolon or other terminator may or may not be required to end a statement.

Uppercase words in example statements indicate the keywords within Oracle SQL. When issuing statements, however, keywords are not case sensitive.

Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate, your opinions are the most important input we receive. At the back of this manual is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals.

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

Contents

CHAPTER

1 Understanding Replication

This chapter introduces the features that support advanced distributed applications:

- basic replication
- advanced replication option
- data propagation methods
- configurations

Note: This manual describes features that are available to users who are interested in using the distributed and advanced replication options. Basic replication requires the distributed option and PL/SQL. Symmetric replication requires the distributed option, PL/SQL, and the advanced replication option.

Overview

The Oracle Server provides a variety of methods to replicate your data. This chapter begins with a discussion of store–and–forward, or asynchronous, replication. For asynchronous replication you can select the method that best suits your needs, from basic primary site replication to advanced dynamic and shared ownership models. Real–time (synchronous) and procedural replication, which have specialized uses, are described later in this chapter. To help you understand these replication options, this chapter discusses supporting mechanisms.

Basic Replication

Basic replication uses *read–only snapshots* to enforce a form of a primary site replication. As shown in Figure 1 – 1, a read–only snapshot is a *full copy* of a table, or a *subset* of a table, that reflects a recent state of the master table. A snapshot is defined by a distributed query that references one or more master tables, views, or other snapshots. A database that contains a master table is referred to as the master database.

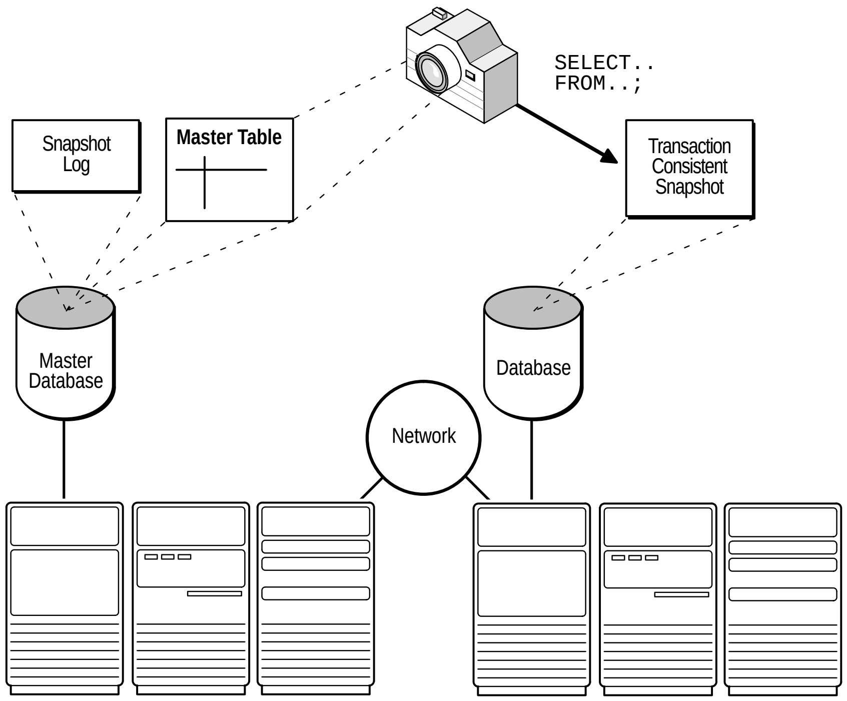


Figure 1 – 1 Table Replication Using Snapshots

Each replica (or copy) of the master table is called a snapshot because the information captured at a moment in time can be periodically *refreshed* to reflect a more recent transaction–consistent state of the master table.

A *simple snapshot* is based on a single remote table and has none of the following: distinct or aggregate functions; GROUP BY or CONNECT BY clauses; subqueries; joins; or set operations. If a snapshot's defining query contains any of these clauses or operations, it is a *complex snapshot*.

Advantages of Read–Only Snapshots

Maintaining read–only snapshots of a master table among the nodes of a distributed database is often a useful feature for the following reasons:

- Queries can be issued against a local snapshot. Therefore, associated query performance is fast because the requested data does not have to be shipped over a network.
- If the master site becomes unavailable, for example, because of a network failure, you can continue to query the read–only copies of this data.

Groups of snapshots can be refreshed to a single point in time, allowing you to maintain transactional consistency between copies of several related master tables.

Advanced Replication

The advanced replication option supports a symmetric, update–anywhere replication model; that is, all copies of data can potentially be updated, and ultimately all sites converge on the same data.

Replication Groups

A *replicated object* is a database object that is copied to multiple sites in a distributed system. *Replication groups* are

- the basic unit for controlling and managing advanced replication
- typically created by the replication administrator for all the replication objects that are
  – associated with a particular application
  – to be replicated to a set of sites

When you issue a *data manipulation language* (DML) or *data–level* statement against a replicated table, that update is ultimately propagated to every other replica of the table. Oracle applies DML changes to each replica in a transactionally consistent manner to ensure data consistency and referential integrity between tables.

Oracle also applies DDL (*data definition language* or *schema–level*) changes to each replica, such as adding a column to a table.

Oracle allows you to replicate

- tables
- objects that support those tables, such as
  - views
  - triggers
  - packages
  - indexes
  - sequences
  - synonyms

Supporting objects are replicated because their SQL Data Definition Language (DDL) statements are replicated. For example, supporting objects are created by propagating and executing the same CREATE statement at each site.

Oracle allows you to define, replicate, and manage groups of replicated objects (replication groups) as a unit. Please note the following:

- The members of a replication group can span multiple schemas.
- A schema can contain multiple replication groups.
- A replicated object can be a member of only one replication group.

Replication Sites

A replication group can be replicated (copied) to one or more replication sites. There are two basic kinds of replication sites: master sites and snapshot sites.

A *master site* must receive a *full copy* of all of the objects in the replication group. Each master site propagates, or pushes, its changes to every other master site for the replication group.

A *snapshot site* can receive a *subset* of the objects in the replication group. For example, you may choose to replicate only selected tables to a snapshot site, or even selected portions of a table. Table–level information is replicated at snapshot sites in the form of read–only or updatable snapshots.

Read–only snapshots can be used only for queries and only the master table can be updated (see the following table). *Updatable snapshots* provide a local, updatable copy of a remote master table and can be defined to contain a full copy of a master table or a defined *subset* of rows in the master table that satisfy value–based selection criteria.

Read–Only Snapshots	**Updatable Snapshots**
for queries only	for queries and updates
can be simple or complex (derived from a single master table or more than one master table)	must be simple (derived from a single master table)

S*napshot sites* must have an associated master site (although this master site can change if necessary), and unlike master sites, snapshot sites only push their changes to their *associated master site*. Snapshot sites can also pull down changes from their associated master site. The propagation mechanisms used by master and snapshot replication sites are explained in greater detail on page 1 – 12.

A *replicated environment* consists of a replication group, the replicated objects in the group, and the snapshot and master sites containing replicas of the group. Every replication group must have one and only one *master definition site*. The master definition site is used as the control point for performing administrative activities. Although data–level changes can be made at any site participating in a replicated environment, schema–level changes must be performed at the master definition site. However, if you experience a network outage, the master definition site can be relocated to another master site from any master site in the system.

As shown in Figure 1 – 2, a replication site can participate in multiple replication groups. For example, Site A in Figure 1 – 2 is a s*napshot site* for schema 3, the *master definition site* for schema 2, and the *master site* for schema 1.

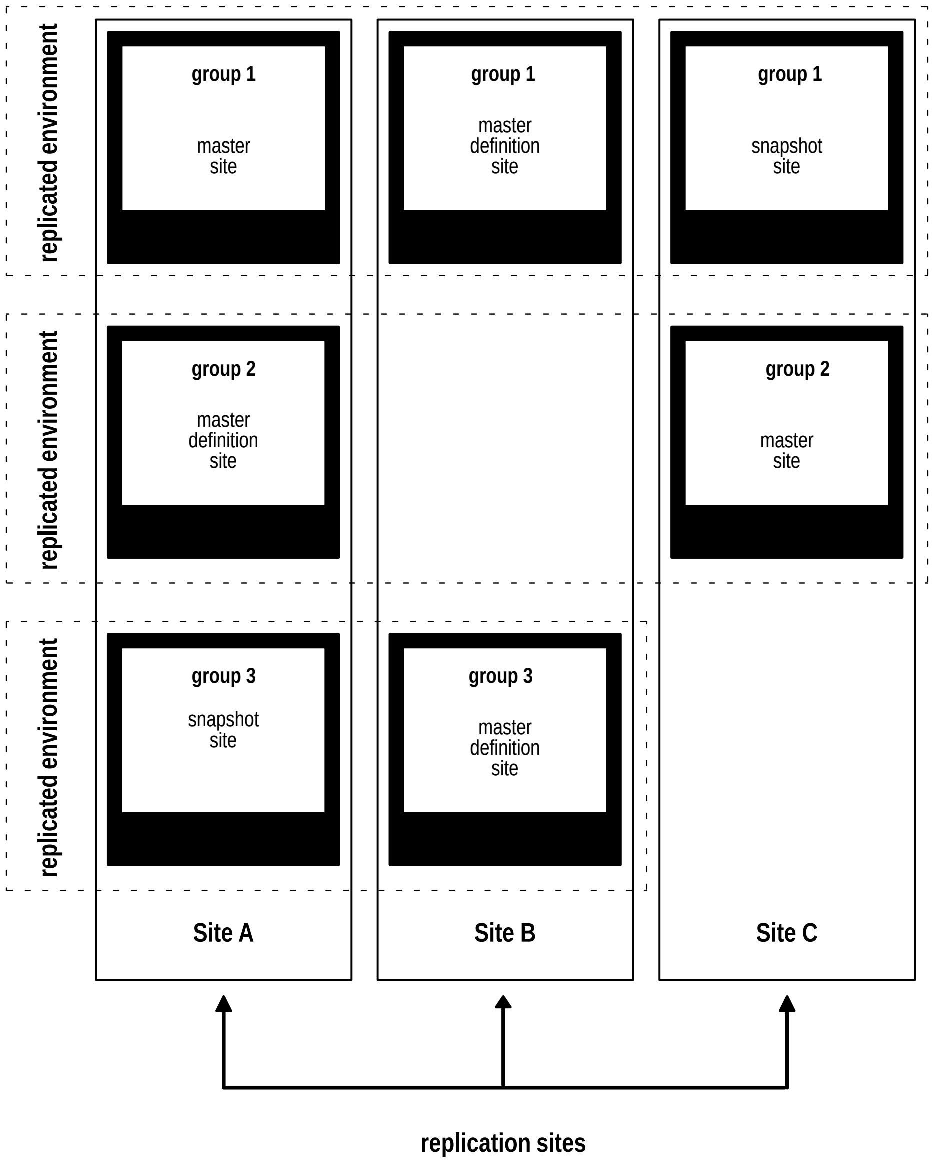


Figure 1 – 2 Replication Sites in a Replicated Environment

Replication Catalog

The *symmetric replication facility* uses a replication catalog to maintain information, such as which objects are being replicated, where they are being replicated, and how updates need to be propagated to these replicas. This replication catalog consists of a set of database tables that can be backed up and recovered (like any other tables).

Additionally, these tables are themselves replicated to each master replication site, ensuring that there is *no single point of failure* in your replicated environment.

Once you have configured your replicated environment, any data–level changes that you make will be propagated to all of the associated replication sites. No special commands or interfaces are required.

Schema–level changes, such as those used to configure and administer a replicated environment, must be made using a replication interface. Oracle provides a number of packaged procedures that you can call to administer your replicated environment. These procedures are described in Chapter 12, as well as in other parts of this document.

Oracle Replication Manager

Oracle Replication Manager, which can be launched as an applet from Oracle Enterprise Manager or run as a stand–alone product, is a graphical tool that lets you configure, schedule, and administer your replicated environment from a single location. Replication Manager's point–and–click interface lets you create replication groups consisting of tables as well as their supporting objects, such as indexes, triggers, views, and conflict resolution procedures. You can drag and drop a group onto other databases to add new replication sites to your environment. If you add or remove objects from a replication group, the changes are automatically deployed at every site.

Replication Manager also helps you troubleshoot and resolve error conditions. You can view the deferred transaction queue at each location, and reschedule or force immediate execution of these transactions as needed. You can also view outstanding administrative requests for each location. Additionally, you can take advantage of Enterprise Manager's event management capabilities, which provide a proactive monitoring capability of replication status across multiple site.

Additional Information: To learn more about this tool and its graphical user–interface (GUI), consult the Oracle Replication Manager online help system.

Note: The rest of this manual refers to the *procedural*, rather than the Replication Manager GUI, interface for replication administration.

Configurations

The symmetric replication facility supports both full–table replication and replication of subsets of tables. The following mechanisms are supported:

- single master with multiple *read–only* snapshot sites
- multi–master replication
- single master with multiple *updatable* snapshot sites
- hybrid configurations of the above

Read–Only Snapshot Sites

Multiple read–only snapshot sites can be used to provide local access to remote master tables. Having a local snapshot of the data improves query response time. Updates can only be issued against the remote master table.

A read–only snapshot is refreshed from its associated master table in a transactionally consistent manner at a time–based interval or on demand.

Read–only snapshots are easily maintained, and do not require the advanced replication option. Examples of read–only snapshot site uses are included on page 2 – 6.

Multi–Master Replication

As shown in Figure 1 – 3, *multi–master replication* supports full table, peer–to–peer replication between master tables. All master tables at all sites can be updated.

Changes applied to any master table are propagated and applied directly to all other master tables. These changes can be propagated either synchronously or asynchronously.

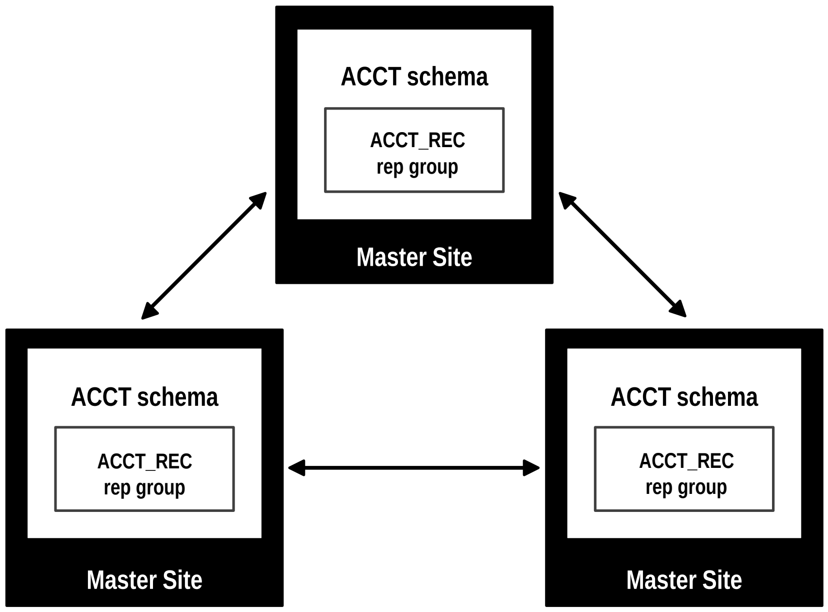


Figure 1 – 3 Multi–Master Replication

Updatable Snapshot Sites

A single master site can be used to consolidate information provided from multiple updatable snapshot sites. In the example shown in Figure 1 – 4, orders can be entered at each of your sales offices, but all orders are processed at the corporate headquarters site.

You can think of the ORDERS *snapshot* at each of the sales offices as a writeable subset of the ORDERS *table* at headquarters.

An updatable snapshot is refreshed from its associated master table in a transactionally consistent manner at a time–based interval or on demand. Changes from the updatable snapshot can be forwarded to its master table either synchronously or asynchronously.

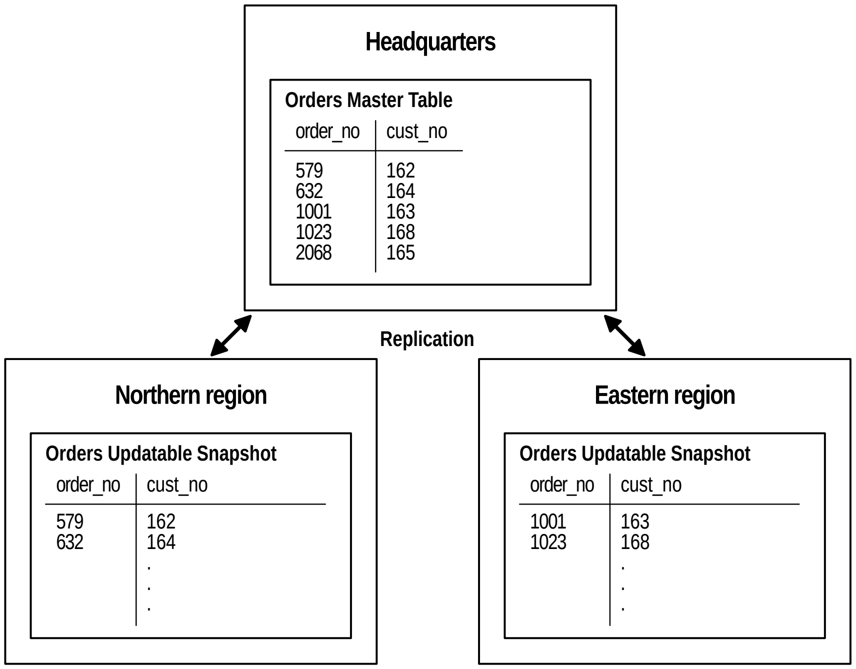


Figure 1 – 4 Using Updatable Snapshot Sites for Information Consolidation

Hybrid Configurations

Multi–master table replication and updatable snapshots can be combined in hybrid (mixed) configurations to meet different needs. Specifically, snapshot masters can be *n–way* replicated (multi–mastered). N–way snapshot master replication means that

- there is any number of snapshot master sites
- full–table and table subset replication can be combined in one system

For example, as shown in Figure 1 – 5 n–way replication between two snapshot masters can support full–table replication between two master sites supporting two geographic regions. Snapshots can be defined on the snapshot masters to replicate full tables or table subsets to sites within each region.

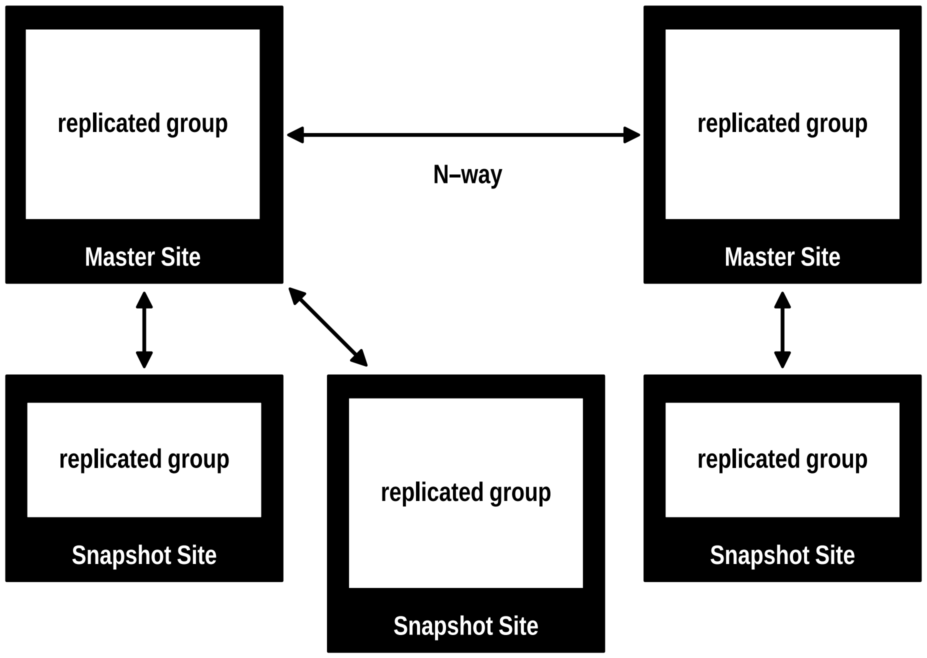


Figure 1 – 5 Hybrid Configuration

An added benefit of this n–way replication is that snapshots can be remastered from the other master sites to improve availability. If one master site fails, its snapshots can refresh themselves from the surviving master site and continue processing. This configuration, in which a master site resembles a ”hub” with ”spokes” to snapshot sites, also allows the two n–way connected master sites to function as fail–over sites for each other (see the “Survivability” section on page 2 – 12).

Some of the key differences between updatable snapshots and replicated masters include the following:

- Replicated masters must contain data for the *full table* being replicated, whereas snapshots can replicate *subsets* of master table data.
- Multi–master replication allows you to replicate changes for each transaction as they occur, while snapshots are set–oriented, propagating changes from multiple transactions in a more efficient, *batch–oriented* operation, but at less frequent intervals.

- If any conflicts occur as the result of changes being made to multiple copies of the same data, these conflicts are detected and resolved by the master sites.

Propagating Changes Between Replicas

When you update an object in a replicated environment, this change must ultimately be propagated to all master sites, as well as to any appropriate snapshot sites.

Asynchronously Propagating Data–Level Changes

Oracle uses two primary mechanisms for asynchronously propagating data–level (DML) changes between replication sites:

- *deferred transactions*
- *snapshot refresh*

Deferred Transactions

For multiple master replication and replication from updatable snapshots to masters, Oracle generates a trigger and stored procedure for that table to support the replication of data–level changes. When you perform a change locally, Oracle fires a generated trigger that builds a *remote procedure call* (RPC) to a packaged procedure at the remote site.

The arguments to these procedures contain the information necessary to apply the change at the remote site. The RPC is stored in the local *deferred transaction queue*. The local deferred transaction queue is stored in a database table, which can be viewed using the *DefTran* view.

As shown in Figure 1 – 6, when you commit an update to a table, Oracle inserts the necessary remote procedure calls into the deferred transaction queue. Oracle uses a second table, which can be viewed using the *RepSites* view, to store the destinations of each of the sites to which to push this queue. When you add another master site, its location is added to this destinations view. There is only one entry in the queue for each deferred transaction, regardless of how many destinations are listed in the replication sites view.

In separate transactions, the entries in the deferred transaction queue are propagated to each site listed in the replication site's destination view. Oracle does not remove a transaction from the local deferred transaction queue until it has been successfully propagated to all of the destinations listed in the replication sites view.

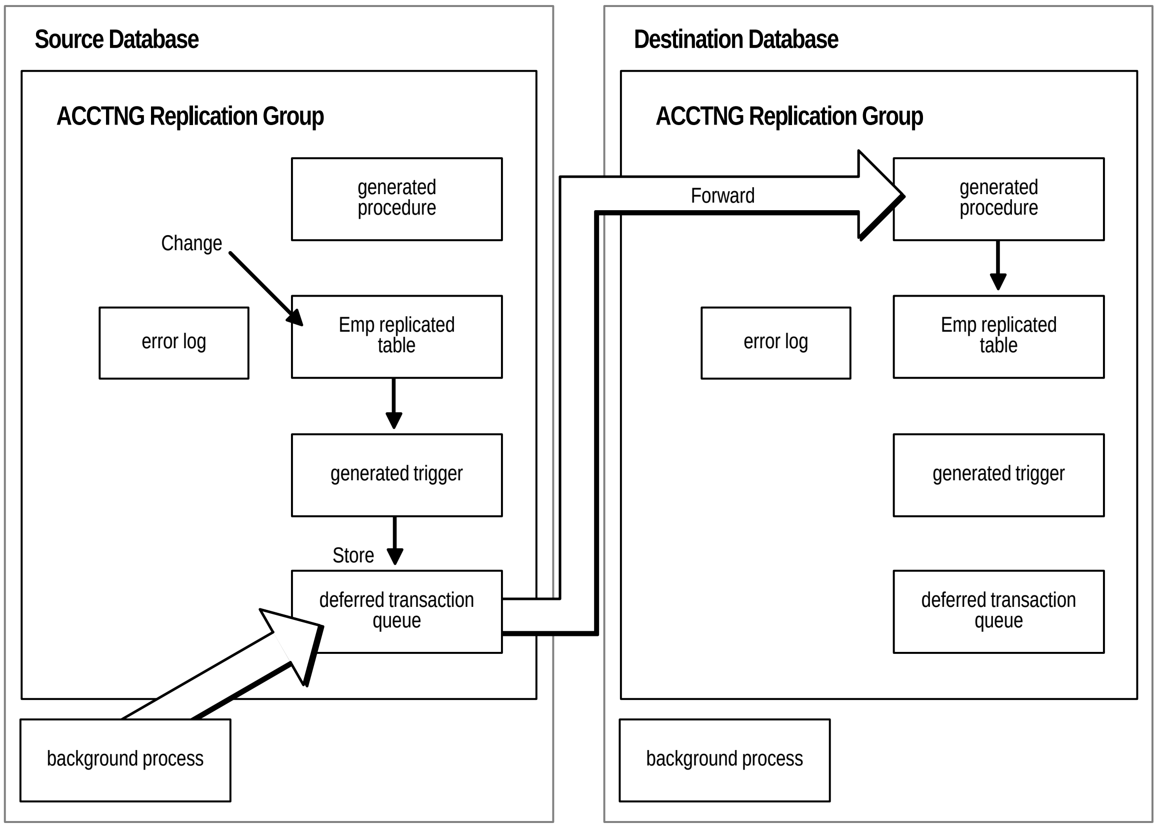


Figure 1 – 6 Propagating Data Level Changes

Snapshot Refresh

Snapshot sites use the deferred transaction mechanism described in the previous section to propagate data–level changes from updatable snapshots to their associated master table.

To propagate changes from the master tables to the associated read–only and updatable snapshots, Oracle uses the *snapshot refresh* mechanism,instead of the row–level replication mechanism. During a snapshot refresh,

- any changes made to the master table since the snapshot was created or previously refreshed are applied to the snapshot
- changes from multiple transactions are propagated in an efficient, batch–oriented operation

Snapshot Refresh Groups

Because snapshot refresh is set–based, if you require that two or more snapshots be refreshed to a single point in time, create a *snapshot refresh group*.

For example, to preserve a master–detail relationship between a pair of snapshots, place the related snapshots in the same refresh group.

Alternative Replication Mechanisms

In addition to deferred transactions and snapshot refreshes, Oracle provides two alternative mechanisms for propagating data–level changes among replicas:

- synchronous row–level replication
- procedural replication

As described below, these mechanisms have specialized uses.

Synchronous replication uses the same *row–level replication* mechanism to propagate data–level changes as deferred transactions, but does not use a deferred transaction queue.

As shown in Figure 1 – 7, when you make a change to a replicated table, Oracle fires a trigger which calls a packaged procedure at each master site that applies the change.

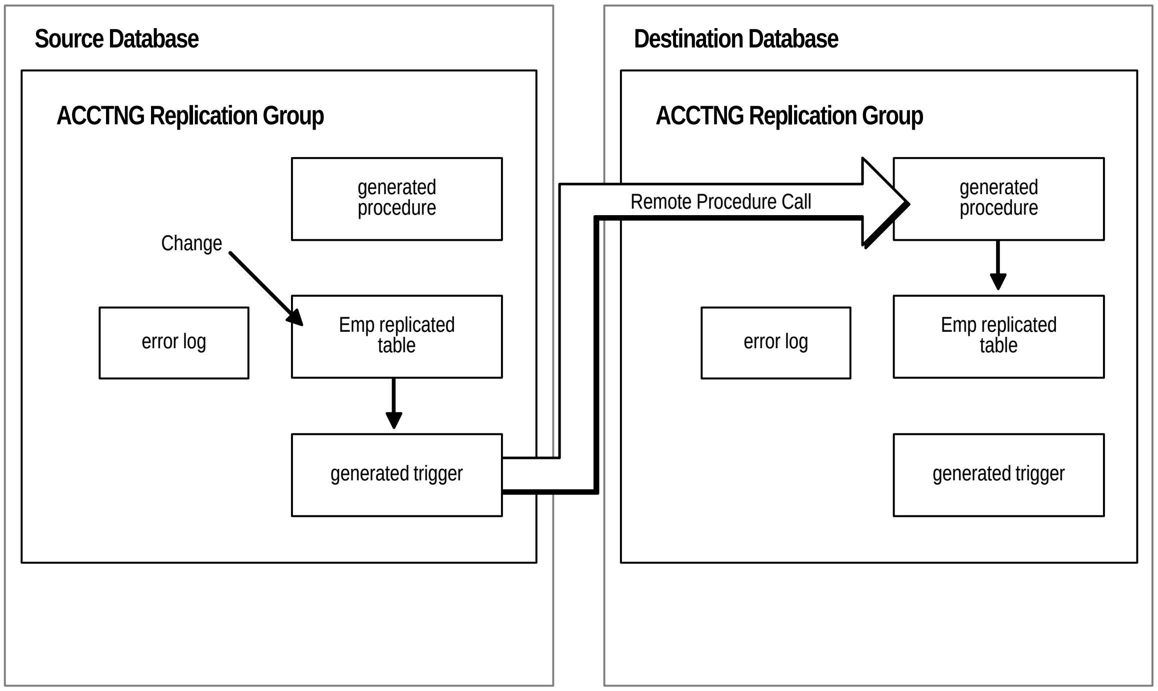


Figure 1 – 7 Propagating Data–Level Changes Synchronously

Your change must be successfully applied at both the local table and at any replicated copies of the table, or rollback occurs. Synchronous replication is most useful in situations where you have a stable network and require that your replicated sites remain continuously synchronized.

You can choose to create a replicated environment in which some sites propagate changes synchronously while others use asynchronous propagation (deferred transactions).

> **Note:** The advantages of synchronous propagation (no conflicting changes, never out of date) can only be fully realized if all sites are both sending and receiving changes synchronously.

Procedural replication only replicates the *call* to a stored procedure that is used to update a table. Procedural replication does not replicate the update itself.

After you replicate a procedure, you can generate a *wrapper* for this procedure at each site. When you call the procedure at the local site, the wrapper ensures that a call is ultimately made to the same procedure at all other sites in the replicated environment. This call can be made either synchronously or asynchronously (using the deferred transaction queue described on page 1 – 12).

Note: Do not confuse this wrapper with the PL/SQL Wrapper.

Suggestion: Procedural replication is often used for large batch–oriented operations that can be run serially (such as purging data that was ”logically” deleted), and can be used in conjunction with row–level replication. For more information on using procedural replication, refer to page 8 – 13.

CHAPTER

2

Designing a Replicated Environment

This chapter describes many of the issues that you will need to consider when designing your replicated environment. The topics discussed include the following:

- store–and forward vs. real–time data propagation
- distributed vs. replicated data
- master vs. snapshot replication
- replication models
- survivability

Store–and–Forward vs. Real–Time Data Propagation

When choosing between store–and–forward (asynchronous) and real–time (synchronous) data propagation, you are primarily making a choice between availability and complexity.

Both synchronous and asynchronous replication have the advantage of allowing you to query and update local copies of the data, thus improving response time.

With a completely synchronous environment, you have the advantage of always having the most up–to–date information at all sites. You always make decisions based on the most current information, and conflicting updates never occur.

With a completely asynchronous environment, you have the advantage of continuous availability. No site is dependent upon another to allow an update to be made. If one site goes down, you can switch to another and continue working. Your business needs will determine which propagation method is most appropriate for you.

Synchronous

You can use synchronous data propagation in either a distributed or replicated environment. When determining whether to use synchronous data propagation, you need to consider the following issues:

- Changes at other sites are immediately reflected at your local site.
- Although in a replicated environment, the same data can be updated at multiple sites, you do not have to worry about conflicting updates occurring.
- If you experience network failure with any replicated site to which you are synchronously propagating changes, you will not be able to perform local updates until either the network failure is resolved or you drop the unavailable site from your replicated environment.
- Your updates may have a slower response time, because you must wait for a response from all sites before committing or rolling back a transaction.
- Synchronous procedural data propagation is an option for environments with *read–often/write–occasionally* data (for example, the price list for certain businesses).

Synchronous replication is appropriate in situations where absolute consistency between replicated data is a requirement.

To make synchronous replication practical, you need to take steps to ensure stable networks and systems, or have flexibility in scheduling updates to allow for delays due to network and system outages.

Asynchronous

You can use asynchronous data propagation in replicated environments only. When determining whether to use asynchronous data propagation, you need to consider the following issues:

- Failures of remote sites or networks do not block other sites from querying and / or updating replicated data locally.
- For maximum fault tolerance, a dedicated *failover* site can support mission critical operations.
- Your response times for updates is improved over using synchronous propagation, because you do not have to wait for a response from a remote site.
- Deferred transactions are propagated at whatever interval is most convenient for you. You can enforce ”near real–time” replication by using an interval of a few seconds; you can replicate your changes at a fixed time or interval; or you can replicate your changes on demand.
- Changes at other sites are not immediately reflected at your local site, resulting in temporary inconsistencies between replicas.
- Conflicting changes can be made at multiple sites. These conflicts will not be detected until the changes are propagated. Oracle provides built–in mechanisms for detecting and resolving these conflicts.

Distributed vs. Replicated Data

If you determine that you prefer to use synchronous data propagation, you must next decide if you need to distribute or replicate your data. This choice will be primarily determined by whether you need all data at all sites or if data can be clearly divided between sites.

Distributed Data

A distributed system consists of multiple servers, each responsible for their own data. This data can be accessed using a network, and appears to the user to be a single server. Although the data can be accessed from multiple locations, there is only one physical copy of each piece of data.

Distributing your data can improve your performance by providing faster updates of the local subset of your data. It can also improve your availability. If one site in a distributed system becomes unavailable, you can continue to query and update the data at the remaining sites.

If you frequently perform queries that access multiple remote sites, you may experience some performance degradation because all of the data is at a single location. Additionally, if one of these sites becomes unavailable, you will not be able to complete this transaction until the site becomes available again.

For these reasons, a distributed model is most appropriate when you frequently query and update a distinct subset of your data from a single location, and seldom query or update the remaining portions. *Oracle7 Server Distributed Systems, Volume I* provides more detail on distributing your data.

Replicated Data

In a replicated environment, each site contains a copy of all necessary data for that site, with multiple sites potentially having multiple updatable copies of the same data. Maintaining multiple copies of your data at multiple sites will require greater system resources, and is therefore most appropriate in situations requiring frequent local access to data from multiple locations.

Replicating your data can improve your performance by providing faster queries of all of your data. It can also provide improved availability to all of your data for queries. Even if your local site goes down, you can still access a complete copy of your data at another replicated location.

Synchronously replicating your data can decrease the availability of your data for updates, however. If one site becomes unavailable, you cannot update any other replicas until the downed site either becomes available or is dropped from the replicated environment. For this reason, you may prefer to use ”near real–time” asynchronous replication, if you determine that you need to replicate, rather than distribute your data.

Master vs. Snapshot Replication

If you choose to use asynchronous replication, you must decide how you want to replicate your data. There are two key decisions that will influence how you design your replicated environment:

- How frequently do you want to propagate your changes?
- What data do you require at each site?

When you use asynchronous replication, you need to choose a replication interval that supports your business requirements.

If you make frequent updates, yet require very up–to–date information, you will need to set a short propagation interval, to simulate real–time replication. If you want to minimize your communication costs, you may prefer to increase this interval to once a day, or even once a week.

If you require information that is up to date as of a particular point in time, you may prefer to propagate your changes at a scheduled time. For example, you might want to retrieve new pricing information at the start of each quarter.

Finally, if you do not know when your network connection will be available, such as if you are using a laptop computer to collect data on a sales call, you may want to propagate your changes on demand.

Because snapshot sites propagate changes only to their associated master site, and because they pull down changes from their master in an efficient, batch–oriented manner, sites requiring on–demand data propagation should typically be snapshot sites.

In addition to determining an appropriate replication interval for each site, you must also determine what data is appropriate for each site.

Certain sites may not require full copies of a replication group, and you may prefer to replicate only a subset of the data to these sites to conserve resources. For example, suppose you have regional branches of your business, each with their own client base. Each branch may require full copies of certain tables, such as price lists, but only subsets of others, such as customer lists. Because master sites must contain full copies of a replicated group, these branch sites would need to be created as snapshot sites.

The following sections of this chapter describe the various models that Oracle's symmetric replication facility supports in more detail, including hybrid models that combine snapshot and multi–master replication.

Replication Models

With symmetric replication, applications can be built that employ both standard primary site replication techniques and advanced replication techniques.

Basic Replication

There are a variety of usage scenarios that can be implemented using read–only snapshots, or basic replication. Some examples are described below. Each of these scenarios describes a form of primary site data replication.

With *primary site* replication, each piece of information is owned by one site, and this ownership never changes. Other sites "subscribe" to the data owned by the primary site, which means that they have access to read–only copies of the replicated data. With primary site replication, you never need to worry about any discrepancies between data that is being updated at two different locations.

Information Offloading

Because snapshots can provide a local copy of your data, they can be accessed faster than remote data. For example, if your the order entry site requires fast online transaction processing (OLTP) capability for order entry against the inventory table, and the marketing / decision support site requires lengthy queries against the same table, you can use *information offloading*. Information offloading, in this example, would provide the decision support site with a separate and *full copy* of the inventory table as a read–only snapshot for local analysis (see Figure 2 – 1).

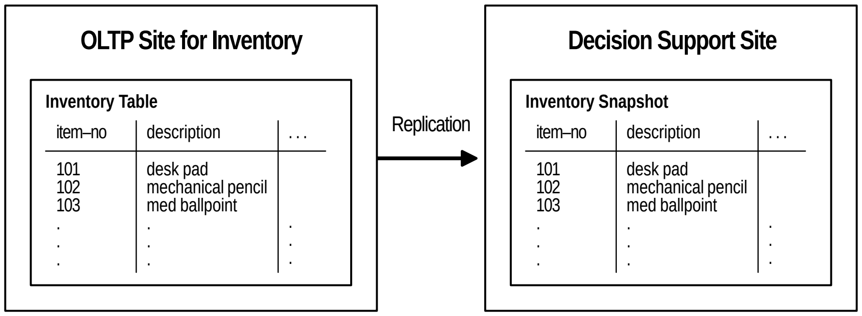


Figure 2 – 1 Information Offloading

Information Distribution

While in the above example, the read–only replica is a full copy of the original table, you may choose to replicate only selected portions of the table at each site. For example, suppose that a central copy of all of your customer information is maintained at your headquarters in New York.

Portions of this information might be used by your sales offices around the world. Instead of replicating the entire table at each sales office, you need only replicate the portions appropriate for that region, as shown in Figure 2 – 2.

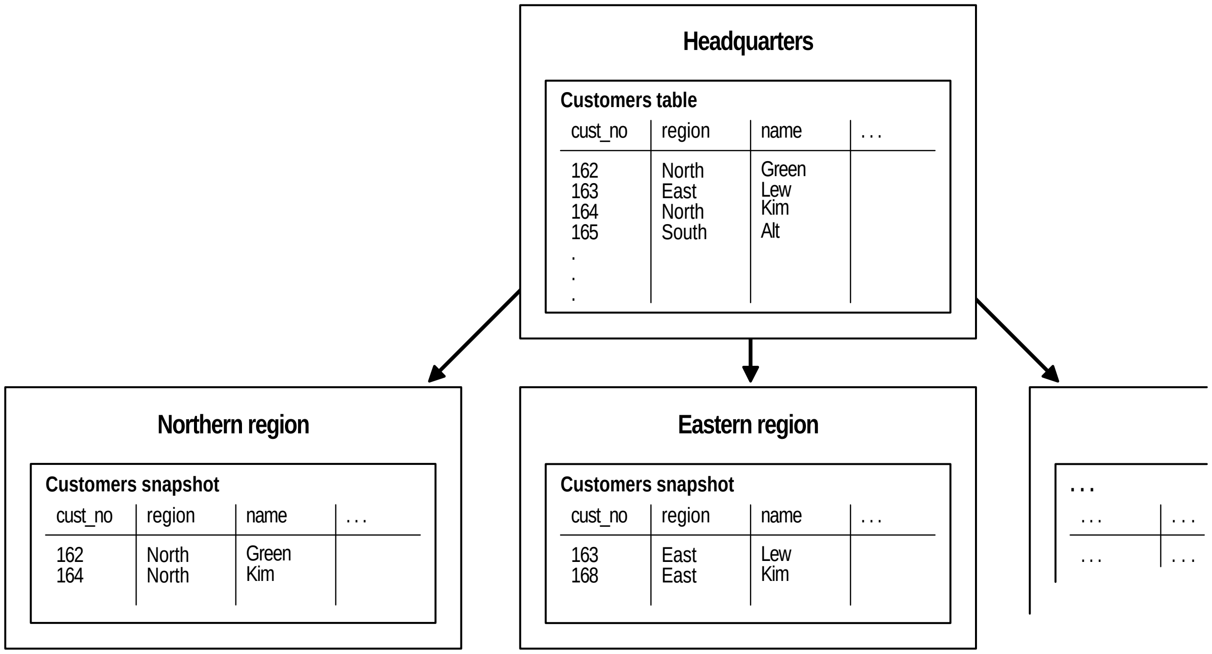


Figure 2 – 2 Information Distribution

Advanced Primary Site Models

Read–only snapshots support a primary site ownership model only. Other replication technologies can support this model by restricting updates at the application level to a single site.

For performance reasons, however, you might want to allow local updates of the data, thus allowing multiple sites to have access to a single table. You can still successfully avoid conflicts by implementing an advanced form of primary site ownership.

Instead of designating one site as the owner of the entire table, each site is allowed to “own” a distinct portion of this table. That is, each site would be allowed to modify only a subset of the rows or columns in each table. You might think of this as allowing each site to own a distinct horizontal or vertical partition of the data in a single table.

Ownership can either be enforced by your application, or can be enforced by using a combination of triggers, views, procedures, and horizontally partitioned updatable snapshots.

Horizontal Partitioning

For example, you could implement a distributed order entry system such that each order entry site in each sales office owned distinct horizontal partitions of tables (such as CUSTOMERS, ORDERS, and ITEMS) that contain the orders and customer information for the customers serviced by that office. Your central headquarters site could then maintain a read–only view of the master table containing all orders and customer information across all sales offices.

The CREATE statement for a snapshot of your CUSTOMERS table might look like

```
CREATE SNAPSHOT customers FOR UPDATE AS
   SELECT * FROM customers@hq.com WHERE region = 'North East';
```

Vertical Partitioning

You can further subdivide the ownership of a table by allowing a site to modify only selected columns of a given row. For example, suppose you have a stock table. You might want to allow different regional sites to update the AMOUNT_AVAILABLE column for different items (rows), but only allow someone from your headquarters site to update the item description columns for every row.

> **Note:** Ownership of vertical partitions requires the use of column groups. Column groups are described on page 6 – 4.

Dynamic Ownership

In addition to basing ownership on a static column as shown in the primary site ownership examples, you can also base ownership on a field that can be updated. This would result in a form of dynamic ownership of data. With *dynamic ownership*, the ability or right to update replicated data moves from site to site while ensuring that at any given point in time only one site may update the data.

Work Flow

One form of dynamic ownership is work flow. *Work flow* is a simple form of exclusive ownership commonly used by business applications. To implement a work flow model of conflict avoidance, your application must guarantee the following:

- The control of ownership is ordered.
- Each site can only update data that it owns; that is, only rows with a given status are updatable.
- Each site must push the ownership to the next site by updating the status to the next state.

For example, within an order processing system, the processing of orders typically follows a well ordered series of steps such as: entered, approved, shipped, billed, collected, and accounted for. Sophisticated centralized systems allow the application modules that perform these steps to act on the same data contained in one integrated database.

Each application module acts on an order, that is, performs updates to the order data, when the state of the order indicates that the previous processing steps have been completed. For example, the application module that ships an order will do so only after the order has been entered and approved.

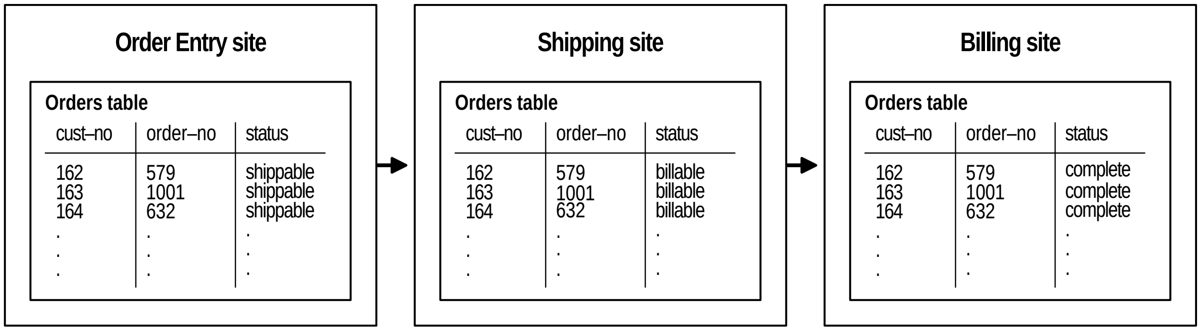


Figure 2 – 3 Dynamic Ownership Order Processing System

By employing a dynamic ownership replication technique, such a system can be distributed across multiple sites and databases. Application modules can reside on different systems. For example, order entry and approval can be performed on one system, shipping on another, billing on another, and so on. Order data is replicated to a site when its state indicates that it is ready for the processing step performed by that site. Data may also be replicated to sites that need read–only access to the data. For example, order entry sites may wish to monitor the progression of processing steps for the orders they enter.

Shared Ownership

All of the usage methodologies described thus far, that is, primary site ownership and dynamic ownership, share a common property — at any given point in time, only one site may update the data while the other sites have read–only access to replicated copies of the data.

In some situations, however, it is desirable to allow multiple sites to update the same data, potentially at the same time. For example, it may be desirable to replicate customer data across multiple sites and systems rather than maintaining customer data centrally or maintaining it separately and redundantly within each system. Different sites, though, may need to update this data.

Update Conflicts

Suppose that you replicate customer data across sales office order entry sites and headquarters sites. One element of the customer data is the customer address. What happens if a customer's address is changed at both a sales office and a headquarters site at the same time?

This occurrence is known as an update conflict. Replicated data has become inconsistent because the replicated data was updated at multiple sites. If you cannot tolerate such inconsistencies, you must either carefully partition ownership of your data or only allow for synchronous propagation of changes between sites. If all sites in your replicated environment are propagating changes to one another synchronously, update conflicts cannot occur. If, however, you have even one site sending or receiving changes asynchronously (for example, if you have an updatable snapshot site), you have the potential for conflicts. For some applications, these temporary inconsistencies can be permitted as long as they can be detected and resolved to ensure that over time the replicated data converges to a consistent state at all sites.

Update Conflict Detection, Notification and Resolution

The symmetric replication facility supports these capabilities. For example, in the scenario described previously, Oracle detects that the update conflict on the customer's address has occurred and automatically invokes an application–specific conflict resolution routine to restore the replicated data to a consistent state. Oracle can also invoke a notification routine to send an alert that a conflict has occurred.

The symmetric replication facility provides a number of standard resolution routines from which the application developer can select. Standard resolution routines include: timestamp determined most recent update, commutative resolution of additive updates, applying the change from the site with the highest priority value, and min / max selection of updates. Alternatively, for more specialized cases, the application developer can write his or her own routines.

In the scenario above, a routine that uses timestamps to determine the most recent update can be employed so that the customer's address converges to the most recent update of the address at all sites. Update conflicts on the address will be automatically detected and immediately resolved at each site by selecting the most recent of the updates.

Sophisticated Uses of Shared Ownership

Shared ownership allows symmetric replication to be employed where primary site ownership and dynamic ownership methodologies would be too restrictive. As such, in those cases where temporary inconsistencies can be permitted and conflict resolution routines devised, it can offer greater flexibility.

For example, an earlier discussion on page 2 – 8 described how a distributed order entry system could be implemented using primary site replication techniques with horizontal partitioning.

In this scenario each sales office owned a distinct horizontal partition of the tables containing orders and customer information for the customers serviced by that office. Each sales office entered orders for its customers, but no others.

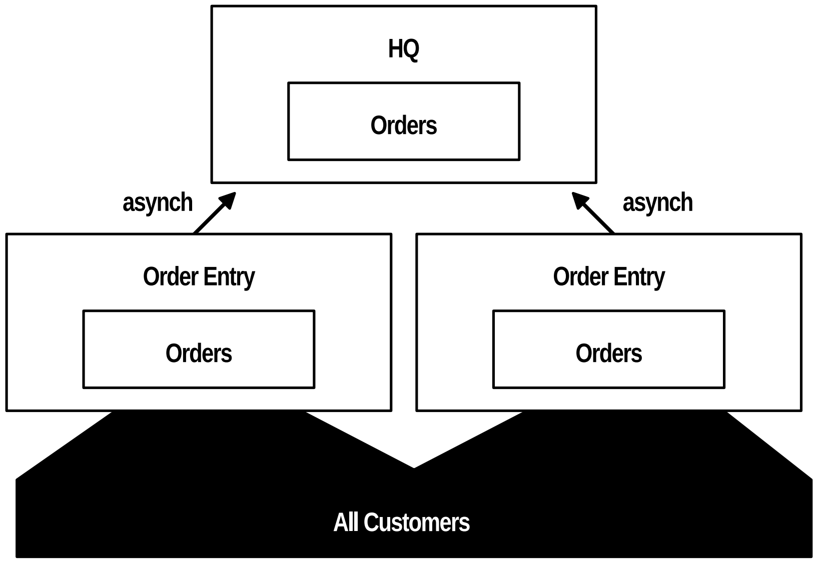


Figure 2 – 4 Shared Ownership Order Entry System

For some businesses, though, this is not the model. For example, a retail chain may have several stores in a metropolitan area. Customers may frequent the store closest to where they live, but they will go into other stores; and these others stores will want to take their orders when they do. If multiple stores perform updates to the same customer and order data, as illustrated in Figure 2 – 4, update conflicts potentially could occur. Sophisticated application developers can identify these conflicts and either select standard resolution routines or devise their own to implement such systems.

Survivability

The Oracle symmetric replication facility also supports system survivability. Two sites maintaining replicated master tables can serve as fail–over sites for each other. If one site fails, processing can continue on the surviving site against the data that has been replicated from the failed system. When the failed system is restored, its database can be re–instantiated from the failover site, or conflict detection / resolution technology can be employed to re–establish consistency between the two databases. Symmetric replication, therefore, adds another option for system fault tolerance.

The Oracle Parallel Server provides system fault tolerance in locally connected cluster or massively parallel environments. In these environments, the Parallel Server will usually be the preferred option. Symmetric replication extends Oracle's system fault tolerance capability to geographically separated systems and local non–clustered environments. Another option is a standby database configuration which can offer performance advantages while also providing protection across geographically separated systems.

CHAPTER

3

Read–Only Snapshots

This chapter explains how to use read–only snapshots to provide copies of tables at remote sites, without requiring the use of remote queries. This chapter describes how to

- create read–only snapshots
- create snapshot logs for simple snapshots
- create snapshot refresh groups
- refresh snapshots

Note: The features described in this chapter are available only to users of the distributed option.

Note that most of the activities described in this chapter can be accomplished much more easily by using Oracle's Replication Manager, a GUI interface for replication. See the documentation for Oracle Replication Manager for more information.

Understanding Read–Only Snapshots

As shown in Figure 3 – 1, when you create a read–only snapshot, Oracle creates several internal objects in the schema of the snapshot. Do not alter, change data in, or delete these objects manually. At the snapshot site, Oracle creates a *base table*, named SNAP$_*snapshotname*, to store the rows retrieved by the snapshot's defining query. For simple snapshots, Oracle also creates an index on the ROWID column of the base table, named I_SNAP$_*snapshotname*. *You should not alter the base table in any way.* Do not add triggers or integrity constraints to the base table, unique indexes or modify their contents.

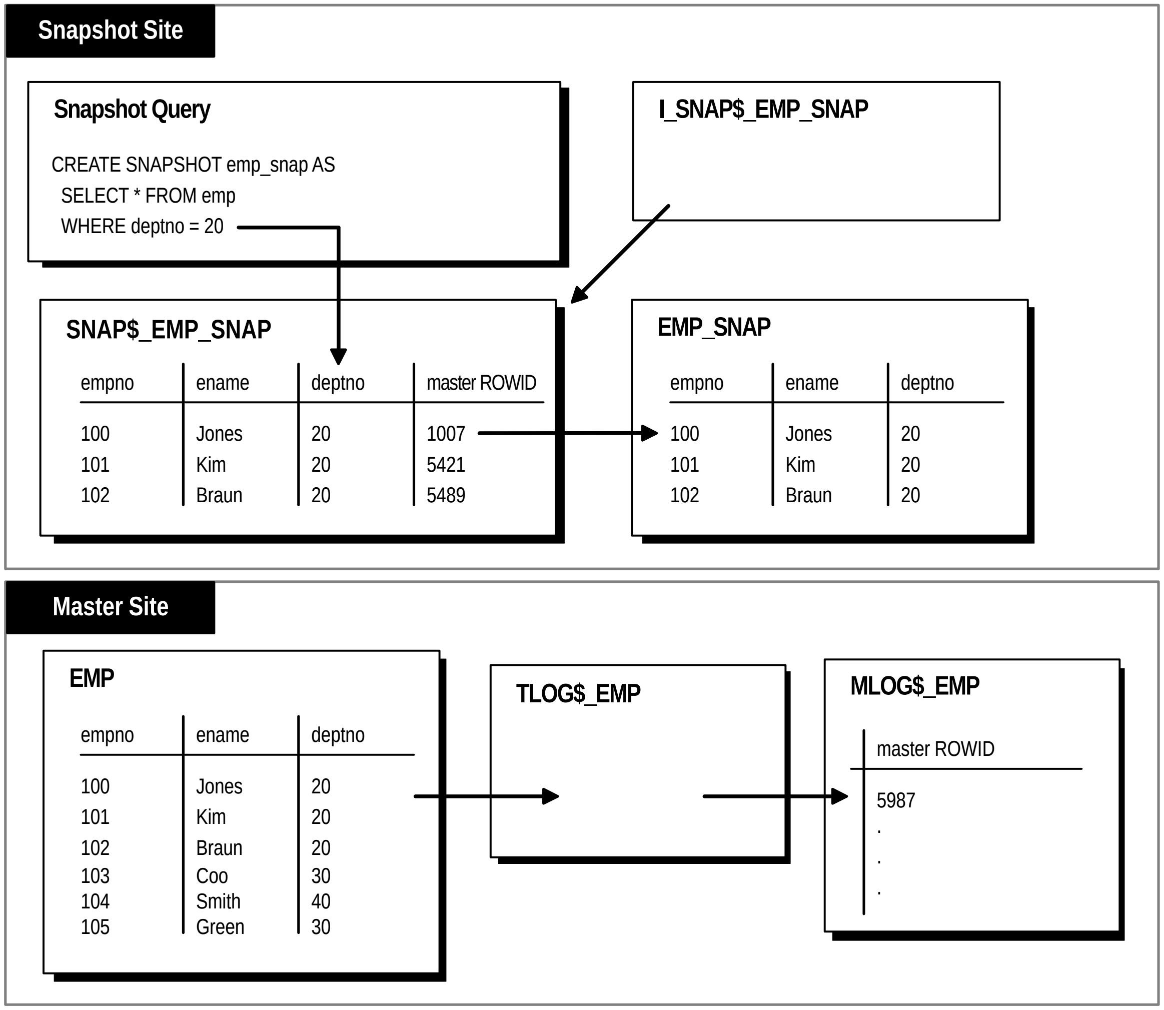


Figure 3 – 1 Snapshot Architecture

Oracle creates a read–only view of the base table that is used whenever you query the snapshot. This view uses the name that you provided when you issued the CREATE SNAPSHOT statement.

Oracle creates a second local view, named MVIEW$_*snapshotname*, on the remote master table. When you refresh a snapshot, Oracle uses this view to refresh the snapshot. Oracle stores the results of this query in the base table, replacing the previous snapshot data.

For simple snapshots, you can choose to create a *snapshot log* for the master table. This log is named MLOG$_*master_table_name* and the trigger used to update this log is named TLOG$_*master_table_name.* The information in this log allows you to perform a fast refresh of a simple snapshot.

With a *fast refresh*, only the changed rows of the snapshot, as indicated by the snapshot log, need to be updated. Each time that you make a change to the master table, Oracle tracks that change in the snapshot log, including the ROWID of the changed row. The generated index (I_SNAP$_) on the ROWID column of the base table allows these changes to be quickly applied to the snapshot. A complex snapshot, or a simple snapshot without a snapshot log, must be completely regenerated from the master table every time you refresh the snapshot. This is known as a *complete refresh.*

You should not drop the generated index. If you will never perform a fast refresh of the snapshot and do not want this index created, you can create your snapshot as a complex snapshot, for example by joining with DUAL.

A snapshot log can be used by multiple simple snapshots of a single master table. After you refresh a snapshot, any rows in the snapshot log that do not apply to any other snapshots of that master are removed from the snapshot log.

Creating Snapshots

You create a snapshot using the SQL command CREATE SNAPSHOT. As when creating tables, you can specify storage characteristics, extent sizes and allocation, and the tablespace to hold the snapshot, or a cluster to hold the snapshot (in which case all of the previous options do not apply). You can also specify how the snapshot is to be refreshed and the distributed query that defines the snapshot; this is unique to snapshots.

For example, the following CREATE SNAPSHOT statement defines a local snapshot to replicate the remote EMP table located in NY:

```
CREATE SNAPSHOT emp_sf
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE (INITIAL 50K NEXT 50K PCTINCREASE 50)
   REFRESH FAST
      START WITH sysdate
      NEXT sysdate + 7
   AS SELECT * FROM scott.emp@sales.ny.com;
```

Whenever you create a snapshot, Oracle immediately fills the base table with the rows returned by the query that defines the snapshot. Thereafter, the snapshot is refreshed as specified by the REFRESH clause; see "Refreshing Snapshots" on page 3 – 15.

Restrictions on Snapshots

Declarative constraints on snapshots and snapshot logs are not supported.

Snapshots of LONG columns are not supported.

Naming Snapshots

Snapshots are contained in a user's schema. A snapshot's name must be unique with respect to other objects in the schema. Although a snapshot name can be up to 30 bytes, keep snapshot names to 19 or fewer bytes. If a snapshot name contains more than 19 characters, Oracle automatically truncates the prefixed names of the underlying table and views, and appends them with a four–digit number to ensure uniqueness. This guarantees that the objects comply with the naming rules for schema objects.

Creating a Clustered Snapshot

You can create a snapshot in a cluster, just as you can a table. For example, the following statement creates a snapshot named EMP_DALLAS in the EMP_DEPT cluster:

```
CREATE SNAPSHOT emp_dallas
   ...
   CLUSTER emp_dept
   ... ;
```

The storage parameters of the cluster's data segment are used for the storage of the clustered snapshot, even if storage parameters are specified for the snapshot.

Creating Complex Snapshots versus Creating Local Views

When creating a complex snapshot, consider an alternative: creating simple snapshots and performing the complex query using a view in the snapshot database. Figure 3 – 2 illustrates the advantages and disadvantages of completing the same operation by the two different methods.

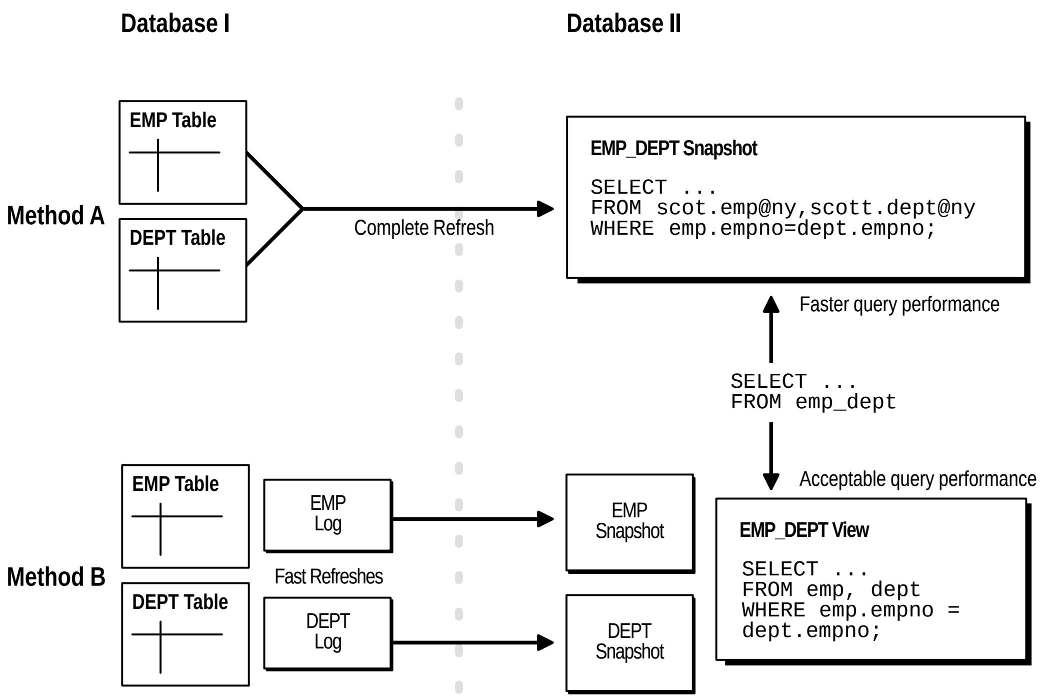


Figure 3 – 2 Two Methods for Complex Snapshots

Complex Snapshot

Method A shows a complex snapshot. The snapshot in Database II exhibits efficient query performance because the join operation has already been completed during the snapshot's refresh. However, complete refreshes must be performed in this case because it is a complex snapshot.

Simple Snapshots with a Joined View

Method B shows two simple snapshots in Database II, as well as a view that performs the join in the snapshots' database. Query performance against the view would not be as good as the query performance against the complex snapshot in Method A. However, the simple snapshots can be more efficiently refreshed using snapshot logs.

In summary, to decide which method to use:

- If you refresh rarely and want faster query performance, use Method A.
- If you refresh regularly and can sacrifice query performance, use Method B.

Privileges Required to Create Snapshots

To create a snapshot, you must have the following sets of privileges:

- To create a snapshot in your own schema, you must have the CREATE SNAPSHOT, CREATE TABLE, and CREATE VIEW system privileges, as well as SELECT privilege on the master tables.
- To create a snapshot in another user's schema, you must have the CREATE ANY SNAPSHOT system privilege, as well as SELECT privilege on the master table. Additionally, the owner of the snapshot must have been able to create the snapshot.

Managing Snapshots

This section describes how to manage read–only snapshots.

Operations on a Master Table that Affect Snapshots

All changes made by INSERT, UPDATE, and DELETE statements issued against a table are reflected in associated snapshots when the snapshots are refreshed.

TRUNCATE automatically forces all snapshots of the truncated table to be completely refreshed during their next refresh.

If you drop a master table, any associated snapshots remain and continue to be accessible. An associated snapshot log (if present) of a dropped master table is also dropped. When you attempt to refresh a snapshot based on a non–existent master table, Oracle returns an error.

If you later re–create the master table, the snapshot can again be successfully refreshed, as long as the defining query of the snapshot can be successfully issued against the new master table. You cannot perform a fast refresh of the snapshot, however, until after you re–create the snapshot log. If you cannot successfully refresh the snapshot after dropping and re–creating the master table, you should drop and re–create the snapshot.

Snapshots and Media Failure

As the result of a media failure, either a database that contains a master table of a snapshot or a database with a snapshot may need to be recovered. If a master database is independently recovered to a past point in time (that is, coordinated time–based distributed database recovery is not performed), any dependent remote snapshot that refreshed in the interval of lost time will be inconsistent with its master table. In this case, the administrator of the master database should instruct the remote administrator to perform a complete refresh of any inconsistent snapshot. For additional information on recovering from media failure, refer to your *Oracle7 Server Administrator's Guide*.

Indexing Snapshots

To increase the query performance when using a snapshot, you can create indexes for the snapshot. To index a column (or columns) of a snapshot, you must create the index on the underlying "SNAP$_" table created to hold the rows of the snapshot.

☞ **Attention:** Do not use declarative constraints to create an index; instead, use the CREATE INDEX statement (but do not use CREATE UNIQUE INDEX).

Setting Storage Parameters for Snapshots

How you should set storage options for a snapshot depends on the type of snapshot (simple or complex):

- In general, a simple snapshot's storage options should mimic the storage options for its master table, since they share the same characteristics. If a number of master tables are clustered in the master database, you should probably cluster the corresponding snapshots in the remote database.

  Note: If a simple snapshot does not duplicate all columns of its master table, modify the snapshot storage items accordingly.

- Because a complex snapshot is always completely refreshed, set its PCTFREE to 0 and PCTUSED to 100 for maximum efficiency.

You can change a snapshot's storage parameters using the ALTER SNAPSHOT command. For example, the following command alters the EMP snapshot's PCTFREE parameter:

```
ALTER SNAPSHOT emp PCTFREE 10;
```

You cannot change a snapshot's defining query; you must drop the snapshot and then re–create it.

Privileges Required to Alter a Snapshot

To alter a snapshot's storage parameters, the snapshot must be contained in your schema or you must have the ALTER ANY SNAPSHOT and ALTER ANY TABLE system privileges.

Dropping Snapshots

You can drop a snapshot independently of its master tables or the snapshot log. To drop a local snapshot, use the SQL command DROP SNAPSHOT. For example:

```
DROP SNAPSHOT emp;
```

If you drop the only snapshot of a master table, you should also drop the snapshot log of the master table, if there is one.

Privileges Required to Drop a Snapshot

Only the owner of a snapshot or a user with the DROP ANY SNAPSHOT system privilege can drop a snapshot.

Using Snapshots

Snapshots are queried just like a table or view. For example, the following statement queries a snapshot named EMP:

```
SELECT * FROM emp;
```

☞ **Attention:** Never manipulate data in the base table of a read–only snapshot.

You cannot issue any INSERT, UPDATE, or DELETE statements when using a read–only snapshot; if you do, an error is returned. Although INSERT, UPDATE, and DELETE statements can be issued against the base table for a snapshot, they can corrupt the snapshot. Updates are allowed on the master table only, which must then be refreshed to update the snapshot. If you want to alter the snapshot, you must create it as an updatable snapshot as described in Chapter 5.

Creating Views and Synonyms Based on Snapshots

Views or synonyms can be defined based on snapshots. The following statement creates a view based on the EMP snapshot:

```
CREATE VIEW sales_dept AS
   SELECT ename, empno
   FROM emp
   WHERE deptno = 10;
```

Privileges Required to Use a Snapshot

To query a snapshot, you must have the SELECT object privilege for the snapshot, either explicitly or via a role.

Managing Snapshot Logs

A snapshot log is a table, in the same database as the master table for a snapshot, that is associated with the master table. Its rows list changes that have been made to the master table, and information about which snapshots have and have not been updated to reflect those changes. You can create a snapshot log to decrease the amount of processing and time needed to refresh the simple snapshot.

☞ **Attention:** Snapshot logs cannot be used with complex snapshots.

A snapshot log is associated with a single master table; likewise, a master table can have only one snapshot log. If multiple simple snapshots are based on the same master table, they all use the same snapshot log.

These sections explain how to create, manage, and drop snapshot logs.

Creation Order of a Simple Snapshot and the Snapshot Log

If you are creating a simple snapshot, it is more efficient to create the snapshot log before the snapshot. Figure 3 – 3 illustrates the two orders of creation.

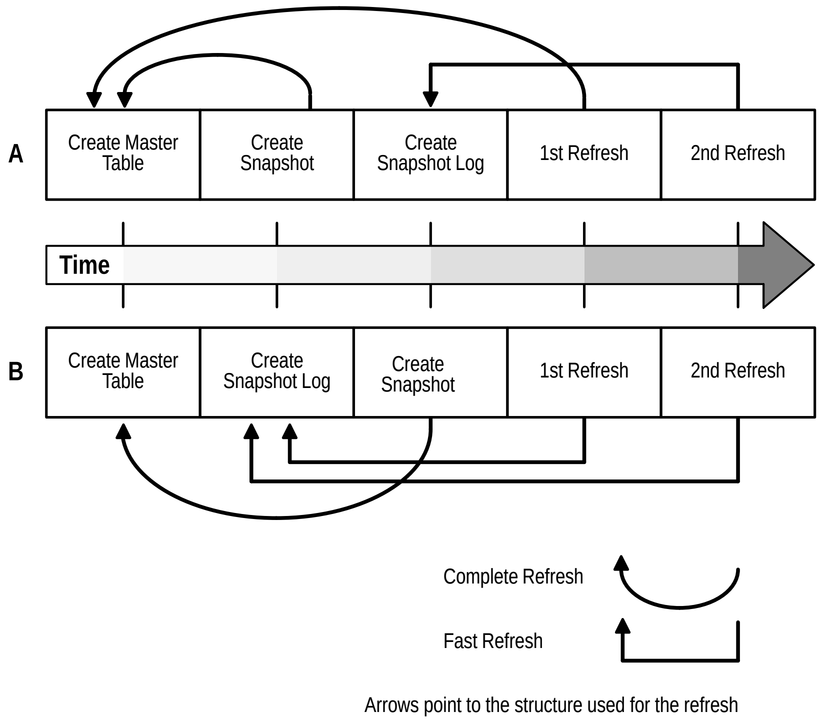


Figure 3 – 3 Creation Order of a Simple Snapshot and the Snapshot Log

In Method A, the first refresh of the snapshot cannot use the log because the log cannot reflect all updates entered between the creation of the snapshot and the creation of the snapshot log; therefore, two complete refreshes are necessary.

In contrast, Method B only requires one complete refresh (when creating the snapshot); subsequent refreshes can immediately use the snapshot log. If the master table is large or a number of simple snapshots are based on the same master table, creating the snapshot log before the snapshots can be much more efficient.

Creating a Snapshot Log

Create a snapshot log in the same database as the master table using the SQL command CREATE SNAPSHOT LOG. You can set storage options for the snapshot log's data blocks, extent sizes and allocation, and tablespace to hold the snapshot log. For example, the following statement creates a snapshot log associated with the EMP table:

```
CREATE SNAPSHOT LOG ON scott.emp
    TABLESPACE users
    STORAGE (INITIAL 10K NEXT 10K PCTINCREASE 50)
    PCTFREE 5;
```

Naming Snapshot Logs

Oracle automatically creates the snapshot log in the schema that contains the master table. Since you cannot specify a name for the snapshot log (one is implicitly given by Oracle), uniqueness is not a concern.

The Internals of Snapshot Log Creation

When you create a snapshot log, Oracle performs several operations internally:

- Oracle creates a table, named MLOG$_*master_table_name,* to store the ROWID and timestamp of rows updated in the master table. The timestamp column is not updated until the log is first used by a snapshot refresh.
- Oracle creates an AFTER ROW trigger on the master table to insert the ROWIDs and timestamps of inserted, updated, and deleted rows into the master snapshot log. The trigger is named TLOG$_*master_table_name.*

Additional Information: Refer to *Oracle7 Server Application Developer's Guide* to learn more about triggers.

The underlying table for a snapshot log and associated trigger are contained in the same schema as the master table. For both the table and the log, the *master_table_name* is truncated at 20 bytes (if necessary) and appended with a four–digit number to ensure uniqueness. This guarantees that the objects comply with the naming rules for schema objects.

Attention: Do not alter or change data in these objects.

Privileges Required to Create Snapshot Logs

If you own the master table, you can create an associated snapshot log if you have the CREATE TABLE and CREATE TRIGGER system privileges. If you are creating a snapshot log for a table in another user's schema, you must have the CREATE ANY TABLE and CREATE ANY TRIGGER system privileges. In either case, the owner of the snapshot log must have sufficient quota in the tablespace intended to hold the snapshot log.

The privileges required to create a snapshot log directly relate to the privileges necessary to create the underlying objects associated with a snapshot log.

Setting Storage Parameters for Snapshot Logs

Set a snapshot log's storage options as follows:

- Set PCTFREE to 0, and PCTUSED to 100.
- Set extent storage parameters according to the update activity (number of INSERT, UPDATE, and DELETE statements) on the master table.

Each row in a snapshot log takes approximately 26 bytes (18 bytes for an Oracle ROWID, seven bytes for a timestamp, and one byte for DML type). Use this number to calculate how much space a snapshot log requires, using the procedure described for calculating space required by non–clustered tables in the *Oracle7 Server Administrator's Guide*.

Altering Snapshot Log Storage Parameters

You can alter a snapshot log's storage parameters using the SQL command ALTER SNAPSHOT LOG. For example:

```
ALTER SNAPSHOT LOG sale_price
   PCTFREE 25
   PCTUSED 40;
```

Privileges Required to Alter Snapshot Logs Only the owner of the master table, or a user with the ALTER ANY TABLE system privilege can alter the storage parameters of a snapshot log.

Managing Snapshot Log Space Use

Oracle automatically tracks which rows in a snapshot log have been used during the refreshes of snapshots, and purges these rows from the log so that the log does not grow endlessly. Because multiple simple snapshots can use the same snapshot log, rows used in the refresh of one snapshot may still be needed to refresh another snapshot; Oracle does not delete rows from the log until all snapshots have used them. However, this automated feature can cause a snapshot log to grow indefinitely if a single associated snapshot is never refreshed.

For example, Snapshot EMP_B is regularly refreshed. However, Oracle cannot purge the rows used during the refresh of Snapshot EMP_B because Snapshot EMP_A needs them for its next refresh. This situation occurs when you have several simple snapshots based on the same master table and

- One snapshot is not configured to be automatically refreshed by Oracle; it has to be manually refreshed.
- One snapshot has a large refresh interval, such as every year.

- A network failure has prevented an automatic refresh of one or more of the snapshots based on the master table.
- A network or site failure has prevented a dropped snapshot from unregistering itself from its master.

Purging the Snapshot Log

Keep the snapshot log as small as possible to minimize the space it uses. To reduce the number of rows in a snapshot log, you can either refresh the snapshots associated with the log, or shrink the log by deleting the rows only required by the Nth least recently refreshed snapshots. To do the latter, execute the PURGE_LOG stored procedure of the DBMS_SNAPSHOT package. For example, to purge the log entries in the example above that are needed only for snapshot EMP_A, the least recently refreshed snapshot, you could execute the following procedure:

```
DBMS_SNAPSHOT.PURGE_LOG( master => emp,
                         num    => 1);
```

Additional Information: The parameters for the PURGE_LOG procedure are described in Table 12 – 187.

Reducing Space Allocation for a Snapshot Log

If a snapshot log grows and allocates many extents, purging the log of rows does not reduce the amount of space allocated for the log. To reduce the space allocated for a snapshot log, first copy the rows in the snapshot log to a new location. Then truncate the log and reinsert the old rows. This avoids having to perform a complete refresh of the dependent snapshots. However, any changes made to the master table between the time that you copy the rows to a new location and when you truncate the log will be lost, until the next complete refresh.

Privileges Required to Delete Rows from a Snapshot Log The owner of a snapshot log or a user with the DELETE ANY TABLE system privilege can purge rows from the snapshot log by executing the PURGE_LOG procedure.

Dropping Snapshot Logs

You can drop a snapshot log independently of its master table or any existing snapshots. You might decide to drop a snapshot log if one of the following is true:

- All simple snapshots of a master table have been dropped.
- All simple snapshots of a master table are to be completely refreshed, not fast refreshed.

To drop a local snapshot log, use the SQL command DROP SNAPSHOT LOG, as in

```
DROP SNAPSHOT LOG emp_log;
```

Privileges Required to Drop a Snapshot Log

Only the owner of the master table, or a user with the DROP ANY TABLE system privilege can drop a snapshot log.

Using Snapshot Refresh Groups

This section describes the procedures provided in the DBMS_REFRESH package that allow you to create, alter, and delete refresh groups. For more information on automatically refreshing snapshots, see page 3 – 16. These procedures should be called from the site where your snapshots are located (not from the site of their associated master tables).

Creating a Refresh Group

To specify the members of a refresh group and the time interval used to determine when the members of this group should be refreshed, call the MAKE procedure of the DBMS_REFRESH package, as shown in the following example:

```
DBMS_REFERESH.MAKE(
    name             => 'acctg',
    list             => 'acct_rec, acct_pay',
    next_date        => SYSDATE,
    interval         => 'SYSDATE + 1/24',
    implicit_destroy => TRUE);
```

This example creates the ACCTG refresh group with two members, ACCT_REC and ACCT_PAY, that will be refreshed every hour. Additional information on setting the refresh interval is provided on page 3 – 16.

The type of refresh performed is determined by the mode that you specified when you created each snapshot. If you did not specify a refresh mode for a snapshot, Oracle performs a fast refresh if possible; otherwise, it performs a complete refresh.

> **Additional Information:** The parameters for the MAKE procedure are described in Table 12 – 59.

Altering a Refresh Group

The DBMS_REFRESH package contains separate procedures for adding new members to a refresh group, removing members from a refresh group, and altering the automatic refresh interval for a refresh group.

Adding Members to a Refresh Group

To add snapshots to a refresh group, call the ADD procedure in the DBMS_REFRESH package, as shown in the following example:

```
DBMS_REFRESH.ADD( name => 'acctg',
                  list => 'acct_bill',
                  lax  => TRUE);
```

This example adds the ACCT_BILL snapshot to the ACCTG refresh group. Setting the last argument to TRUE lets ACCT_BILL be added to ACCTG even if it must first be removed from another group.

Additional Information: The parameters for the ADD procedure are described in Table 12 – 56.

Removing Members from a Refresh Group

To remove snapshots from a refresh group, call the SUBTRACT procedure in the DBMS_REFRESH package, as shown in the following example:

```
DBMS_REFRESH.SUBTRACT( name => 'acctg',
                       list => 'acct_bill');
```

This example removes the ACCT_BILL snapshot from the ACCTG refresh group. This snapshot will no longer be automatically refreshed. After removing a snapshot from a refresh group, you should either refresh it manually or add it to another refresh group.

If you set IMPLICIT_DESTROY to TRUE in the MAKE call for the refresh group, Oracle automatically deletes the group whenever you subtract the last member.

Additional Information: The parameters for the SUBTRACT procedure are described in Table 12 – 61.

Altering the Refresh Interval

If you want to change how often a snapshot group is refreshed (for example, if you want the snapshots refreshed once a day as opposed to once a week), call the CHANGE procedure in the DBMS_REFRESH package as shown in the following example:

```
DBMS_REFRESH.CHANGE( name      => 'acctg',
                     next_date => SYSDATE,
                     interval  => 'SYSDATE + 1');
```

This example changes the refresh interval of the ACCTG group to once a day.

Additional Information: The parameters for the CHANGE procedure are described in Table 12 – 57.

Deleting a Refresh Group

If you want to remove all of the snapshots from a refresh group and delete the refresh group, call the DESTROY procedure in the DBMS_REFRESH package, as shown in the following example:

```
DBMS_REFRESH.DESTROY( name => 'acctg');
```

This example removes all of the snapshots from the ACCTG refresh group. These snapshots will no longer be automatically refreshed. You must either refresh these snapshots manually, or add the snapshots to new refresh groups.

Additional Information: The parameters for the DESTROY procedure are described in Table 12 – 58.

Refreshing Snapshots

You can refresh a snapshot to make the snapshot reflect a more recent state of its master tables. There are two types of refreshes: a fast refresh and a complete refresh. A fast refresh uses the snapshot log of a master table to refresh a simple snapshot by transmitting only the changes needed to bring the snapshot up to date. Only simple snapshots (that is, those consisting of a subset of rows and columns of a single table) can execute a fast refresh. A complete refresh entirely replaces the existing data in a simple or complex snapshot. Also, snapshots can be refreshed automatically or manually, either individually or in groups.

Consider the following issues when deciding how to refresh a snapshot:

- Generally a simple snapshot should use fast refreshes because they are more efficient than complete refreshes.
- If the master tables receive predictable updates, automatically refresh the associated snapshots at the appropriate interval.
- After bulk loads to the master tables, manually refresh all snapshots based on the master tables. This propagates the new rows of the master tables to associated snapshots.
- If you need to refresh a collection of snapshots to a single point in time, such as when there is a parent/child relationship between a pair of snapshots, use snapshot refresh groups.

Privileges Required to Refresh a Snapshot

To refresh a snapshot, you must meet the following criteria:

- You must own the snapshot or have the ALTER ANY SNAPSHOT system privilege.
- The snapshot owner (or the user that you have connected as, if you are using a database link) must have SELECT privileges on the master table and, for fast refreshes, on the snapshot log.

Automatically Refreshing Snapshots

If you want to have your snapshots automatically refreshed at a periodic interval, you must complete the following steps:

- You must specify a snapshot refresh interval and type of refresh.
- You must have one or more SNP background processes that wake up periodically and refresh any snapshots that are due to be refreshed.

Specifying a Snapshot Refresh Interval

If you want to refresh an individual snapshot automatically, specify a refresh interval using the START WITH and NEXT parameters in the REFRESH clause of a CREATE SNAPSHOT or ALTER SNAPSHOT statement. This automatically creates a snapshot refresh group that consists of exactly one snapshot. This refresh group has the same name as the snapshot itself.

If you want to refresh a collection of snapshots to a single point of time automatically, you must create a snapshot refresh group using the DBMS_REFRESH.MAKE procedure. To refresh a snapshot refresh group automatically, supply NEXT_DATE and INTERVAL values when you create the group.

Specifying the Refresh Type

When you refresh a snapshot, you can specify that Oracle perform a FAST, COMPLETE, or FORCEd refresh. You can specify one of these three refresh types in the REFRESH clause of a CREATE SNAPSHOT or ALTER SNAPSHOT statement. The snapshots in a refresh group do not have to have the same refresh type. If you do not specify a refresh type, Oracle performs a FORCEd refresh. FORCE performs a fast refresh if possible, or a complete refresh otherwise.

Starting a Background Process

The snapshot refresh facility works by using job queues to schedule the periodic execution of the DBMS_REFRESH.REFRESH procedure. Job queues require that at least one SNP background process be running. This background process wakes up periodically, checks the job queue, and executes any outstanding jobs. The SNP background processes are controlled by the initialization parameters JOB_QUEUE_PROCESSES and JOB_QUEUE_INTERVAL. For more information on job queues, and the initialization parameters that you must set, see Chapter 10.

Understanding the Snapshot Refresh Interval

When setting a snapshot's refresh interval, understand the following behavior:

- The dates or date expressions in the START WITH and NEXT parameters of an individual snapshot, or INTERVAL and NEXT_DATE parameters in the procedure calls for a refresh group, must evaluate to a future point in time. The INTERVAL value is evaluated immediately before the refresh begins.

Thus, you should select an interval that is greater than the length of time required to perform a refresh. A date literal must be enclosed in single quotes, while date expressions do not require quotes.

- If a snapshot should be refreshed periodically at a set interval, use the NEXT or INTERVAL parameter with a date expression similar to “SYSDATE + 7”. For example, if you set the automatic refresh interval to “SYSDATE + 7” on Monday, but for some reason, such as network failure, the snapshot is not refreshed until Thursday, “SYSDATE + 7” now evaluates to every Thursday, not Monday. If you always want to refresh a snapshot group at a specific time automatically, regardless of the last refresh (for example, every Monday), the INTERVAL or NEXT parameters should specify a date expression similar to “NEXT_DAY(TRUNC(SYSDATE), ’MONDAY’)”.

Table 3 – 1 lists some common date expressions used for snapshot refresh intervals.

Date Expression	Evaluation
`SYSDATE + 7`	Exactly seven days from the last refresh
`SYSDATE + 1/48`	Every half hour
`NEXT_DAY(TRUNC(SYSDATE), ’MONDAY’)+3/24`	Every Monday at 3 PM
`NEXT_DAY(ADD_MONTHS(TRUNC(SYSDATE,’Q’),3), ’THURSDAY’)`	The first Thursday of each quarter

Table 3 – 1 Examples of Common Refresh Intervals

Examples The following example shows valid combinations for specifying a refresh interval:

```
CREATE SNAPSHOT snap
   . . .
   REFRESH COMPLETE
           START WITH ’01–JUN–94’
           NEXT sysdate + 7
   AS . . . ;
```

This statement creates the SNAP snapshot, specifying complete automatic refreshes, the first of which occurs on June 1, 1994, with an automatic refresh interval of seven days from the most recent refresh.

The following command creates the ACCT refresh group, which is composed of three snapshots that will be refreshed every Monday:

```
dbms_refresh.make(
      name             => ’acct’,
      list             => ’scott.acct, scott.finance, scott.inventory’,
      next_date        => SYSDATE,
      interval         => ’next_day(SYSDATE + 1, ’’MONDAY’’)’,
      implicit_destroy => TRUE,
      lax              => TRUE);
```

☞ **Attention:** Note that because the interval function evaluates to a literal, it must be enclosed in single quotes. Because you must use two single quotes to represent one single quote within a literal, in this example, the literal MONDAY is enclosed in two sets of single quotes.

Troubleshooting Automatic Refresh Problems

Several factors can prevent the automatic refresh of snapshots: not having an SNP background process, an intervening network or instance failure, or an instance shutdown. You may also encounter an error if you attempt to define a master detail relationship between two snapshots. You should define master detail relationships only on the master tables by using declarative referential integrity constraints; the related snapshots should then be placed in the same refresh group to preserve this relationship. Although not prevented by Oracle, if you attempt to define these constraints at the snapshot level, when refreshed, the snapshots may temporarily enter a state where they violate the integrity constraints that you have defined, producing a runtime error.

When any of the factors described above prevents the automatic refresh of a snapshot group, the group remains due to be refreshed. (Remember, when you specify a refresh interval for an individual snapshot, Oracle automatically creates a refresh group for that snapshot.) See page 9 – 8 for additional troubleshooting information.

Snapshots Failing to Refresh

If Oracle encounters a failure, such as a network failure, when attempting to refresh a snapshot refresh group it attempts the refresh again. The first attempt is made after one minute, the second attempt after two minutes, the third after four minutes, and so on, with the interval doubling between each attempt. When the retry interval exceeds the refresh interval, Oracle continues to retry the refresh at the normal refresh interval.

Thus, snapshot refresh groups due to be refreshed will generally be refreshed automatically shortly after you start an SNP background process or resolve any network failures.

However, if Oracle continues to encounter errors when attempting to refresh a snapshot, it considers the group broken after its sixteenth unsuccessful attempt.

Oracle indicates that a snapshot is broken by setting the BROKEN column of the USER_REFRESH and USER_REFRESH_CHILDREN views to Y.

The errors causing Oracle to consider a snapshot refresh group broken are recorded in a trace file. After you have corrected these errors, you must manually refresh the group by calling the procedure DBMS_REFRESH.REFRESH described on 3 – 19. This resets the broken flag to N, and automatic refreshes will proceed from this point.

Snapshots Continually Refreshing

If you encounter a situation where your snapshots are being continually refreshed, you should check the refresh interval that you specified. This interval is evaluated before the snapshot is refreshed. If the interval that you specify is less than the amount of time it takes to refresh the snapshot, the snapshot will be refreshed each time the SNP background process checks the queue of outstanding jobs.

OSDoc

Additional Information: The name of the snapshot trace file is of the form SNPn, where n is platform specific. Consult your platform–specific Oracle documentation for the name on your system.

Snapshot Logs Growing Without Bounds

If a snapshot log is growing without bounds check for snapshots that were dropped but remain registered, You may need to purge part of the log by calling DBMS_SNAPSHOT.PURGE_LOG, as described on page 3 – 12.

Manually Refreshing Snapshots

If you want to manually force a refresh to occur, you have two options.

- You can manually refresh an existing snapshot refresh group.
- You can manually refresh one or more snapshots, that may, or may not, be part of one or more refresh groups.

Manually Refreshing a Snapshot Refresh Group

To refresh a refresh group manually, call the REFRESH procedure in the DBMS_REFRESH package, as shown in the following example:

```
DBMS_REFRESH.REFRESH('acctg');
```

This example causes the refresh of the ACCTG group to occur immediately instead of waiting for the next automatic refresh interval. Manually refreshing a refresh group does not affect the next automatic refresh of the group.

Additional Information: The parameter for the REFRESH procedure is described in Table 12 – 60.

Manually Refreshing One or More Snapshots

To consistently refresh one or more snapshots that are not members of the same refresh group, use the REFRESH procedure in the package DBMS_SNAPSHOT.

This command allows you to provide a comma–separated list of snapshots that you want to refresh to a transaction–consistent point in time. These snapshots can belong to other refresh groups. Refreshing them using this procedure will not affect their regularly scheduled refresh interval if they are part of an automatic snapshot refresh group. However, if they are members of other groups, you should be aware that those groups will no longer be transactionally consistent.

The following example performs a complete refresh of SCOTT.EMP, a fast refresh of SCOTT.DEPT, and a default refresh of SCOTT.SALARY. By default, DBMS_SNAPSHOT.REFRESH refreshes the snapshot using the mode specified when the snapshot was created. If no mode was specified, the default mode is FORCE.

```
DBMS_SNAPSHOT.REFRESH( list   => 'scott.emp, scott.dept, scott.salary',
                       method => 'CF');
```

Additional Information: The parameters for the REFRESH procedure are described in Table 12 – 188.

Snapshot Refresh and Remote Databases

You must fully qualify remote database names when issuing a CREATE SNAPSHOT statement or snapshot refreshes may fail due to naming and privilege conflicts. For example, if you created a snapshot using the following commands:

```
CONNECT scott/tiger
CREATE DATABASE LINK sales.hq.com USING 'hq.sales.com';
CREATE SNAPSHOT mysnap AS SELECT * FROM emp@sales.hq.com;
```

then refreshes of the snapshot will fail if the user is not SCOTT.

Instead, you should replace the above CREATE SNAPSHOT statement with the following:

```
CREATE SNAPSHOT mysnap AS SELECT * FROM scott.emp@sales.hq.com;
```

Your manual refreshes should then succeed. Automatic refreshes, however, will only work if the username and password strings are embedded in the database link, as shown in the following statement:

```
CREATE DATABASE LINK sales.hq.com CONNECT TO scott
     IDENTIFIED BY tiger USING 'sales.hq.com';
```

The username and password that you specify allow the SNP background process to connect to the database. For the refresh to succeed, this user must have SELECT privileges on the master table and, for fast refreshes, the snapshot log.

Listing Snapshot Information

Query the DBA_SNAPSHOTS catalog view to obtain a listing of all of the snapshots in a database. Query DBA_SNAPSHOT_LOGS for information about logs for your database.

Listing Refresh Group Information

The data dictionary contains the following views that have information about refresh groups:

- USER_REFRESH
- USER_REFRESH_CHILDREN

CHAPTER

4

Multi–Master Replication

This chapter explains how to use the symmetric replication facility to create and maintain replicated master sites. The topics discussed include the following:

- database links and security in a replicated environment
- creating the master definition site
- creating replicated objects
- generating replication support
- creating additional master sites
- propagating data–level changes between master sites
- propagating schema–level changes between master sites
- quiescing a replicated environment
- altering a replicated object
- removing a master site

Note that most of the activities described in this chapter can be accomplished much more easily by using Oracle's Replication Manager, a GUI interface for replication. See the documentation for Oracle Replication Manager for more information.

OSDoc

Additional Information: The symmetric replication facility is available to users of the advanced replication option only. The advanced replication option is separately installable, and may require additional system resources. Consult your operating system–specific Oracle installation instructions for more information.

Replication Support for Master Sites

The symmetric replication facility supports the replication of tables, views, indexes, synonyms, triggers, and packages at master sites. All master sites participating in a replicated environment must have the same replicated objects. If you only want to replicate a subset of the objects to a given site, you should consider designating that site as a snapshot site.

Replication Support for Tables

The symmetric replication facility supports replication of both data–level (DML) and schema–level (DDL) changes to replicated tables. Whenever you perform an INSERT, UPDATE, or DELETE on a table replicated using row–level replication, Oracle ensures that the change is ultimately applied at each replica of the table. Whenever you update a table by calling a replicated procedure, your procedure call is ultimately replicated to all other sites in the replicated environment. Propagation of these changes between master sites is explained in detail starting on page 4 – 26.

The symmetric replication facility supports the following data types: NUMBER,DATE, VARCHAR2, CHAR, ROWID, and RAW. Your table can include columns with other data types (e.g. LONG and LONG RAW, but these columns will not be populated by the symmetric replication facility.

Whenever you create, alter, or drop a table using the procedures provided in the DBMS_REPCAT package, that change is ultimately applied at all other master sites in the replicated environment, by the replication facility. Propagation of these changes between master sites is explained in detail starting on page 4 – 35.

Replication Support for Non–table Objects

Any changes that you make to a non–table object using the procedures provided in the DBMS_REPCAT package are ultimately replicated to all other sites in the replicated environment. You should always use the procedures provided in the DBMS_REPCAT package to update non–table replicated objects.

For example, suppose that you choose to replace a stored procedure at a master site. If that procedure is used to update a table that is replicated using row–level replication, any changes made through the procedure will continue to be properly forwarded to the other replicated sites.

If you are using procedural replication, however, you must only update this procedure using the method for altering a replicated object described on page 4 – 45. This ensures that the proper replication support for this procedure is generated at all replication sites.

Designing Your Replicated Environment

When you begin to design your replicated environment, you need to make the following decisions:

- You should first determine which objects you want to replicate. Objects must be part of a replicated object group. A replicated object group can include objects from multiple schemas, but each object can only belong to one object group.
- Next, you should determine where you want these objects replicated. You also need to determine which sites will be master sites, which must contain all replicated objects, and which will be snapshot sites, which can contain a subset of the replicated objects. A typical replicated environment may consist of two master sites each supporting several snapshot sites.
- You must also determine how you want Oracle to propagate your changes among the replication sites — synchronously or asynchronously. If you choose to propagate your changes asynchronously, you must also determine how frequently you want to propagate your changes.
- Finally, you must choose to either partition ownership of your data and avoid update conflicts, or you must select a method of resolving these conflicts, should they occur. If all of your master and snapshot sites are propagating their changes synchronously, you need not worry about conflicts.

Before Creating a Replicated Environment

After you have determined which objects you want to replicate, you must ensure that you have the privileges necessary to create these objects at each site. Additionally, once you have determined which sites will make up your replicated environment, you must ensure that these sites can communicate with one another by creating the necessary database links.

There are three categories of users in a replicated environment:

replication administrators	Users who are responsible for configuring and maintaining a replicated environment.
symmetric replication facility	Certain replication activities are run as SYS and must execute on remote nodes.
end users	Users querying and updating replicated objects.

Replication Administrators

A *replication administrator* is responsible for configuring the replicated environment. You can create one replication administrator to administer all replicated objects at a site, or you can have separate administrators for the replicated objects in each schema. All symmetric replication procedures should be run while you are connected as a replication administrator.

> **Note:** The packages DBMS_REPCAT_ADMIN and DBMS_REPCAT_AUTH are owned by SYS. Access to these packages should not be granted widely.

Granting Privileges

To create a replication administrator for a single schema, call DBMS_REPCAT_ADMIN.GRANT_ADMIN_REPGROUP, as shown in the following example:

```
DBMS_REPCAT_ADMIN.GRANT_ADMIN_REPGROUP(userid => 'acctng');
```

In this example, the ACCTNG user now has the privileges needed to administer any replicated objects in the ACCTNG schema at the current site. The ACCTNG user must already exist, before you call GRANT_ADMIN_REPGROUP. You can now GRANT or REVOKE privileges for this user as needed. This procedure is most useful if your replicated object group does not span schemas.

> **Additional Information:** The parameter for the GRANT_ADMIN_REPGROUP procedure is described in Table 12 – 177, and the exceptions are listed in Table 12 – 178.

To create a replication administrator for all replicated groups at your current site (also known as a *global replication administrator*), complete the following steps:

1. Create the replication administrator account, as shown in the following example:

   ```
   CREATE USER repadmin IDENTIFIED BY repadminpassword;
   ```

2. Grant the replication administrator the necessary privileges to administer all replicated objects at the current site by calling DBMS_REPCAT_ADMIN.GRANT_ADMIN_ANY_REPGROUP as shown in the following example:

```
DBMS_REPCAT_ADMIN.GRANT_ADMIN_ANY_REPGROUP(userid => 'repadmin');
```

3. You can now GRANT additional privileges to the replication administrator as necessary.

Additional Information: The parameter for the GRANT_ADMIN_ANY_REPGROUP procedure is described in Table 12 – 175, and the exceptions are listed in Table 12 – 176.

Revoking Privileges Call the REVOKE_ADMIN_REPGROUP or REVOKE_ADMIN_ANY_REPGROUP procedures in the DBMS_REPCAT_ADMIN package to revoke the privileges and roles granted above.

☞ **Attention:** Calling the REVOKE_*privs* procedures will revoke all privileges granted with the associated GRANT_*privs* procedures, even if identical privileges were granted through additional channels.

Additional Information: The parameter for the REVOKE_ADMIN_REPGROUP procedure is described in Table 12 – 181, and the exceptions are listed in Table 12 – 182. The parameter for the REVOKE_ADMIN_ANY_REPGROUP procedure is described in Table 12 – 179, and the exceptions are listed in Table 12 – 180.

Creating Database Links

To create the necessary links to perform replication administration, you must complete the following steps at each master site in the replicated environment:

- Connect to the site as the replication administrator.
- Ensure that the initialization parameter GLOBAL_NAMES is set to TRUE.

- Create a private database link to every other master site in the replicated environment. That is, if the schema is replicated on N masters, you must create N–1 links. Each database link must include the fully qualified database name of the remote master site and the username and password of the replication administrator at that site.

In the following example, assume that you have three master sites whose global database names are sales.widgetek.com, inventory.widgetek.com, and hq.widgetek.com. Now assume that you have created user REPADMIN at each site with password SECRET, and that this user will act as the replication administrator for this replicated environment. At the SALES site, you would need to create the database links as shown below:

```
CONNECT repadmin/secret
CREATE PUBLIC DATABASE LINK inventory.widgetek.com
    USING 'inventory.widgetek.com';
CREATE DATABASE LINK inventory.widgetek.com CONNECT TO repadmin
    IDENTIFIED BY secret;
CREATE PUBLIC DATABASE LINK hq.widgetek.com
    USING 'hq.widgetek.com';
CREATE DATABASE LINK hq.widgetek.com CONNECT TO repadmin
    IDENTIFIED BY secret;
```

Using a public link to provide the database specification of the remote database eliminates the need to include a USING clause in each of your private database links. To determine the global database name of a remote database, you can query the GLOBAL_NAME data dictionary view.

Replication Administration Usage Notes

When you call a procedure in the DBMS_REPCAT package at a master site, the symmetric replication facility attempts to perform any deferred replication administration for any replicated schema that is passed as an argument to the DBMS_REPCAT procedure.

All calls to procedures in the DBMS_REPCAT package for a given replicated object group should be performed serially. That is, only one person should be adding to, or altering a replicated environment at a time, or your environment may become out of sync. This includes activities that you may have set up to be performed by a background process.

To be safe, you should disable any DBMS_REPCAT.* jobs in the local job queue whenever you are administering a replicated environment.

You can use the USER_JOBS view to determine which jobs in the queue are associated with a given replicated object group. Then call DBMS_JOB.BROKEN to disable these jobs temporarily.

After completing your administrative activities, you can call this procedure again to re–enable the jobs.

If your object group spans schemas you will probably find it easiest to designate one person as the replication administrator for all replicated objects. If your object groups do not span schemas, you can designate one person as the replication administrator for each schema containing replicated objects. This person would be responsible for all calls made using the DBMS_REPCAT package to administer an object group at all master sites in the replicated environment. Different users can administer different schemas.

Because the DBMS_REPCAT package does not provide any additional access control, EXECUTE privileges on this package should not be granted widely.

A user granted EXECUTE privileges on DBMS_REPCAT does not gain any privileges on non–replication catalog views, but can modify the replication catalog views and disrupt a replicated environment. If you desire more flexibility, you should create a cover package for DBMS_REPCAT that provides the appropriate level of access control, and then grant wider access only on this cover package.

Note that granting EXECUTE on DBMS_REPCAT to a user does not give that user any greater privileges on the replicated objects. You must call the appropriate DBMS_REPCAT_ADMIN procedure to ensure that your replication administrator has the necessary privileges on all objects at all sites to perform the necessary DDL and DML operations that are supported by the procedures in DBMS_REPCAT.

Symmetric Replication Facility

Certain replication activities are run as SYS and require access to remote nodes. By creating a surrogate replication administrator at the remote site, you preclude the need for a SYS to SYS database link between master sites. The *surrogate replication administrator* performs actions on behalf of the symmetric replication facility at the remote site.

Granting Privileges

To create a surrogate replication administrator at your current site, complete the following steps:

1. Create the surrogate replication administrator account, as shown in the following example:

   ```
   CREATE USER surrogate IDENTIFIED BY surrogate_password;
   ```

   You will probably find it easiest to create the same user as the surrogate replication administrator at all master sites in your replicated environment.

2. Grant the surrogate replication administrator the necessary privileges to act on behalf of the symmetric replication facility for all replicated object groups at the current site by calling DBMS_REPCAT_AUTH.GRANT_SURROGATE_REPCAT, as shown in the following example:

```
DBMS_REPCAT_AUTH.GRANT_SURROGATE_REPCAT(userid => 'surrogate');
```

3. You can now GRANT additional privileges to the surrogate replication administrator as necessary.

> **Additional Information:** The parameter for the GRANT_SURROGATE_REPCAT procedure is described in Table 12 – 183, and the exceptions are listed in Table 12 – 184.

Revoking Privileges Call the REVOKE_SURROGATE_REPCAT procedure in the DBMS_REPCAT_AUTH package to revoke the privileges and roles granted above.

> **Additional Information:** The parameter for the REVOKE_SURROGATE_REPCAT procedure is described in Table 12 – 185, and the exceptions are listed in Table 12 – 186.

Creating Database Links

You need to create the links used by the symmetric replication facility in a similar manner to those created for each replication administrator. The replication facility, however, requires special privileges at each site to perform required operations. Therefore, the user that you specify in the CONNECT TO clause must have these privileges. This special user is called the surrogate replication administrator.

To create the necessary links for the symmetric replication facility, you must complete the following steps at each master site:

- Connect to the local site as SYSOPER or SYSDBA.
- Create a private database link to every other remote master site in the replicated environment. That is, if the schema is replicated to N masters, you must create N–1 links. Each database link must include the fully qualified database name of the remote master site and the username and password of the surrogate replication administrator at that site. For example, to create a link to the HQ master site from the previous example, you could issue the following statement:

```
CREATE DATABASE LINK hq.widgetek.com CONNECT TO surrogate
    IDENTIFIED BY surrogate_password USING 'hq.widgetek.com';
```

End Users

End users require no additional privileges beyond those necessary to update the local objects. The ability to apply these changes at the replicated sites is determined by the links used to propagate the changes, and whether the changes are propagated synchronously or asynchronously.

Synchronous Propagation

When you update a table at a site that is propagating changes synchronously, Oracle fires a trigger that calls a package which in turn makes a remote procedure call to the site or sites that are receiving the change synchronously.

To call the remote procedure, Oracle must connect to the remote database using a database link. Because the generated trigger package at the local site has the same owner as the owner of the table being updated, Oracle first looks for a private database link for the table owner. If no such link exists, Oracle next looks for a private database link for the connected user. If Oracle does not find a private database link, it next looks for a public database link to the remote site.

The privilege domain at the remote site is determined by the connection at the remote site. Oracle connects to the remote site using the username and password specified in the CONNECT TO clause of the database link, if one was given; otherwise, Oracle attempts the connection using the username and password for the local user making the update. The user that Oracle ultimately connects as must have the EXECUTE privilege on the generated procedures for the object being updated for the update to complete successfully.

Asynchronous Propagation

The privilege domain of an asynchronously replicated transaction when executed at the remote site is determined by

- the value of the EXECUTE_AS_USER argument to the DBMS_DEFER_SYS.EXECUTE procedure
- the available database links to the remote site

When the replication administrator schedules the execution of the deferred transaction queue, he or she can require that the deferred transaction be pushed as the user who originally queued the transaction, or as the connected session user.

To push the deferred transaction to the remote location, Oracle must first establish a connection to the remote site. To determine which link to use, Oracle checks the EXECUTE_AS_USER setting. If this parameter is FALSE, the default, Oracle looks for a database link for the user who originated the transaction.

If this parameter is TRUE, Oracle looks for a database link for the connected session user. (Typically, the replication administrator pushes the deferred transaction queue.)

If the available link does not include a username and password, Oracle uses the username and password associated with the current session (regardless of the setting of EXECUTE_AS_USER). If the current session is a background process (such as when you schedule periodic execution of the deferred transaction queue), there is no available username and password, and the connection fails.

Once a connection is established, the privilege domain of the transaction at the remote site is determined by the privilege domain of the connection. For example, if the connection is made using the database link created for the replication administrator, the privilege domain of the replication administrator at the *remote* site determines whether the transaction can be successfully executed at the remote site. To execute the transaction successfully, this user (the remote replication administrator) must have the EXECUTE privilege on the generated procedures for the object being updated.

If all transactions will be pushed to the remote site as the replication administrator (EXECUTE_AS_USER = TRUE), no additional database links are necessary. To ensure that the replication administrator at the remote site has the necessary privileges to apply these transactions, you must either GRANT the replication administrator at each site the EXECUTE ANY PROCEDURE system privilege, or GRANT the replication administrator the EXECUTE object privilege on each of the generated replication procedures.

Note: When you replicate an object, its associated privileges are not replicated. To replicate the GRANT command to each site, you can use the DBMS_REPCAT.EXECUTE_DDL procedure described on page 7 – 14.

If transactions will be pushed to the remote site as the originating user (EXECUTE_AS_USER = FALSE), each user should have a private database link to each site participating in the replicated environment. This link should include a username and password. The user specified in the CONNECT TO clause must have the necessary EXECUTE privileges on the generated replication procedures.

For example, if user SCOTT at the New York site has UPDATE privileges on the ACCT.EMP and ACCT.DEPT tables, and you create the following private link while connected as SCOTT

```
CREATE DATABASE LINK hq.widgetek.com
    CONNECT TO scott IDENTIFIED BY tiger;
```

then user SCOTT at the remote HQ site must have EXECUTE privileges on the ACCT.EMP$RP and ACCT.DEPT$RP packages. Because the generated packages are considered to have the same owner as the owner of the table, the table owner automatically has the ability to execute the packages. All other users must either be granted the EXECUTE ANY PROCEDURE privilege at each site, or be granted EXECUTE on each generated package at each remote site for every table they may update.

Creating a Multi–Master Replicated Environment

To create a multi–master replicated environment, you (the replication administrator) must use the procedures in the DBMS_REPCAT package to complete the following steps:

- First, you must select which of the sites in your new multi–master replicated environment will be the master definition site.

  This site is used to create all other master sites for a given replicated object group. When selecting a master definition site, you should choose the site that is most likely to be available at all times. You can change master definition sites if necessary.

- Next, make sure that you have created all of the necessary links between the sites that you have selected to participate in your replicated environment. You must also have granted the necessary privileges to the replication administrator and surrogate replication administrator. Only the replication administrator can create a replicated environment.

- To create your replicated environment, you must begin by creating an empty master replicated object group at the site you have selected to be the master definition site. Create this replicated object group by calling DBMS_REPCAT.CREATE_MASTER_REPGROUP.

- For each object, such as a table or procedure, that you want to add to the replicated object group, you must perform the following steps:
  - First, add the object to the replicated object group by calling the CREATE_MASTER_REPOBJECT procedure.
  - If the object is a table that requires conflict resolution routines, call the appropriate ADD_*conflicttype*_RESOLUTION procedures. These procedure are described in Chapter 6.
  - For each object that requires replication support, call GENERATE_REPLICATION_SUPPORT to create the necessary triggers, packages, and procedures. The owner of the replicated object must have the EXECUTE privilege on the DBMS_DEFER package in order for Oracle to create these objects successfully.

  Note: If you will be adding additional master sites to your replicated environment, you may prefer to defer generating replication support until after these sites have been added. However, because the tables will be copied to the new master sites without the supporting triggers and packages any changes that occur before you generate replication support will not be replicated.

  You must either call GENERATE_REPLICATION_SUPPORT for each object before adding any master sites and again after adding any master sites, or you must ensure that no updates are made to the tables at any of the master sites until after you have added all master sites and then call GENERATE_REPLICATION_SUPPORT.
- For each additional master site that you want to include in your replicated environment, call ADD_MASTER_DATABASE.
- For sites in your replication environment that are propagating their changes asynchronously, you must schedule this propagation by calling DBMS_DEFER_SYS.SCHEDULE_EXECUTION.
- Because all activity at the master definition site was suspended when you called CREATE_MASTER_REPGROUP, you must now call RESUME_MASTER_ACTIVITY to restore normal activity.

☞ **Attention:** You must generate replication support before attempting to update any tables you have registered as replicated objects. If you are unsure whether replication support has been generated for a particular object, see the RepGenerated view (see page 13 – 6).

Creating a Replicated Object Group

Only objects in a replicated object group can be replicated to other master and snapshot sites. When you create a replicated object group, the name of the object group is added to the RepGroup view at that site. Additionally, the object group name and the global database name (and the connection qualifier, if any, associated with the object group for the local site are added to the RepSite view for the site, and by default, this site is designated as the master definition site.

To create a new, empty, quiesced master replicated object group, call DBMS_REPCAT.CREATE_MASTER_REPGROUP, as shown in the following example:

```
DBMS_REPCAT.CREATE_MASTER_REPGROUP(
    gname          => 'acct',
    group_comment => 'created by '||JSMITH||' on '||1–24–96,
    master_comment => 'created by '||JSMITH||' on '||1–24–96);
```

When you create a new replicated object group, you may choose to add a comment to the RepGroup or RepSite views. In this example, the comment indicates when the replicated object group was first created and by whom.

Additional Information: The parameters for the CREATE_MASTER_REPGROUP procedure are described in Table 12 – 100, and the exceptions are listed in Table 12 – 101.

Creating a Replicated Object

Only objects in a replicated object group can be replicated to another site. To add an object to a replicated object group, call the procedure CREATE_MASTER_REPOBJECT in the DBMS_REPCAT package. As shown in the following example, this object need not already exist:

```
DBMS_REPCAT.CREATE_MASTER_REPOBJECT(
    sname              => 'acct_rec',
    oname              => 'emp',
    type               => 'table',
    use_existing_object => TRUE,
    ddl_text           => 'CREATE TABLE acct_rec.emp AS . . .',
    comment            => 'created by . . .',
    retry              => FALSE,
    copy_rows          => TRUE,
    gname              => 'acct');
```

This example adds the EMP table to the replicated object group ACCT. The comment is added to the RepObject view. The types of objects that you can replicate are: tables, views, indexes, synonyms, triggers, procedures, functions, packages, and package bodies.

Note: If a table has a foreign key constraint that references columns in the table, you may need to precreate and populate the table at the new master site.

☞ **Attention:** Note that the symmetric replication facility does not support replication of clustered tables.

To avoid name conflicts, the name of the replicated table should not exceed 27 bytes in length.

You must call this procedure at the master definition site. Assuming that you have no other master sites yet, the object is created at the master definition site, if it does not already exist, using the DDL that you provided.

If you pass a SQL statement as the DDL text, it must not include a trailing semicolon. If you supply a PL/SQL package, package body, function, or procedure, it must include the trailing semicolon.

Before calling this procedure the replicated object group must be quiesced. If you are creating a new replicated object group, the system is automatically quiesced when you call CREATE_MASTER_REPGROUP; otherwise, you must call SUSPEND_MASTER_ACTIVITY.

You can also use this procedure to add an object to an existing replicated environment, which may include additional master sites. If your environment does include one or more additional master sites the object is asynchronously added to the replicated object group at those sites as described on page 4 – 35.

Replicating the Object at Each Master Site

When Oracle attempts to create the object at each master site, two parameters affect how the object is created: USE_EXISTING_OBJECT and COPY_ROWS. The default value for both of these parameters is TRUE.

If an object does not already exist at a master site, this procedure creates the object for you. For tables, you can choose to either populate the table with the data from the master definition site, or to populate the table yourself. If you choose to populate the table yourself, you are responsible for the consistency of the table at each master site.

If a table already exists at a master site, you can choose to

- use the existing table but allow Oracle to ensure consistency by updating the existing table with values from the master definition site, if any discrepancies are found
- use the existing table and be responsible for consistency yourself

Oracle raises a *duplicateobject* exception if the table at the master site is not the same shape as the table at the master definition site. Shape of a table refers to the number of columns, the name of these columns, and the datatype of the columns. This exception is stored in the RepCatLog view.

If any non–table object already exists at a master site, Oracle raises a *duplicateobject* exception if the object at the master site does not have the same SQL definition as the object at the master definition site. This exception is stored in the RepCatLog view.

Precreating an object can be useful if you find it faster to copy the files yourself (for example, by exporting the tables of a quiesced master site and importing them into a potential master site), instead of having the replication facility copy them for you. If you do this, you should take great care to ensure that the contents of the objects at both sites are identical. You also need to ensure that no changes are replicated to that site until you have imported the data.

Table 4 – 1 displays the appropriate settings for these parameters.

Object Already Exists?	COPY_ROWS	USE_EXISTING _OBJECT	Result
yes	TRUE	TRUE	*duplicateobject* message if objects do not match. For tables, use data from master definition site.
yes	FALSE	TRUE	*duplicateobject* message if objects do not match. For tables, Admin must ensure contents are identical.
yes	TRUE/FALSE	FALSE	*duplicateobject* message
no	TRUE	TRUE/FALSE	Object is created. Tables populated using data from master definition site.
no	FALSE	TRUE/FALSE	Object is created. Admin must populate tables and ensure consistency of tables at all sites.

Table 4 – 1 Object Creation at Master Sites

Additional Information: The parameters for the CREATE_MASTER_REPOBJECT procedure are shown in Table 12 – 102, and the exceptions are listed in Table 12 – 103. See also the discussion of offline instantiation on page 4 – 19 and page 5 – 10.

Using an Alternate Primary Key

If you are using row–level replication, Oracle must know which rows to compare when you push changes from one site to another. Because two rows may have different row IDs at different replication sites, Oracle uses the primary key for a table to determine which rows to compare. If you do not want to use the primary key for a table, or if the table does not have a primary key, you must call the SET_COLUMNS procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.SET_COLUMNS(
    sname       => 'acct_rec',
    oname       => 'emp',
    column_list => 'emp_no,ename');
```

In this example, both the employee number and the employee name are used to determine matching rows when pushing changes between replicated copies of the EMP table.

Because this key is used to determine matching rows at different sites, the columns that you specify in the COLUMN_LIST must result in a unique identifier for the row. The SET_COLUMNS procedure does not enforce uniqueness. This key is used as a substitute for the primary key for comparing replicated rows only, and is not a general substitute for defining a primary key constraint on a table.

Additional Information: The parameters for the SET_COLUMNS procedure are described in Table 12 – 167, and the exceptions are listed in Table 12 – 168.

Adding a Master Site

To add another master site to your replicated environment, call the ADD_MASTER_DATABASE procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.ADD_MASTER_DATABASE(
    gname                => 'acct',
    master               => 'acct_ny.ny.com',
    use_existing_objects => TRUE,
    copy_rows            => TRUE,
    comment              => 'master site added by '||user||
                            ' on '||sysdate,
    propagation_mode     => 'asynchronous');
```

In this example, the ACCT_NY database is added as a new master replication site for the ACCT replicated object group. This site will propagate changes to all other existing sites asynchronously, and all existing sites will asynchronously propagate changes to this site. For information on selecting a propagation mode, refer to page 4 – 33.

You must call ADD_MASTER_DATABASE at the master definition site. The replication catalog views at the new master site are updated with the information necessary to create the replicated object group. The replicated object group at the new master site is populated asynchronously as described on page 4 – 35.

The replicated objects are created in the replicated object group as described in "Replicating the Object at Each Master Site" on page 4 – 15.

Note: Oracle attempts to create the objects in dependency order. If you have circular dependencies between objects, you may need to precreate and populate these objects at the new master site in order for this procedure to complete successfully.

Note: If a table has a foreign key constraint that references columns in the table, you may need to precreate and populate the table at the new master site.

Additional Information: The parameters for the ADD_MASTER_DATABASE procedure are described in Table 12 – 64, and the exceptions are listed in Table 12 – 65.

Adding a Site to an Existing Replicated Environment

If you are adding a new master site to an existing replicated environment (one in which replication activity has already occurred), you must perform the following steps:

- Make sure that you have the necessary links and privileges for the new site as described on page 4 – 4. You must create links from this site to all existing sites, and from all existing sites to this site.
- Suspend all replication activity for the replicated object group by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY at the master definition site, as described on page 4 – 43.
- Add the new master site by calling ADD_MASTER_DATABASE as described above. When you add a site to an existing replicated environment, Oracle recreates all of the generated triggers and their associated packages at all existing master sites. This is necessary to ensure that DML changes at these existing sites are propagated to the new site using the appropriate method.

Warning: Do not resume replication activity or do additional DBMS_REPCAT.* administration for the replicated object group until the new master site appears in the RepSite view at the master definition site. Any changes that you make at any site will not be propagated to the new site until it is included in the RepSite view and you may not be able to resynchronize your data.

- If your new master database site will be propagating changes asynchronously, you must also schedule propagation of changes between the new master site and existing master sites, as described on page 4 – 26. Additionally, if any existing sites will be asynchronously propagating changes to this site, you must be certain that you have scheduled execution of the deferred transaction queue at those sites as well.
- After creating a new master site you must resume replication activity by calling DBMS_REPCAT.RESUME_MASTER_ACTIVITY at the master definition site.

Adding a Site to an Existing Replicated Environment Using Offline Instantiation

Offline instantiation of a master site allows you to create a new master site while limiting the time required for existing sites in your replicated system to be quiesced. It is primarily useful for those sites with very large databases where the time required to transfer the data through network links to the new site would be prohibitive.

Offline instantiation requires only that your existing master sites be quiesced long enough to do an export of the database objects to tape from the master site being used as the source. You can then unquiesce the sites, transport the tape to the new site, import the export file, then bring the new site online.

The following are the steps necessary to instantiate a new site:

- Make sure that you have the necessary links and privileges for the new site as described on page 4 – 4. You must create links from this site to all existing sites, and from all existing sites to this site.
- Suspend all replication activity for the replicated object group by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY at the master definition site, as described on page 4 – 43.

> **Warning:** Do not resume replication activity or do additional DBMS_REPCAT.* administration for the replicated object group until the new master site appears in the RepSite view at the master definition site. Any changes that you make at any site will not be propagated to the new site until it is included in the RepSite view and you may not be able to resynchronize your data.

- From the master definition site, call the procedure DBMS_OFFLINE_OG.BEGIN_INSTANTIATION (gname, new_site). This procedure will add the specified new_site to the replicated object group, gname.
- Export the user–defined tables for each schema making up the object group, gname. This can be done from any master site. Note that, if your object group consists of more than one schema, you will need to export the user–defined tables for *each* schema. For more information about using the Export utility, see page 7 – 12 and *Oracle7 Server Utilities*.
- From the master definition site, call the procedure DBMS_OFFLINE_OG.RESUME_SUBSET_OF_MASTERS (gname, new_site). At this point, normal, non–administrative activities can resume at the existing master sites. The new site, however, will remain quiesced.

Therefore, propagation of job queues to the new site is disabled, but the jobs will remain in the queue for later propagation when the new site is unquiesced.

- At this point, the tape holding the export file can be transferred to the new site.
- After the tape is available at the new site, from the new master site, call DBMS_OFFLINE_OG.BEGIN_LOAD (gname, new_site). This procedure disables triggers at the new site so that no updates can be made while you import your data. The procedure also disables propagation of the job queue from the new site to the master sites.
- Use the Import utility to import the data from tape. For more information about using the Import utility, see page 7 – 12 and *Oracle7 Server Utilities*.
- When the import is completed, reenable the new site's triggers and job queue propagation by calling DBMS_OFFLINE_OG.END_LOAD (gname, new_site).
- Return to the master definition site and call the procedure DBMS_OFFLINE_OG.END_INSTANTIATION (gname, new_site). This procedure resumes propagation to the new site.

Notes:

- If you will be creating multiple new sites, you must perform the above steps for each new site. New sites cannot be instantiated as a group.
- Be careful to call the listed procedures from the appropriate site. Failure to do so could cause unexpected and unwanted results.
- If you need to reenable queue propagation manually, use the procedure DBMS_DEFER_SYS.SET_DISABLED(). See page 12 – 20 for more information.
- The propagation method used for offline instantiation is asynchronous.

Generating Replication Support

The triggers, packages and procedures needed to support replication are not created until you call the GENERATE_REPLICATION_SUPPORT procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT(
    sname            => 'acct_rec',
    oname            => 'emp',
    type             => 'table',
    distributed      => TRUE,
    gen_rep2_trigger => FALSE
    gen_obj_owner    => 'REPADMIN');
```

You must call this procedure from the master definition site for each object in the replicated object group (tables, packages, package bodies, and procedures). If you are generating replication support for an object that is not owned by the replication administrator, the owner of the object must have the EXECUTE privilege on the DBMS_DEFER package.

> **Note:** If your compatibility mode is set to 7.3.0 or greater, the DISTRIBUTED option must be set to TRUE. If the compatibility mode at any of your snapshot sites is earlier than 7.3.0, you must set GEN_REP2@_TRIGGER to TRUE *and* set compatibility mode of the master site to 7.3.0.0.

Because the generated triggers may include calls to the generated procedures (to support synchronous replication), Oracle generates replication support in two phases. When you call GENERATE_REPLICATION_SUPPORT, Oracle begins phase one by synchronously broadcasting the request to all sites to create the necessary generated packages. These packages are created asynchronously as described on page 4 – 35. For procedural replication, phase one generates the package specification.

Phase two does not begin until each site indicates to the master definition site that it has generated the packages necessary to support replication. Oracle then begins phase two by synchronously broadcasting the request to generate the necessary triggers and their associated packages at each site. Once again, these objects are created asynchronously as described on page 4 – 35. For procedural replication, phase two generates the package body.

> **Note:** Oracle has been optimized to allow additional calls to GENERATE_REPLICATION_SUPPORT and to allow CREATE_MASTER_REPOBJECT to proceed after Oracle has broadcast the request to create the packages at each site.

It is not necessary to wait until all packages have actually been created at all of the sites to begin processing these types of requests. Any other procedures will not be executed until after GENERATE_REPLICATION_SUPPORT completes phase two.

There may be times when you only need to create the generated packages, such as when you change conflict resolution methods for an object, and other times when you only need to generate the triggers, such as when you change propagation methods for a site. In these situations, you can save time by using the DBMS_REPCAT procedures GENERATE_REPLICATION_PACKAGE (described on page 6 – 26) and GENERATE_REPLICATION_TRIGGER (described on page 4 – 35) as appropriate. Once again, Oracle can begin processing these requests before phase two of GENERATE_REPLICATION_SUPPORT begins.

Transaction Ownership

Under synchronous propagation, a transaction is owned by the owner of the trigger and will be propagated to the remote sit with that owner's privileges. You can change the ownership of the transaction, usually to the replication administrator who has full privileges at the remote site, by using GEN_OBJ_OWNER.

Row–Level Replication

If you want to use row–level replication for a table, you should call GENERATE_REPLICATION_SUPPORT immediately after any calls that define the replicated table, including CREATE_MASTER_REPOBJECT, ALTER_MASTER_REPOBJECT, ALTER_MASTER_PROPAGATION, SET_COLUMNS, and any conflict resolution routines, such as ADD_UPDATE_RESOLUTION. You may make multiple definition calls (for example, you may add multiple conflict resolution methods), before generating replication support. If you have only modified the propagation method, you may prefer to call GENERATE_REPLICATION_TRIGGER. If you have only modified the conflict resolution method for a table, you may prefer to call GENERATE_REPLICATION_PACKAGE.

Procedural Replication

If you are generating support for a package (body), Oracle generates the package (body) wrapper and you need to designate a prefix for the package (body). You should use the same prefix for the package body as you do for the package. If you do not designate a prefix, the default prefix is "defer_".

All of the parameters to your replicated procedure must be IN parameters, and must be of type: NUMBER, DATE, VARCHAR2, CHAR, ROWID or RAW. When generating wrappers, you should call GENERATE_REPLICATION_SUPPORT immediately after calling CREATE_MASTER_REPOBJECT, ALTER_MASTER_PROPAGATION, or ALTER_MASTER_REPOBJECT.

If you have only modified the propagation method, you may prefer to call GENERATE_REPLICATION_TRIGGER. You may also call GENERATE_REPLICATION_TRIGGER instead of GENERATE_REPLICATION_SUPPORT after adding a new master site (assuming that you have already called GENERATE_REPLICATION_SUPPORT for each object).

If you are using procedural replication, you should generate support for your replicated packages. Refer to page 8 – 13, for information on how to design your replicated procedures to work with tables that also support row–level replication. If you do not generate replication support for your replicated packages, you assume responsibility for making sure that the transactions are properly applied at each replicated site.

Additional Information: The parameters for the GENERATE_REPLICATION_SUPPORT procedure are described in Table 12 – 145, and the exceptions are listed in Table 12 – 146.

Synchronously Propagating DML Changes Among Master Sites

As shown in Figure 4 – 1, whenever you make a Data Manipulation Language (DML) change to a local table replicated using synchronous row–level replication, this change is synchronously propagated to the other master sites in the replicated environment using generated triggers and their associated packages. When you apply your local change, these triggers issue calls to generated procedures at the remote master sites.

Oracle ensures that all distributed transactions either commit or rollback in the event of a failure. See the discussion of distributed updates in *Oracle7 Server Distributed Systems, Volume I* for more information.

Restrictions

LONG and LONG RAW columns cannot be not replicated using row–level replication and, if present, are skipped prior to logging in the deferred RPC queue.

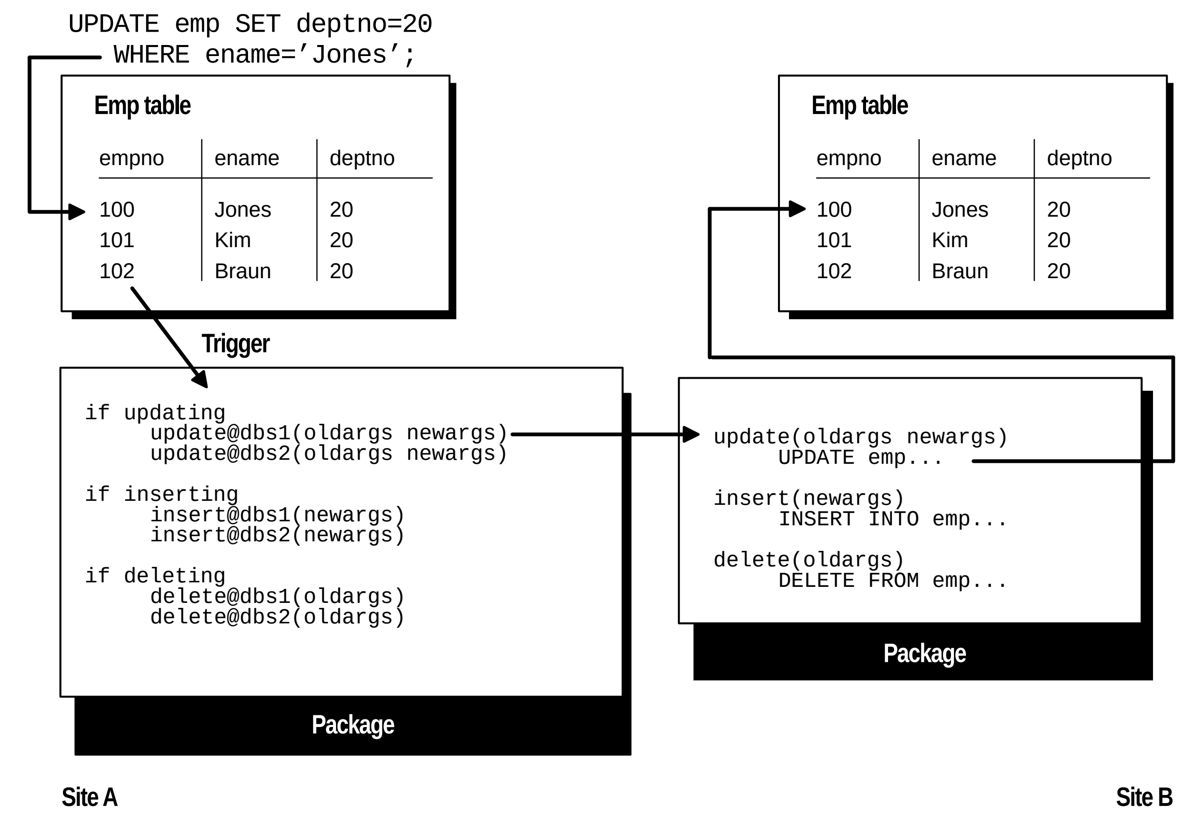


Figure 4 – 1 Propagating Changes using Synchronous Row–Level Replication

Warning: Because of the locking mechanism used by synchronous replication, deadlocks can occur. When you perform a synchronously replicated update, Oracle first locks the local row and then uses an AFTER ROW trigger to lock the remote row. These locks are released when the transaction commits at each site.

Destination of Synchronously Replicated Transactions

The necessary remote procedure calls to support synchronous replication are included in the generated trigger and its associated package for each object. When you call GENERATE_REPLICATION_SUPPORT, Oracle regenerates these triggers at all master sites to add the necessary remote procedure calls for the new site. Conversely, if you remove a master site, Oracle removes these calls from the generated trigger.

Privilege Domain at the Remote Site

The privilege domain in which the transaction is executed at the remote site is determined by the database link that is used to connect to the remote site. Because the remote procedure is called from within a local generated procedure, Oracle looks for a private database link for the owner of this generated procedure. The owner of a generated procedure can also be specified by using the parameter GEN_TRIG_OWNER of the GENERATE_REP_SUPPORT procedure to place triggers and packages in any valid schema.

Oracle uses the username and password supplied with this link to connect to the remote database. The privilege domain of this user at the remote site determines whether the remote generated procedure can be successfully executed to apply the transaction at the remote site. If no username and password is supplied with the link, Oracle uses the username and password associated with the local connected session; that is, the username and password for the user performing the local update.

If Oracle cannot find a link for the owner of the table, it looks for a public link. If Oracle cannot find a private database link, it will use a public database link, if one is available. Once again, if no username and password is supplied with the link, Oracle uses the username and password for the local connected session.

Suggestion: Including a username and password as part of the database link greatly simplifies the number of privileges that you need to grant to each user at each site. For example, if all of the objects in your replicated object group have the same owner, by creating a private database link for this user (with the username and password), you could simply grant this user EXECUTE ANY PROCEDURE at each site.

Conflict Detection

If all of your sites are communicating synchronously with one another you should never experience an update conflict. However, if even one site is sending changes asynchronously to another site, you may experience conflicts at any site in your replicated environment.

If the change is being propagated synchronously, an error is raised and a rollback will be required. If the change is propagated asynchronously, Oracle automatically detects these conflicts and either logs the conflict or, if you designate an appropriate resolution method, resolves the conflict. Conflict detection and conflict resolution are described in Chapter 6.

Asynchronously Propagating DML Changes Among Master Sites

As shown in Figure 4 – 2, whenever you make a Data Manipulation Language (DML) change to a local table replicated using asynchronous row–level replication, this change is asynchronously propagated to the other master sites in the replicated environment using generated triggers and their associated packages.

When you apply your local change, these triggers are fired to build deferred calls to generated procedures at the remote master sites. Procedural replication, illustrated on page 8 – 17, uses procedure wrappers to build deferred transactions. The deferred transactions, however, are propagated in the same manner, whether you use procedural replication, row–level replication, or some combination of both.

Propagation of deferred transactions is controlled by job queue processes that you set up. You may choose to forward calls at frequent intervals, such as every few seconds, thus simulating event–based propagation.

Alternatively, you may initiate propagation at points in time when connectivity is available or communications costs are lowest, such as during evening hours. Queued transactions can be propagated to different destinations at different frequencies according to priority. If a remote system is unavailable, the deferred transactions targeted for that system remain in the local queue for later propagation.

The DBMS_DEFER_SYS package contains the procedures that you must use to forward changes from your current master to another master site in the same replicated environment. There are two different methods that you can use to propagate changes to a master site:

- You can automatically propagate the changes at a regular interval by calling the SCHEDULE_EXECUTION procedure once for each master site.
- You can manually push the changes made at a given master site by calling the EXECUTE procedure to forward any changes made since the last time changes were propagated from the given master site, either manually or automatically.

Restrictions

LONG and LONG RAW columns cannot be not replicated using row–level replication and, if present, are skipped prior to logging in the deferred RPC queue.

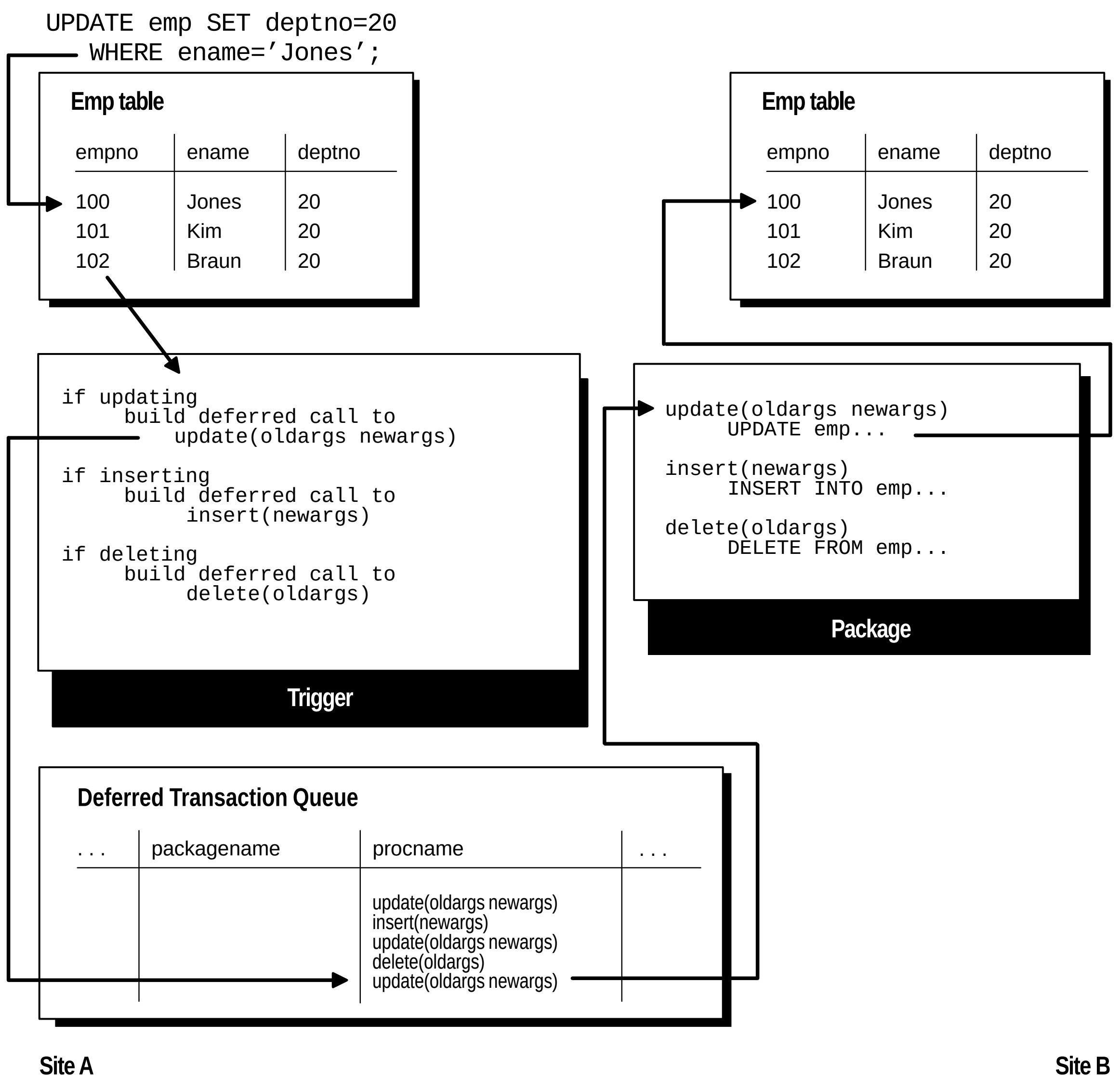
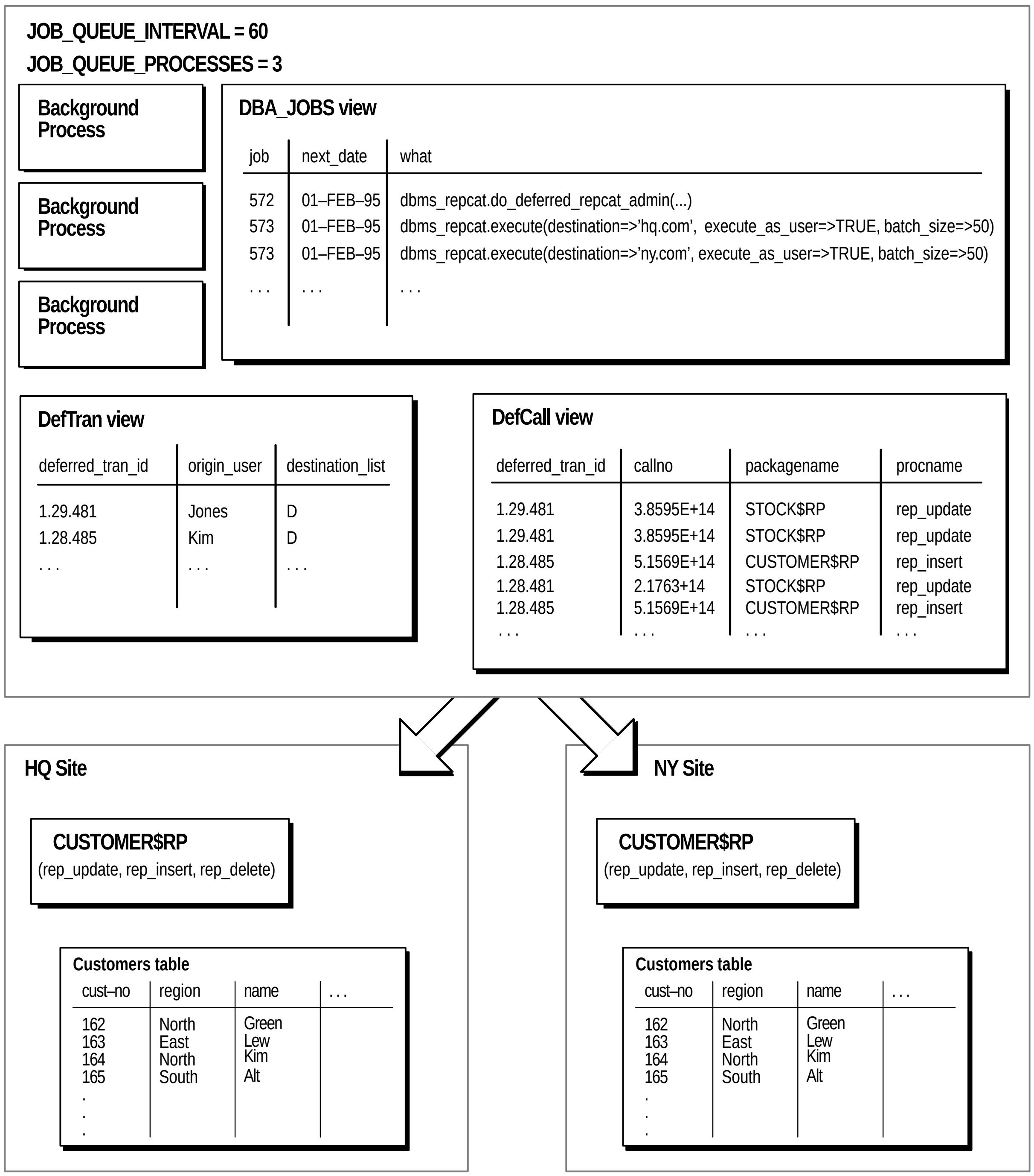


Figure 4 – 2 Applying Changes to an Updatable Snapshot

As shown in Figure 4 – 3, the arguments that you pass to the EXECUTE or SCHEDULE_EXECUTION procedures determine how the transactions are executed at the remote site. For scheduled executions, Oracle creates a job queue entry for you, and the settings of the job queue initialization parameters also affect the execution of the transactions.



job	next_date	what
572	01–FEB–95	dbms_repcat.do_deferred_repcat_admin(...)
573	01–FEB–95	dbms_repcat.execute(destination=>'hq.com', execute_as_user=>TRUE, batch_size=>50)
573	01–FEB–95	dbms_repcat.execute(destination=>'ny.com', execute_as_user=>TRUE, batch_size=>50)
. . .	. . .	. . .

deferred_tran_id	origin_user	destination_list
1.29.481	Jones	D
1.28.485	Kim	D
. . .	. . .	. . .

deferred_tran_id	callno	packagename	procname
1.29.481	3.8595E+14	STOCK$RP	rep_update
1.29.481	3.8595E+14	STOCK$RP	rep_update
1.28.485	5.1569E+14	CUSTOMER$RP	rep_insert
1.28.481	2.1763+14	STOCK$RP	rep_update
1.28.485	5.1569E+14	CUSTOMER$RP	rep_insert
. . .	. . .	. . .	. . .

cust–no	region	name	. . .
162	North	Green	
163	East	Lew	
164	North	Kim	
165	South	Alt	
.			
.			
.			

cust–no	region	name	. . .
162	North	Green	
163	East	Lew	
164	North	Kim	
165	South	Alt	
.			
.			
.			

Figure 4 – 3 Deferred Transactions

Destination of Deferred Transactions

For both scheduled and manual executions, you must specify the fully qualified database name for the remote site to which you want to push the deferred transactions.

☞ **Attention:** EXECUTE and SCHEDULE_EXECUTION do not expand database link names. If you do not specify a fully qualified database link name, your transactions cannot be successfully propagated to the remote site.

Connecting to Remote Destinations

The privilege domain in which the transaction is executed at the remote site is determined by the database link that is used to connect to the remote site. If the EXECUTE_AS_USER parameter is TRUE, Oracle looks for a database link for the connected session user. If EXECUTE_AS_USER is FALSE (the default), Oracle looks for a database link for the user who originated the transaction (that is, the user who issued the DML statements on the local table).

To ensure that the transactions are able to execute successfully at the remote site, you should create the necessary private database links for these users. These links should include a username and password, and the username specified should have the appropriate privileges on the generated procedures at the remote site.

Oracle first searches for the appropriate private database link. If none is found, it attempts to complete the connection using a public database link. If the database link includes a username and password, Oracle completes the connection using this information. If the link does not include a username and password, Oracle attempts to use the username and password from the current local session. This username and password combination must exist as a valid user at the remote site in order for the connection to succeed.

Note that if your database link does not include the username and password information, Oracle uses the username and password for the connected session, even if EXECUTE_AS_USER is FALSE. Because the user at the remote site determines the privilege domain in which the transaction is executed at the remote site, this may cause unexpected results.

If you are using a background process to push your deferred transaction queue (that is, if you called SCHEDULE_EXECUTION), your database link *must* include a username and password, or the connection will fail. This is because the current session, a background process, has no associated username and password.

Maintaining Data Integrity

The symmetric replication facility uses two–phase commit to ensure that transactions queued for a remote site are never lost. A transaction is not removed from the queue at the local site until it is successfully propagated to the remote site. Note that successful propagation does not imply successful completion of the transaction at the remote site. If the transaction cannot be successfully applied at the remote site, such as when an unresolvable conflict occurs, the transaction is logged in the DefError view at the remote site.

Deferred transactions maintain transactional consistency and ordering. Multiple procedure calls submitted within a single local transaction are executed as a single transaction remotely. Deferred transactions are executed remotely in the same order as they are committed locally. Order of execution within a transaction is also preserved. The deferred transaction executes every remote procedure call at each system in the same order as it was executed within the local transaction. Each remote procedure call and each deferred transaction is executed exactly once on each remote system.

Conflict Detection

When you push changes from one master site to another, Oracle compares the values for the row at the originating master site, before any changes were applied, to the current values for the row at the destination master site. If the values are different, a conflict has occurred. Conflict detection and conflict resolution are described in Chapter 6.

Scheduling Execution of the Deferred Transaction Queue

To establish communication between master sites, call the SCHEDULE_EXECUTION procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.SCHEDULE_EXECUTION(
    dblink    => 'acct_ny.ny.com',
    interval  => 'SYSDATE + 1',
    next_date => SYSDATE);
```

In this example, once a day, any deferred remote procedure calls queued at your current master site will be forwarded and applied to the ACCT_NY master site.

The scheduling information that you supply to this procedure is recorded in the DefSchedule view described in Table 13 – 24. You can either schedule the changes to be propagated once at a specified time, or to be forwarded periodically using a given formula.

To remove a scheduled job from the DefSchedule view, call the procedure UNSCHEDULE_EXECUTION.

When you call SCHEDULE_EXECUTION, Oracle creates a job queue entry using the INTERVAL and NEXT_DATE information that you supply to schedule the call to EXECUTE. Every few seconds (based on the value of the JOB_QUEUE_INTERVAL parameter), a background process checks the job queue to determine if there are any pending jobs. If so, the job is executed. You can experience some delay if you do not have enough available job queue background processes (as determined by the value of the JOB_QUEUE_PROCESSES parameter) to execute the outstanding jobs.

To schedule asynchronous propagation among all of your master sites, you must call this procedure N – 1 times at each master site, where N is the total number of master sites in your replicated environment. For example, if you have three master sites, you would need to call this procedure twice at each master site. To change the interval at which changes are forwarded, simply call this procedure again with a new interval formula.

> **Additional Information:** The parameters for the SCHEDULE_EXECUTION procedure are described in Table 12 – 25.

Forcing Execution of the Deferred Transaction Queue

To force the deferred transactions queued at your current master site to be pushed to another master site, call the EXECUTE procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.EXECUTE(destination => 'acct_ny.ny.com');
```

In this example, any deferred transactions queued at your current site are immediately pushed to the ACCT_NY master site. In this example, these transactions will be pushed as the user who originally performed the update at the local site.

When you call EXECUTE, Oracle forwards and applies any outstanding remote procedure calls queued at your current location, to the given master site and records any errors or unresolvable update conflicts in the DefError view at the destination site.

This procedure can also be called automatically by an Oracle background process that is scheduled using the SCHEDULE_EXECUTION procedure. Anytime EXECUTE is called automatically, Oracle records the date information in the LAST_DATE field of the DefSchedule view.

> **Additional Information:** The parameters for the EXECUTE procedure are shown in Table 12 – 23.

Removing a Master Site from the Deferred Execution List

To stop automatic propagation of deferred transactions between master sites, call the UNSCHEDULE_EXECUTION procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.UNSCHEDULE_EXECUTION( dblink => 'acct_ny.ny.com');
```

In this example, any deferred remote procedure calls queued at your current master or snapshot site will no longer be automatically forwarded to the master site referred to by the dblink ACCT_NY.NY.COM.

No changes will be pushed to this site from your current site until you either manually push them by calling EXECUTE or you reschedule automatic execution. You should call UNSCHEDULE_EXECUTION if you drop a master site from your replicated environment.

> **Additional Information:** The parameters for the UNSCHEDULE_EXECUTION procedure are described in Table 12 – 28, and the exceptions are listed in Table 12 – 29.

Determining if Propagation of the Deferred Transaction Queue is Enabled

To determine whether propagation of the deferred transaction queue from the current site to another site is enabled, call the DISABLED procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
    DBMS_DEFER_SYS.DISABLED ( dblink => 'acct_ny.ny.com');
```

The procedure will return TRUE, if propagation to the specified site is enabled, if not, it will return FALSE.

Selecting a Propagation Method

When you add a new database site, you get to select a default propagation method for that site. Because this propagation method determines how the new site both sends changes to and receives changes from all existing sites, the order in which you add sites is important.

For example, suppose that you have three sites: A, B, and C. If you first create site A as the master definition site, and then add site B with a propagation_mode of SYNCHRONOUS, site A will send changes to site B synchronously and site B will send changes to site A synchronously. There is no need to schedule execution at either site, because neither site is creating deferred transactions. Now suppose that you create site C with a propagation_mode of ASYNCHRONOUS. The propagation modes are now as illustrated in Figure 4 – 4.

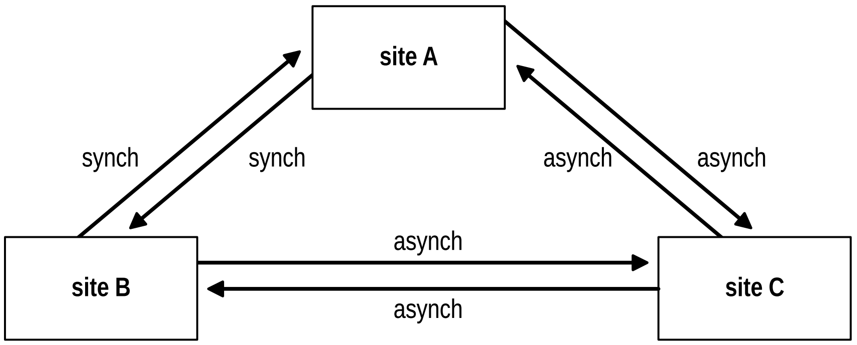


Figure 4 – 4 Selecting a Propagation Method

You must now schedule propagation of the deferred transaction queue from site A to site C, from site B to site C, and from site C to site A and to site B. But even more importantly, what if you created site A as the master definition site, then site C with a propagation_mode of ASYNCHRONOUS, then added site B with a propagation_mode of SYNCHRONOUS? Now the propagation modes would be as shown in Figure 4 – 5.

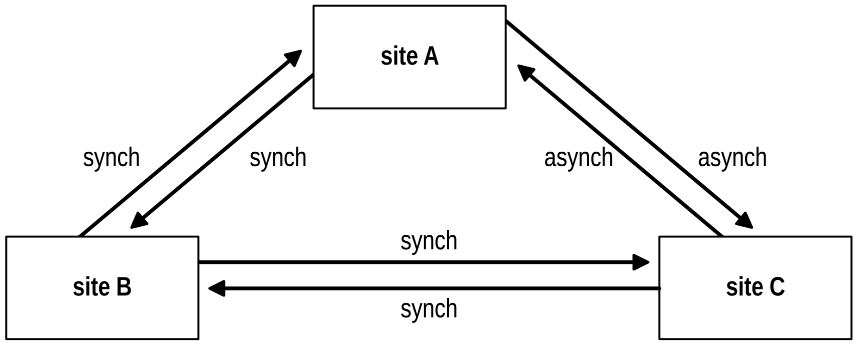


Figure 4 – 5 Ordering Considerations

Each time that you add a new site to your replicated environment, you must consider the effects of this addition on how changes are sent to and received from existing sites.

Altering Propagation

If you are not satisfied with the default propagation method between sites, you can alter the method used to send changes from one site to a given set of sites by calling DBMS_REPCAT.ALTER_MASTER_PROPAGATION, as shown in the following example:

```
DBMS_REPCAT.ALTER_MASTER_PROPAGATION(
            gname            => 'acct',
            master           => 'site_a',
            dblink_list      => 'site_b, site_c',
            propagation_mode => 'synchronous');
```

Site A, as illustrated in Figure 4 – 5, would synchronously propagate its updates to sites B and C, after the triggers are regenerated. Site A would continue to receive changes from these sites asynchronously.

You must call this procedure from site A, and the replicated object group for which you are altering the propagation mode must be quiesced. After altering the propagation mode for an object group, be sure to make the appropriate GENERATE_REPLICATION_SUPPORT, DBMS_DEFER_SYS.SCHEDULE_EXECUTION, or DBMS_DEFER_SYS.UNSCHEDULE_EXECUTION call at the designated master site.

Additional Information: The parameters for the ALTER_MASTER_PROPAGATION procedure are described in Table Table 12 – 72, and the exceptions are listed in Table Table 12 – 73.

Generating Replication Triggers

After altering the propagation mode of an object group, you need to regenerate the supporting triggers for these objects at each site in the replicated environment. To generate the supporting triggers and their associated packages for all members of an object group for a given set of master sites, call the DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER procedure, as shown in the following example:

```
DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER( gname => 'acct');
```

Because no list of master sites was specified in this example, Oracle will regenerate the supporting triggers and their associated packages for the objects in the GNAME object group at all master sites. You must call this procedure from the master definition site for the given replicated object group. Oracle must successfully create the necessary triggers at the master definition site for this procedure to complete successfully. These objects are asynchronously created at the other master sites as described on page 4 – 35.

Attention: The GENERATE_REPLICATION_TRIGGER procedure is overloaded to allow you to generate support for a single object at all master sites or for an object group at a list of sites. Because the parameter types are the same for both calls, you may need to use named notation to indicate whether you are calling the procedure for a single object or for an object group.

Additional Information: The parameters for the GENERATE_REPLICATION_TRIGGER procedure are described in Table 12 – 147, and the exceptions are listed in Table 12 – 148.

Propagating DDL Changes Among Master Sites

Whenever you make a Data Definition Language (DDL) change in a replicated environment, for example, by calling DBMS_REPCAT.ALTER_MASTER_REPOBJECT, this change must be propagated to all other sites in the replicated environment. Note that, in this manual, "DDL changes" refer to any schema–level changes, whether they involve user–supplied DDL statements (as is the case with ALTER_MASTER_REPOBJECT), or not (as is the case with CREATE_MASTER_REPOBJECT and other DDL–like procedures).

Warning: Schema level changes should only be made using the procedures provided in the DBMS_REPCAT package. As a replication administrator, you must ensure that any local customizations made outside the scope of the replication facility do not interfere with replication activities.

1. When you make a DDL change using the procedures provided in the DBMS_REPCAT package, the request for a DDL change is synchronously broadcast to each master site, where the request is recorded in the RepCatLog view. The DDL statement is stored in a child table of the RepCatLog view at each site. At the master definition site, the RepCatLog view also indicates that the master definition site is awaiting a callback from each master site. If this synchronous broadcast fails, for any reason, you receive an error message, and the transaction is rolled back.

2. Whenever you synchronously broadcast an event (such as when you make a DDL change), Oracle automatically inserts a job into the job queue, if one does not already exist for the replicated schema. This job periodically executes the procedure DO_DEFERRED_REPCAT_ADMIN. Whenever you synchronously broadcast an event, Oracle attempts to start this job immediately in order to apply the replicated changes at each master site.

   Assuming that Oracle does not encounter any errors, DO_DEFERRED_REPCAT_ADMIN will be run whenever a background process is available to execute the job.

   The initialization parameter JOB_QUEUE_INTERVAL determines how often the background process wakes up. You can experience a delay if you do not have enough background processes available to execute the outstanding jobs. For more information on scheduling jobs, see Chapter 10.

   If the initialization parameter JOB_QUEUE_PROCESSES is set to zero at a master site, you must manually connect to that site and invoke DO_DEFERRED_REPCAT_ADMIN to execute asynchronous requests at that site. Because this procedure may use dynamic SQL to perform DDL, you must never invoke it as a remote procedure call.

3. The next time that DO_DEFERRED_REPCAT_ADMIN is called at each master site, that site checks the RepCatLog view to see if there are any actions that need to be performed. When Oracle sees the request for a DDL change in the RepCatLog view, it applies the DDL statement in the child table to the current master site and updates any local views as appropriate. This event can occur asynchronously at each master site.

4. The success or failure of this action at each master site is noted in the RepCatLog view at each site. Ultimately, this information is propagated to the master definition site. If the event completed successfully at a master site, the callback for that site is removed from the RepCatLog view at the master definition site.
5. If the event completed successfully at all sites, all entries in the RepCatLog view at all sites, including the master definition site, will have been removed.

By synchronously broadcasting the change Oracle ensures that all sites are aware of the change, and thus are capable of remaining in sync. By allowing the change to be applied at the site at a future point in time, Oracle provides you with the flexibility to choose the most appropriate time to apply changes at a site.

If an object requires automatically generated replication support, you must regenerate this support after you alter the object by calling the GENERATE_REPLICATION_SUPPORT procedure. Oracle then updates the generated triggers, packages and procedures as necessary to support replication of the altered object at all master sites.

Note that although the DDL must be successfully applied at the master definition site in order for these procedures to complete without error, this does not guarantee that the DDL is successfully applied at each master site. The RepCatLog view contains interim status and any asynchronous error messages generated by the request.

Any snapshot sites that are affected by a DDL change are updated the next time you perform a refresh of the snapshot site. While all master sites can communicate with one another, snapshot sites can only communicate with their associated master site. For more information on snapshot site refresh, see page 5 – 20.

If you must alter the shape of a snapshot as the result of a change to its master, you must drop and recreate the snapshot by calling the DBMS_REPCAT.DROP_SNAPSHOT_REPOBJECT procedure and then the DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT procedure.

Quiescing the Replicated Environment

Many of the procedures used to alter your replicated environment require that you resynchronize the replicated data at each master site. This is referred to as quiescing the system. You can quiesce the system by calling SUSPEND_MASTER_ACTIVITY. After making the necessary changes to your replicated environment, you can resume normal replication activity by calling RESUME_MASTER_ACTIVITY. Both of these procedures must be called at the master definition site.

During a Quiesce

The following steps describe how the symmetric replication facility resynchronizes the master sites in your replicated environment when you call DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY. All master sites must be available to quiesce the replicated environment.

1. As shown in Figure 4 – 6, when you call SUSPEND_MASTER_ACTIVITY, Oracle synchronously broadcasts this request to each of the master sites, and this request is recorded in the RepCatLog view. Once the site begins to quiesce, its status in the Repgroup view changes from “normal” to “quiescing”. This is true of the master definition site as well. At the master definition site, the RepCatLog view also indicates that the master definition site is awaiting a callback (a message indicating that the site has successfully quiesced) from each of the master sites. Each master site must report back to the master definition site when its status changes from “quiescing” to “quiesced”. The master definition site cannot change its status to “quiesced” until all of the other master sites have reported back that they are now “quiesced”.

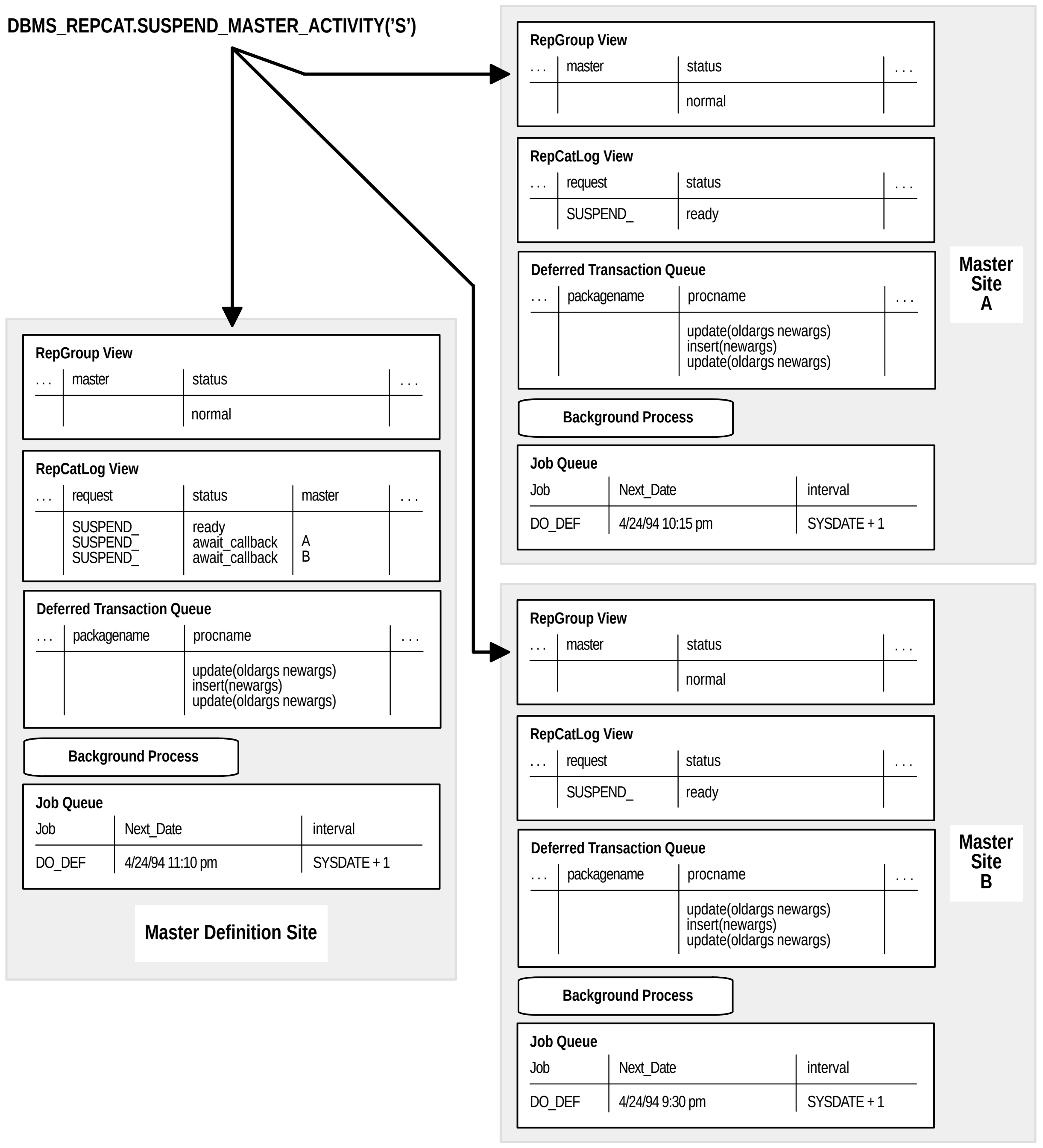


Figure 4 – 6 Quiescing the Replicated Environment
Step 1: Calling SUSPEND_MASTER_ACTIVITY

2. Every few minutes a background process wakes up at each master site and checks the RepCatLog view to see if there are any actions that need to be performed. As shown in Figure 4 – 7, the quiesce request implies that Oracle must push any deferred transactions between the master sites. This is accomplished by forcing the NEXT_DATE value for the associated job to SYSDATE. The pushing of the deferred transaction queue can occur asynchronously at each master site.

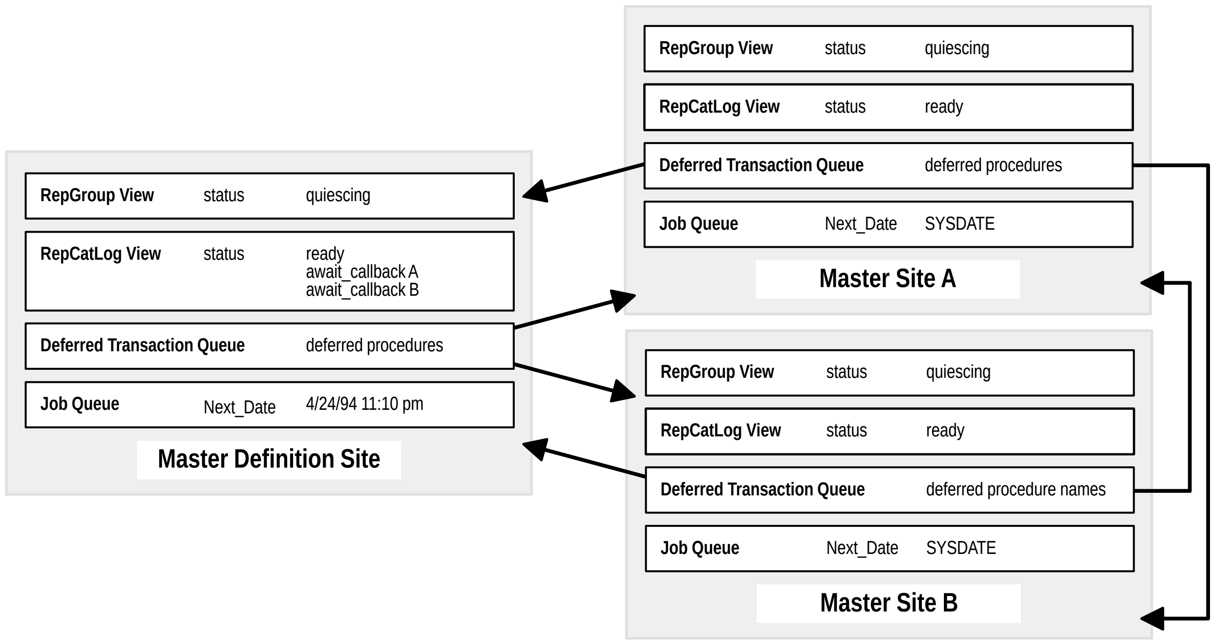

RepGroup View status quiescing
RepCatLog View status ready
Deferred Transaction Queue deferred procedures
Job Queue Next_Date SYSDATE
Master Site A
RepGroup View status quiescing
RepCatLog View status ready await_callback A await_callback B
Deferred Transaction Queue deferred procedures
Job Queue Next_Date 4/24/94 11:10 pm
Master Definition Site
RepGroup View status quiescing
RepCatLog View status ready
Deferred Transaction Queue deferred procedure names
Job Queue Next_Date SYSDATE
Master Site B

Figure 4 – 7 Quiescing the Replicated Environment Step 2: Pushing the Deferred Transaction Queue

3. As each master site successfully pushes its outstanding transactions, its status changes to “quiesced”. As shown in Figure 4 – 8, ultimately, this status is sent to the master definition site, and the “await_callback” entry for that master site is removed from the RepCatLog view at the master definition site.

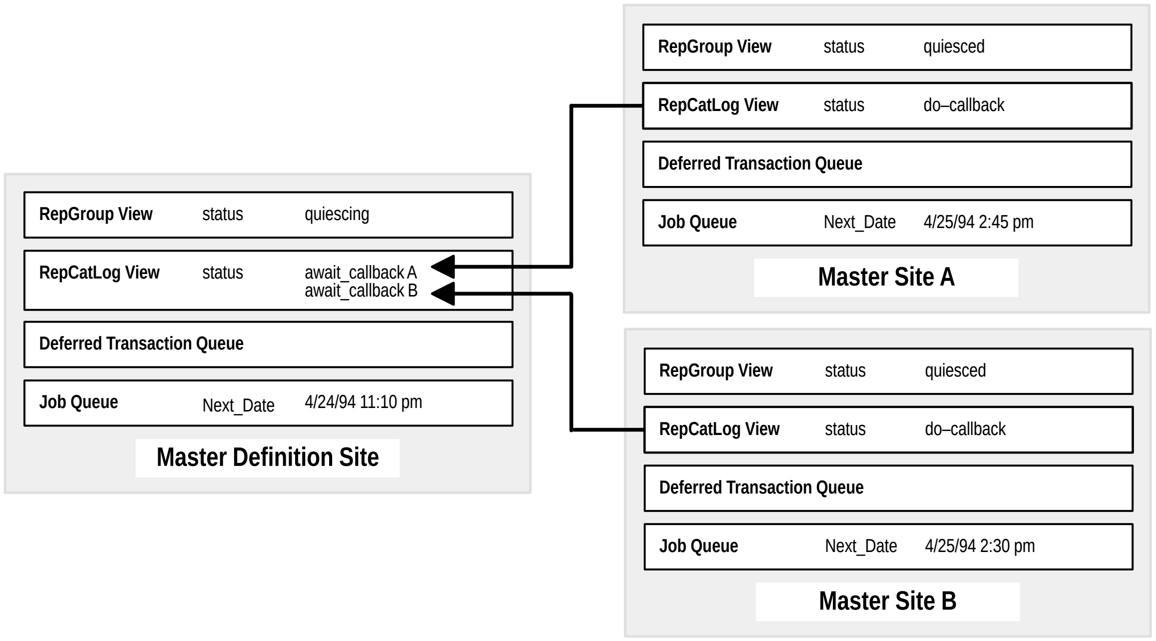

RepGroup View status quiesced
RepCatLog View status do–callback
Deferred Transaction Queue
Job Queue Next_Date 4/25/94 2:45 pm
Master Site A
RepGroup View status quiescing
RepCatLog View status await_callback A await_callback B
Deferred Transaction Queue
Job Queue Next_Date 4/24/94 11:10 pm
Master Definition Site
RepGroup View status quiesced
RepCatLog View status do–callback
Deferred Transaction Queue
Job Queue Next_Date 4/25/94 2:30 pm
Master Site B

Figure 4 – 8 Quiescing the Replicated Environment
Step 3: Master Sites Return Status to Master Definition Site

4. When the master definition site has successfully quiesced, and is satisfied that all of its associated master sites are also “quiesced”, it changes its status to “quiesced”, as shown in Figure 4 – 9.

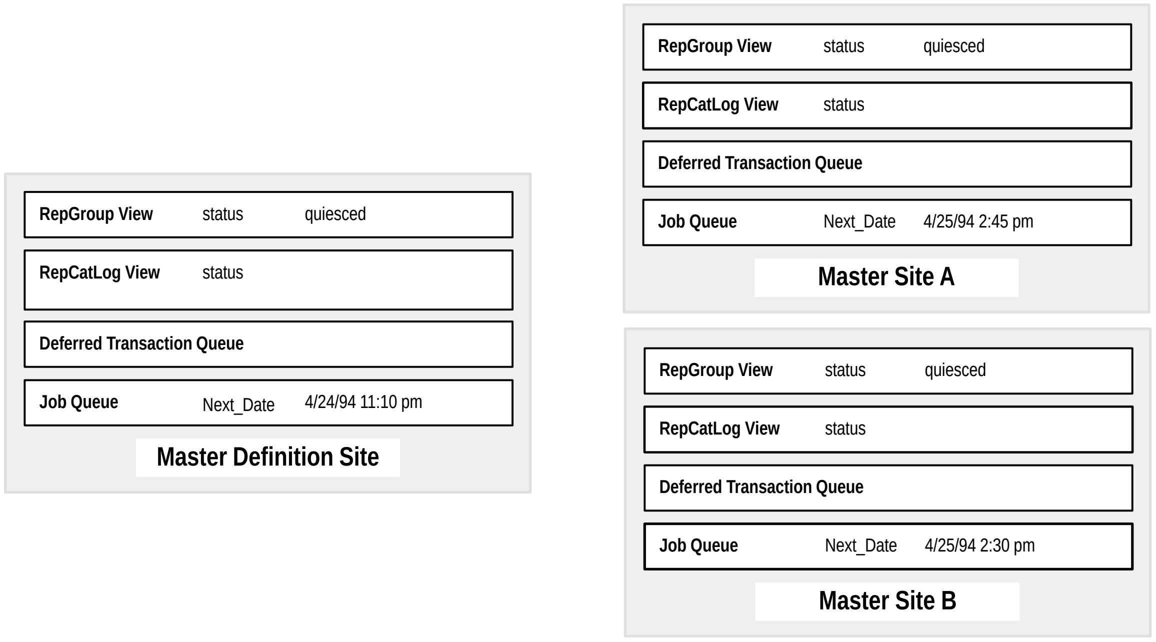

RepGroup View status quiesced
RepCatLog View status
Deferred Transaction Queue
Job Queue Next_Date 4/24/94 11:10 pm
Master Definition Site
RepGroup View status quiesced
RepCatLog View status
Deferred Transaction Queue
Job Queue Next_Date 4/25/94 2:45 pm
Master Site A
RepGroup View status quiesced
RepCatLog View status
Deferred Transaction Queue
Job Queue Next_Date 4/25/94 2:30 pm
Master Site B

Figure 4 – 9 Quiescing the Replicated Environment Step 4: Quiesce Completed

Suspending Replication Activity

To suspend replication activity for an object group, call the procedure SUSPEND_MASTER_ACTIVITY in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAST.SUSPEND_MASTER_ACTIVITY(
     gname            => 'acct'
     execute_as_user  => 'FALSE'));
```

This example suspends all replication activity for the ACCT object group. You can continue to query objects in this object group, but any modifications that cause replication triggers to fire will not succeed.

☞ **Attention:** The current implementation of SUSPEND_MASTER_ACTIVITY quiesces all replicated object groups at each master site. No updates can occur to replicated data if the status of any replicated object group at that site is "quiescing" or "quiesced".

You must call this procedure from the master definition site. The replicated object group must be in normal operation when you call this procedure. It suspends all activity at all master replication sites that invoke calls to generated replication procedures. All pending queued procedure calls are processed. Local updates that do not fire replication triggers will still succeed. Each master remains in this state until you invoke the procedure RESUME_MASTER_ACTIVITY.

This procedure typically operates asynchronously at the master definition and the master sites. The RepCatLog view contains interim status.

You must call DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY before calling any of the procedures listed below:

- DBMS_REPCAT.ADD_MASTER_DATABASE
- DBMS_REPCAT.ALTER_MASTER_REPOBJECT
- DBMS_REPCAT.ALTER_MASTER_PROPAGATION
- DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT
- DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER
- DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE
- DBMS_REPCAT.CREATE_MASTER_REPOBJECT

There may be additional situations in which you find it necessary to suspend replication activity. For example, administrators may wish to suspend activity and manually perform queries and updates on the replicas to restore equivalence if an unexpected conflict is detected, which was not resolved.

At a minimum, you must wait until the status field in the RepGroup view is “quiescing” at the master definition site. If the presence of a non–empty deferred transaction queue at a site could cause a problem, you should wait until this status is “quiesced” before proceeding. If an enabled replication trigger at a site could cause a problem, you should wait until the status is “quiesced” before proceeding.

Additional Information: The parameter for the SUSPEND_MASTER_ACTIVITY procedure is described in Table 12 – 169, and the exceptions are listed in Table 12 – 170.

Transaction Ownership

Under synchronous propagation, a transaction is owned by the owner of the trigger and will be propagated to the remote sit with that owner’s privileges. You can change the ownership of the transaction, usually to the replication administrator who has full privileges at the remote site, by using GEN_OBJ_OWNER.

Resuming Replication Activity

After quiescing your replicated environment, you can resume normal replication activity by calling the RESUME_MASTER_ACTIVITY procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.RESUME_MASTER_ACTIVITY(gname => 'acct');
```

In this example, deferred remote procedure calls can once again be queued in master sites containing the ACCT replicated object group. You must call this procedure from the master definition site. When you call this procedure, the request is synchronously broadcast to all other master sites in the replicated environment, and the status for the replicate object group is updated to “normal” in the RepCatLog view.

Additional Information: The parameters for the RESUME_MASTER_ACTIVITY procedure are described in Table 12 – 165, and the exceptions are listed in Table 12 – 166.

Altering a Replicated Object

DDL changes to a replicated object must be made by calling DBMS_REPCAT.ALTER_MASTER_REPOBJECT from the master definition site. The ALTER_MASTER_REPOBJECT procedure lets you supply the DDL that you want to have applied to the replicated object.

To ensure that the change is properly applied at all master sites, you should apply these changes in the following order:

1. If the object being altered is a master table for any updatable snapshots, you should replicate any snapshot changes to the master table as described on page 5 – 12.
2. Before altering a replicated object, you must quiesce the system by calling SUSPEND_MASTER_ACTIVITY at the master definition site. This pushes any queued transactions to the master sites in the replicated environment.
3. Alter the object by calling ALTER_MASTER_REPOBJECT at the master definition site. If the object requires replication support, you must regenerate any necessary triggers and packages by calling GENERATE_REPLICATION_SUPPORT.
4. Check the RepCatLog view at the master definition site to ensure that the object was successfully modified at each master site. The DDL changes to the object and any supporting objects are asynchronously applied at each master site as described on page 4 – 35.
5. If you altered a table used as the master for any snapshots, you must now drop and re–create those snapshots. Any other replicated objects at a snapshot site would be automatically re–created the next time you called REFRESH_SNAPSHOT_REPGROUP at the snapshot site.
6. You may now resume normal replication activity by calling RESUME_MASTER_ACTIVITY at the master definition site.

Supplying the DDL

To alter an object in your replicated environment, call the ALTER_MASTER_REPOBJECT procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.ALTER_MASTER_REPOBJECT(
    sname    => ’acct_rec’,
    oname    => ’emp’,
    type     => ’table’,
    ddl_text => ’ALTER TABLE acct_rec.emp ADD
                (healthplan NUMBER(7,2) DEFAULT 100 NOT NULL)’,
    comment  => ‘updated by ’||user ||’ on ‘||SYSDATE);
```

This example adds another column to the EMP table. The DDL text supplied is asynchronously applied at each master site as described on page 4 – 35. If you had made this change using the ALTER TABLE command, instead of by passing this command as an argument to the ALTER_MASTER_REPOBJECT procedure, this change would not have been propagated to the other master sites in the replicated environment.

Local customization of individual replicas at snapshot or master sites is outside the scope of Oracle's symmetric replication facility. As a replication administrator, you must ensure that local customizations do not interfere with any global customizations done with ALTER_MASTER_REPOBJECT.

Additional Information: The parameters for the ALTER_MASTER_REPOBJECT procedure are described in Table 12 – 74, and the exceptions are listed in Table 12 – 75.

Dropping a Replicated Object

This section describes how to drop an object from a replicated object group. This change ultimately affects all master and snapshot sites. Information on adding an object to the replicated object group is described on page 4 – 14.

To drop a replicated object from a replicated object group, call the DROP_MASTER_REPOBJECT procedure in the DBMS_REPCAT package at the master definition site, as shown in the following example:

```
DBMS_REPCAT.DROP_MASTER_REPOBJECT(
    sname        => 'acct_rec',
    oname        => 'emp',
    type         => 'table',
    drop_objects => FALSE);
```

In this example, when the procedure is executed, all automatically generated supporting objects, such as triggers and packages, used to replicate EMP will be dropped from each site. Because DROP_OBJECTS is set to FALSE, the default, the EMP table will continue to exist at all of the master sites. You can request that the object and any dependent objects be dropped at all sites by setting DROP_OBJECTS to TRUE.

This change is synchronously broadcast to each master site as described on page 4 – 35. Because this procedure is then executed asynchronously at each site, the RepCatLog view contains the interim status and any asynchronous error messages generated by the request.

Before dropping a replicated table, be certain that no snapshots depend on this object.

Additional Information: The parameters for the DROP_MASTER_REPOBJECT procedure are described in Table 12 – 123, and the exceptions are listed in Table 12 – 124.

Removing a Master Site

To remove one or more master databases from your replicated environment, complete the following steps in order:

1. Remove the databases from the replicated environment by calling REMOVE_MASTER_DATABASES at the master definition site.
2. Remove the replicated object group from each database by calling DROP_MASTER_REPGROUP at each of the master sites dropped in step 1.

Dropping a Master Site If You Cannot Quiesce

Oracle Corporation recommends that DBMS_REPCAT.REMOVE_MASTER_DATABASES be called for a replicated object group only when that object group is quiesced. When an object group cannot be quiesced and one or more masters must be removed from the object group, the replication administrator must

- clean the deferred RPC queue and
- remove any data inconsistencies.

The next time the object group is quiesced, in particular, the replication administrator must complete the following steps in order as soon as possible after the masters are removed.

1. Quiesce the object group by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY at the master definition site.
2. Delete transactions whose destination set includes the removed masters by calling DBMS_DEFER_SYS.DELETE_TRANSACTION at each remaining master.

3. Delete transactions whose destination set includes former and remaining masters by calling DBMS_DEFER_SYS.DELETE_TRANSACTION at each former master.
4. Resolve each conflict in the DefError view at each remaining master, either by applying the error with DBMS_DEFER_SYS.EXECUTE_ERROR or by deleting the error with DBMS_DEFER_SYS.DELETE_ERROR.
5. Ensure that the DefError and DefTranDest views are empty at each remaining master.
6. If the DefTran view is not empty at a remaining master, truncate the SYSTEM.DEF$_CALL table at that master.
7. Ensure that all replicated data is consistent. The DBMS_RECTIFIER_DIFF package can be used to determine and fix differences.
8. Unquiesce the object group by calling DBMS_REPCAT.RESUME_MASTER_ACTIVITY at the master definition site.

Dropping a Master Site from a Replicated Environment

To remove one or more master databases from a replicated environment, call the REMOVE_MASTER_DATABASES procedure in the DBMS_REPCAT package.

Calling REMOVE_MASTER_DATABASES causes the replication triggers and their associated packages to be regenerated at all remaining master sites.

You cannot remove the master definition site. The databases being removed do not have to be accessible when you call this procedure. All other master databases in the replicated environment must be available.

For example, suppose A is the master definition site and sites B, C, D, and E are master sites for the replicated schema R. If masters C and E become inaccessible and should no longer be masters, you should execute the following procedure at site A:

```
DBMS_REPCAT.REMOVE_MASTER_DATABASES(
    gname       => ’R’,
    master_list => ’C,E’);
```

The following code is also acceptable:

```
master_table dbms_utility.dblink_array;
...
master_table(1) := 'C';
master_table(2) := 'E';
DBMS_REPCAT.REMOVE_MASTER_DATABASES(
    gname        => 'R',
    master_table => master_table);
```

After calling this procedure, you should call DROP_MASTER_REPGROUP at each of the master sites that you removed. Otherwise, all of the replication views, such as RepGroup, will still be available (making the object group appear to be replicated), but the site will no longer receive changes from other master sites.

Additional Information: The parameters for the REMOVE_MASTER_DATABASES procedure are described in Table 12 – 161, and the exceptions are listed in Table 12 – 162.

Dropping a Replicated Object Group from a Former Master Site

To drop a replicated object group from your current site, call the DROP_MASTER_REPGROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.DROP_MASTER_REPGROUP(
    gname         => 'acct',
    drop_contents => TRUE,
    all_sites     => FALSE);
```

In this example, when this procedure is executed, the ACCT object group is no longer replicated at the current site. Because the second argument is TRUE, all replicated objects in this object group are dropped from the database.

Dropping the replicated object group removes all entries in the local replication views, such as RepGroup, that pertain to that replicated object group. Dropping the replicated object group at a master site also does not automatically remove that master site from the RepSite view at all other master sites. Before calling this procedure, you should call REMOVE_MASTER_DATABASES at the master definition site to ensure that replicated changes are no longer sent to the object group that you are dropping. Additionally, before calling this procedure, you can avoid unnecessary errors if you make sure that this object group is not being used as the master replication object group for any snapshot sites.

To drop the master definition site, you must first have dropped all of the master sites in the replicated environment. To drop the replicated object group from all sites, including the master definition site, you can call this procedure at the master definition site, and set ALL_SITES to TRUE.

Additional Information: The parameters for the DROP_MASTER_REPGROUP procedure are described in Table 12 – 121, and the exceptions are listed in Table 12 – 122.

Listing Information about a Replicated Environment

Whenever you install symmetric replication capabilities at a site, Oracle installs the *replication catalog,* which consists of tables and views. These tables are used by master and snapshot sites to determine such information as what objects are being replicated, where they are being replicated, and if any errors have occurred while administering replication.

Warning: Do not modify the replication catalog tables directly. All modification should be made to replication catalog views by using the procedures provided in the DBMS_REPCAT package.

Each of these views is briefly described below. They are described in more detail in Chapter 13.

Each view has three versions, which have different prefixes: USER_*, ALL_*, and SYS.DBA_*. The views used to determine the status of the objects in your replication environment are as follows:

RepGroup	Indicates which schemas are being replicated at a given site.
RepSite	Indicates where each object group is being replicated
RepObject	indicates which objects in each schema are being replicated.
RepCatLog	Indicates the status of any asynchronously propagated administrative changes.

CHAPTER

5

Snapshot Site Replication

This chapter explains how to use create and maintain snapshot sites. The topics discussed include the following:

- creating updatable snapshots
- creating a snapshot site
- propagating changes between sites
- refreshing updatable snapshots

Note that most of the activities described in this chapter can be accomplished much more easily by using Oracle's Replication Manager, a GUI interface for replication. See the documentation for Oracle Replication Manager for more information.

Replication Support at Snapshot Sites

A snapshot site can contain a subset of the information stored at its master site or all the objects at a master site. A snapshot site can contain the following objects:

- read–only snapshots of master tables in the replicated object group
- updatable snapshots of master tables in the replicated object group
- replicated triggers used to queue deferred remote procedure calls from the snapshot site to its associated master site
- objects that are replicated only in the sense that their SQL definitions are replicated (procedures, packages, functions, synonyms, and views)

Replication Support for Snapshots

Read–only snapshots require no special support from the symmetric replication facility. Read–only snapshots are created and refreshed in the same manner as described in Chapter 3.

☞ **Attention:** This chapter describes only the *differences* between updatable and read–only snapshots. If you are not already familiar with snapshots, snapshot refresh groups, and the refresh mechanism, you must read Chapter 3 before proceeding.

Similar to read–only snapshots, an updatable snapshot is a full copy of a table or a subset of a table that reflects a recent state of the master table. Unlike read–only snapshots, updatable snapshots must be derived from a single master table; that is, they must be simple snapshots.

Also, like read–only snapshots, updatable snapshots must be periodically refreshed to apply the changes made to the master table. However, unlike read–only snapshots, when you refresh an updatable snapshot, the changes to the snapshot must also be taken into account.

When you create your updatable snapshots using the procedures provided with the symmetric replication facility, the changes made to the updatable snapshot are either synchronously or asynchronously applied at the master site, in much the same manner as changes are propagated between two master sites.

Changes from the master site, however, are asynchronously propagated to the snapshot site in the form of a refresh, in much the same manner as read–only snapshots are refreshed from their masters. The refresh mechanism for updatable snapshots is described in detail on page 5 – 12.

Updatable Snapshot Architecture

In addition to the objects created for read–only snapshots that are described on page 3 – 2, when you create an updatable snapshot, two additional objects are created at the snapshot site:

- Oracle creates a table, named USLOG$*_snapshot_name,* to store the ROWID and timestamp of rows updated in the snapshot. The timestamp column is not updated until the log is first used by a snapshot refresh.
- Oracle creates an AFTER ROW trigger on the snapshot base table to insert the ROWIDs and timestamps of updated and deleted rows into the updatable snapshot log. The trigger is named USTRG$*_snapshot_name.*

When you create the snapshot as a replicated object, Oracle creates an additional trigger and associated package on the snapshot base table to call the generated procedures at the master site to apply the changes.

If you are propagating your changes synchronously, the trigger package directly executes the generated procedures at the master site, if you are using asynchronous propagation, the trigger package inserts the necessary deferred transactions into the deferred transaction queue at the snapshot site.

Of course, the primary difference between the architecture of read–only and updatable snapshots is that for read–only snapshots, Oracle creates a read–only view of the underlying base table, while for updatable snapshots, this view is writable.

If your CREATE SNAPSHOT statement includes a restriction in the where clause, this restriction is reflected in the values that are displayed when you create or refresh the snapshot. For example, if you issue the following statement:

```
CREATE SNAPSHOT emp FOR UPDATE
   AS SELECT * FROM scott.emp@sales.ny.com
     WHERE empno > 500;
```

your EMP snapshot will only display employees with employee numbers greater than 500. This behavior is identical to read–only snapshots. However, you are not restricted from updating this number, or inserting an employee with an employee number less than 500.

These values will remain in the table, and can be updated or deleted, until the next time that you refresh the snapshot. The refresh will remove any remaining rows with an employee number less than 500 for the snapshot. For more information on how these changes are propagated between the snapshot and its associated master table, see page 5 – 12.

If you want to restrict the data in the updatable snapshot to always satisfy the requirements specified in the WHERE clause of the original CREATE statement, you should define your own view on the snapshot base table, using a CHECK constraint.

Replication Support for Non–snapshot Objects

Do not alter non–snapshot objects at the snapshot site. These objects *must* be altered only at the master definition site by using the DBMS_REPCAT package.

If your snapshot site contains any non–snapshot replicated objects that were altered using the DBMS_REPCAT package at its associated master site (these would typically result from DDL changes propagated from the master definition site), these changes can be applied at the snapshot site the next time that you refresh the replicated schema with REFRESH_OTHER_OBJECTS set to TRUE.

Before Creating a Snapshot Site

Before creating a snapshot site, you must already have created at least one master site, as described in Chapter 4. If you have multiple master sites in your replicated environment, you should select which master site your snapshot site will be created from.

This master site will also be used to refresh any read–only or updatable snapshots at your new snapshot site. You can change a snapshot site's master site if required.

Creating Database Links for Snapshot Sites

A snapshot site for a replicated object group must have a database link to its associated master site that contains a username (for example, SCOTT), a password (for example, TIGER), and the fully qualified database name of the master. The appropriate database links must be created before you call CREATE_SNAPSHOT_REPGROUP or SWITCH_SNAPSHOT_MASTER.

Granting the Necessary Privileges

The user at the master site that was specified in the CONNECT TO clause of the database link from the snapshot site must either be granted replication administrator privileges or be the owner of the schemas in the replicated object group in order for deferred transactions to be propagated to the master site. Additionally, for the refresh of any updatable snapshots to succeed, this user must either own the snapshots or be SYS, or have the ALTER ANY SNAPSHOT privilege.

Creating a Snapshot Replication Site

This section outlines the procedure that you must perform to create a snapshot site containing updatable snapshots. Each of these steps is described in more detail later in this chapter.

Optionally, you may perform the following step at the master site:

- Create a snapshot log for the master table using the CREATE SNAPSHOT LOG command. This allows you to perform fast refreshes of your snapshots. Refer to page 3 – 8 for information on creating and using snapshot logs.

Perform the following steps at *each snapshot site:*

- Create the necessary snapshots using the FOR UPDATE clause of the CREATE SNAPSHOT command, as described on page 5 – 8.
- Create the necessary links between the snapshot site and its master site, as described on page 5 – 4.
- Create the replicated object group at the snapshot site by calling CREATE_SNAPSHOT_REPGROUP.
- Create the replicated objects, including updatable snapshots, at the snapshot site by calling CREATE_SNAPSHOT_REPOBJECT.

  This procedure generates the necessary replication support for each object.
- Ensure that multiple snapshots will be refreshed to a single point in time by using the DBMS_REFRESH.MAKE procedure to maintain transactional consistency and preserve master detail relationships.

Creating the Replicated Object Group

Create a new empty snapshot replicated object group in your local database by calling the CREATE_SNAPSHOT_REPGROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPGROUP(
    gname            => ’accts’,
    master           => ’acct_hq.hq.com’,
    comment          => ’created on ...’,
    propagation_mode => ’asynchronous’);
```

☞ **Attention:** Notice that the replicated object group name must match the master group name.

In this example, the ACCTS object group is created in the current database. When you add replicated objects to this object group by calling CREATE_SNAPSHOT_REPOBJECT, they will be refreshed using the ACCT_HQ database as their master. Changes from the snapshot site will be asynchronously propagated to its associated master site as described on page 5 – 19.

Because each snapshot site may contain a different subset of the objects at its associated master site, there is no snapshot equivalent to the DBMS_REPCAT.ADD_MASTER_DATABASE procedure. If you will be creating multiple snapshot sites with similar members, you may want to create them using a script, which can be modified and re–executed at each site.

Additional Information: The parameters for the CREATE_SNAPSHOT_REPGROUP procedure are described in Table 12 – 105, and the procedures are listed in Table 12 – 106.

Creating a Replicated Object

Add a replicated object to your snapshot site by calling the procedure CREATE_SNAPSHOT_REPOBJECT in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
    sname    => 'accts_rec',
    oname    => 'emp',
    type     => 'snapshot',
    ddl_text => 'CREATE SNAPSHOT accts_rec.emp FOR UPDATE AS
                SELECT * FROM emp@acct_hq.hq.com WHERE
                deptno > 500',
    comment  => 'created on ...',
    gname    => 'acct')
    gen_obj_owner ==> 'REPADMIN';
```

In this example, Oracle creates a snapshot at the snapshot site of the EMP table located at the master site. This snapshot contains a subset of the rows in the EMP table; that is, it only contains the rows for those employees whose department numbers are larger than 500. For example, this might be all sales staff in the western region.

☞ **Attention:** Notice that the object name and updatable snapshot name must be identical. The master table must be a replicated object registered at the master site.

In this example, the SQL statement necessary to create the snapshot is passed as an argument to this procedure. Alternatively, you could have created the snapshot before calling this procedure and simply have omitted the DDL.

⚠ **Warning:** If you create an updatable snapshot using the FOR UPDATE clause of the CREATE SNAPSHOT command, but do *not* create the snapshot as a replicated object by calling CREATE_SNAPSHOT_REPOBJECT, any changes that you make to the updatable snapshot will be lost when you refresh the snapshot.

In addition to creating the snapshot as a replicated object at the snapshot site, Oracle also adds the object name and type to the RepObject view at the local site. This view is used to determine which objects need to push their changes to the master site.

Because the replicated object in this example is of type SNAPSHOT and its associated master table uses row–level replication, Oracle installs the appropriate replication support at the snapshot site. For snapshots of tables using procedural replication, be sure to replicate the associated procedure or package at the snapshot site.

☞ **Attention:** Although not required, you will typically want all snapshots at a given snapshot site to be in the same snapshot refresh group. To ensure that snapshots in the same refresh group are refreshed consistently, their associated master tables must be located at the same master site.

Additional Information: The parameters for the CREATE_SNAPSHOT_REPOBJECT procedure are shown in Table 12 – 107, and the exceptions are listed in Table 12 – 108.

Creating Non–Snapshot Objects

For objects other than snapshots, you must not supply the DDL. Oracle copies the object from the master site that you designated when you created the snapshot site. If the object already exists at the snapshot site, Oracle compares the two objects and raises a *duplicateobject* exception if they do not match.

Creating Snapshots

For snapshots, you can either precreate the snapshot, or supply the DDL as part of the CREATE_SNAPSHOT_REPOBJECT call. For information on creating read–only snapshots, refer to Chapter 3.

Updatable snapshots are created and deleted in the same manner as read–only snapshots. See Chapter 3. To create an updatable snapshot, simply add the FOR UPDATE clause to the CREATE SNAPSHOT statement as shown in the following example:

```
CREATE SNAPSHOT emp FOR UPDATE
   AS SELECT * FROM scott.emp@sales.ny.com;
```

Restrictions on Updatable Snapshots

Declarative constraints on snapshots and snapshot logs are not supported.

Snapshots of LONG columns are not supported.

Updatable snapshots must be simple snapshots; that is each row in the snapshot is based on a single row in a single remote table. A simple snapshot's defining query has no distinct or aggregate functions, GROUP BY or CONNECT BY clauses, subqueries, joins, or set operations.

Symmetric replication does not support replication of a subset of columns. All CREATE statements must be of the form

```
CREATE SNAPSHOT . . . FOR UPDATE
   AS SELECT * FROM . . .;
```

The following SQL statement is *not* supported:

```
CREATE SNAPSHOT . . . FOR UPDATE
   AS SELECT empno, ename FROM . . .;
```

Naming Updatable Snapshots

Naming conventions for updatable snapshots are the same as for read–only snapshots. See page page 3 – 4.

Privileges Required to Create Updatable Snapshots

To create an updatable snapshot, you must have the following sets of privileges:

- To create a snapshot in your own schema, you must have the CREATE SNAPSHOT, CREATE TABLE, CREATE TRIGGER, and CREATE VIEW system privileges, as well as SELECT privilege on the master tables.
- To create a snapshot in another user's schema, you must have the CREATE ANY SNAPSHOT system privilege, as well as SELECT privilege on the master table. Additionally, the owner of the snapshot must have been able to create the snapshot.
- To create the snapshot as a replicated object, you must have been granted EXECUTE privileges on SYS.DBMSOBJGWRAPPER at the master site.

Transaction Ownership

Under synchronous propagation, a transaction is owned by the owner of the trigger and will be propagated to the remote sit with that owner's privileges. You can change the ownership of the transaction, usually to the replication administrator who has full privileges at the remote site, by using GEN_OBJ_OWNER.

Dropping a Replicated Object

To drop a replicated object from a snapshot site, call the DROP_SNAPSHOT_REPOBJECT procedure in the DBMS_REPCAT package at that snapshot site, as shown in the following example:

```
DBMS_REPCAT.DROP_SNAPSHOT_REPOBJECT(
    sname        => 'acct_rec',
    oname        => 'emp',
    type         => 'snapshot',
    drop_objects => TRUE);
```

In this example, the EMP snapshot will no longer be replicated, all supporting objects will be dropped, and the snapshot itself will be dropped. Had DROP_OBJECTS been set to FALSE, the snapshot would no longer be replicated, but the snapshot would remain in the schema until you removed it using the DROP SNAPSHOT command. The trigger and associated package used to add transactions to the deferred transaction queue, as well as any queued transactions, are not dropped until you drop the snapshot itself.

Additional Information: The parameters for the DROP_SNAPSHOT_REPOBJECT procedure are described in Table 12 – 137, and the exceptions are listed in Table 12 – 138.

Dropping a Snapshot Object Group

To drop a snapshot object group from your replicated environment, call the DROP_SNAPSHOT_REPGROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.DROP_SNAPSHOT_REPGROUP(
    gname         => 'accts',
    drop_contents => TRUE);
```

In this example, the ACCTS replicated object group is dropped from your current snapshot site. All objects generated to support the replicated object group are dropped, and the replicated objects in the object group no longer propagate changes to their former master site.

To drop these objects completely, you must set DROP_CONTENTS to TRUE, as shown in the example. If you set DROP_CONTENTS to FALSE, the trigger generated to support replication of snapshot modifications remains.

Additional Information: The parameters for the DROP_SNAPSHOT_REPGROUP procedure are described in Table 12 – 135, and the exceptions are listed in Table 12 – 136.

Offline Instantiation

Offline instantiation of a snapshot site is primarily useful for those sites with a very large amount of snapshot data where the time required to transfer the data through network links to the new site would be prohibitive.

Creating a snapshot site using offline instantiation requires that you first create a snapshot for each table in a new snapshot replication group at the *master site*, then do an export of the base tables to a file or files that can then be transported (via tape or another medium) to the new site and used to instantiate the new snapshot site.

Perform the following steps at the specified sites:

Master Site:

- It is recommended that you create a snapshot log for each master site before instantiating the new snapshot. You must also be the owner of the schema at the master site from which the snapshots will be derived.
- Create a snapshot for each of the tables in the same schema as the master schema, but give the snapshot a name that is different from the corresponding master table. You must also include the database link from the snapshot site to the master site.
  For example:

  ```
  CREATE SNAPSHOT FOOITEM AS SELECT * FROM ioug1.item@dbs1
  ```

  would work, but

  ```
  CREATE SNAPSHOT FOOITEM AS SELECT * FROM ioug1.ITEM
  ```

  would fail.

☞ **Attention:** Before creating your snapshots, ensure that you have the necessary storage space available at the master site.

- As schema owner, use the Export utility to export the snapshots' base tables (those prefixed with SNAP$_). The export file(s) can then be transported to the new snapshot site.

New Snapshot Site:

- Mount the tape containing the export file.
- Use the procedure CREATE_SNAPSHOT_REPGROUP (gname, master_site) to create the replicated object group for the snapshot that you plan to import from the master site. Note that a replicated snapshot group must have the same name as the object group at the master site.
- For each schema and snapshot, use the procedure DBMS_OFFLINE_SNAPSHOT.BEGIN_LOAD (gname, sname, snapshot_oname, master_site) to create empty snapshots in the specified schema and object group, as well as all the necessary supporting objects.
- You can now import the base tables from the Export file(s).
- When the import is complete, for each schema and snapshot, use the procedure DBMS_OFFLINE_SNAPSHOT.END_LOAD (gname, sname, snapshot_oname).

Master Site:

- Use the DROP SNAPSHOT statement to drop the snapshots at the master site that were created for offline instantiation.

For more information about using the Import/Export utilities, see page 7 – 12 and *Oracle7 Server Utilities*.

Propagating DML Changes

This section describes how updates made to a replicated snapshot are propagated to its associated master table, and how updates to the master table are, in turn, propagated to the snapshots. While master table changes are always propagated to the snapshot site asynchronously, in the form of a refresh, snapshot site changes can be propagated either synchronously or asynchronously. The PROPAGATION_MODE parameter of the CREATE_SNAPSHOT_REPGROUP and ALTER_SNAPSHOT_PROPAGATION commands determines how changes from the snapshot site are propagated to the master site.

How Changes are Propagated

The method that you choose to refresh and/or propagate your snapshot updates will be determined by the frequency of changes that you make to the data at the snapshot and master sites.

For example, you might want communication from the snapshot to the master to seem event–driven. By frequently propagating changes to the master site (such as every 10 seconds), you can ensure that shortly after each modification to an updatable snapshot, the change is forwarded to its associated master table. Yet, you might only refresh the snapshot once, at the start of each day, or you may never refresh the snapshot, if updates are performed only at the snapshot site.

Asynchronously Replicating Snapshot Updates to the Master Site

As shown in Figure 5 – 1, whenever you apply a change to a replicated updatable snapshot that is asynchronously propagating changes to its associated master site, the following events occur:

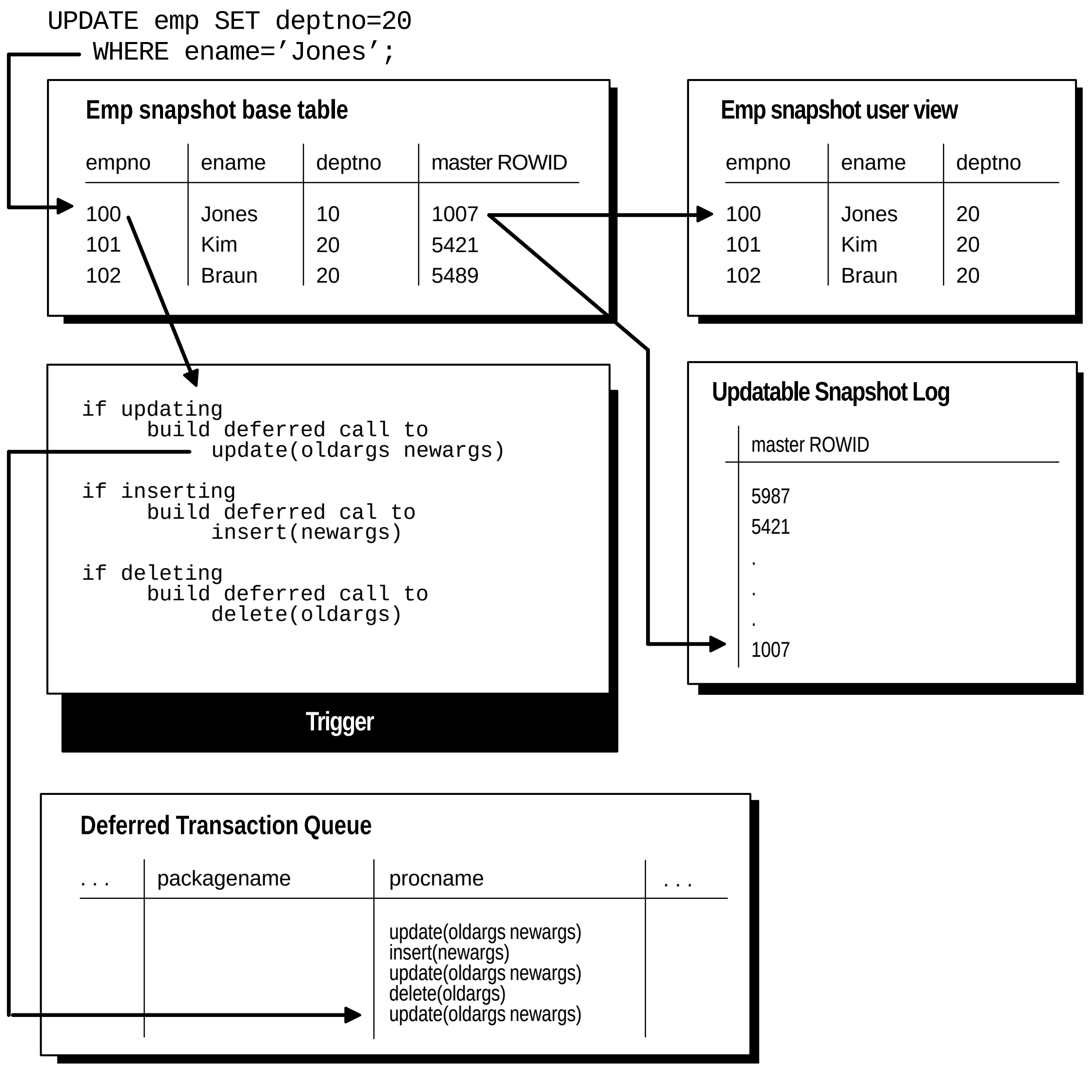


Figure 5 – 1 Applying Changes to an Updatable Snapshot

- The change is applied to the snapshot base table. This change is also reflected in the user view of the snapshot.
- The ROWID of any rows updated or deleted is recorded in the snapshot log. This information is used during a snapshot refresh to determine what changes to apply to the snapshot.
- A call is added to the deferred transaction queue. These calls are pushed to the associated master table at whatever interval you specify.

When Changes are Propagated

Updates are asynchronously propagated from the snapshot site to its master site whenever one of the following actions occurs:

- The updatable snapshot is refreshed by calling one of the following procedures, with the PUSH_DEFERRED_RPC argument set to TRUE:
  - DBMS_REFRESH.MAKE
  - DBMS_REFRESH.REFRESH
  - DBMS_SNAPSHOT.REFRESH

  Each of these methods of snapshot refresh is described, starting on page 3 – 15. You may also want to set the REFRESH_AFTER_ERRORS argument to TRUE if you want the refresh to continue even if there are errors logged in the DefError table at the master definition site.
- The deferred transaction queue at the snapshot site is pushed by calling either DBMS_DEFER_SYS.EXECUTE or DBMS_DEFER_SYS.SCHEDULE_EXECUTION. The deferred transaction queue at a snapshot site is pushed in the same manner as the queue at an asynchronous master site is pushed; see page 4 – 26 for details. For snapshot sites, you should set the BATCH_SIZE parameter to 0.

Conflict Detection

When you asynchronously push changes from a snapshot to its master, Oracle compares the old values for the row at the snapshot site (that is, the values before any changes were applied) to the current values for the row at the master site. If the values are different, a conflict has occurred. If any conflicts are detected at the master site, Oracle invokes the appropriate conflict resolution routine, if any was specified. For example, you might create a routine to resolve a conflict between two rows by selecting the row with the most recent timestamp, or by combining the column values of the conflicting rows.

Note that if conflict resolution routines are employed, the values of some of the rows in your snapshot may change or even be removed after a refresh is performed. If you did not specify a conflict resolution routine at the master site, or if the routine specified is unable to resolve the conflict, the conflict is logged and must be resolved manually at the master site.

Additional Information: For additional information on conflict detection and resolution, refer to Chapter 6.

How Snapshot Changes are Applied to the Master Table

The DML changes propagated to the master table are applied in the same manner as changes are replicated from one master site to another master site. As shown in Figure 5 – 2, the deferred call is pushed from the snapshot site. The package at the master site applies the change to the master table. If this change results in a conflict, it must either be resolved by the appropriate conflict resolution routine, or logged in an error table.

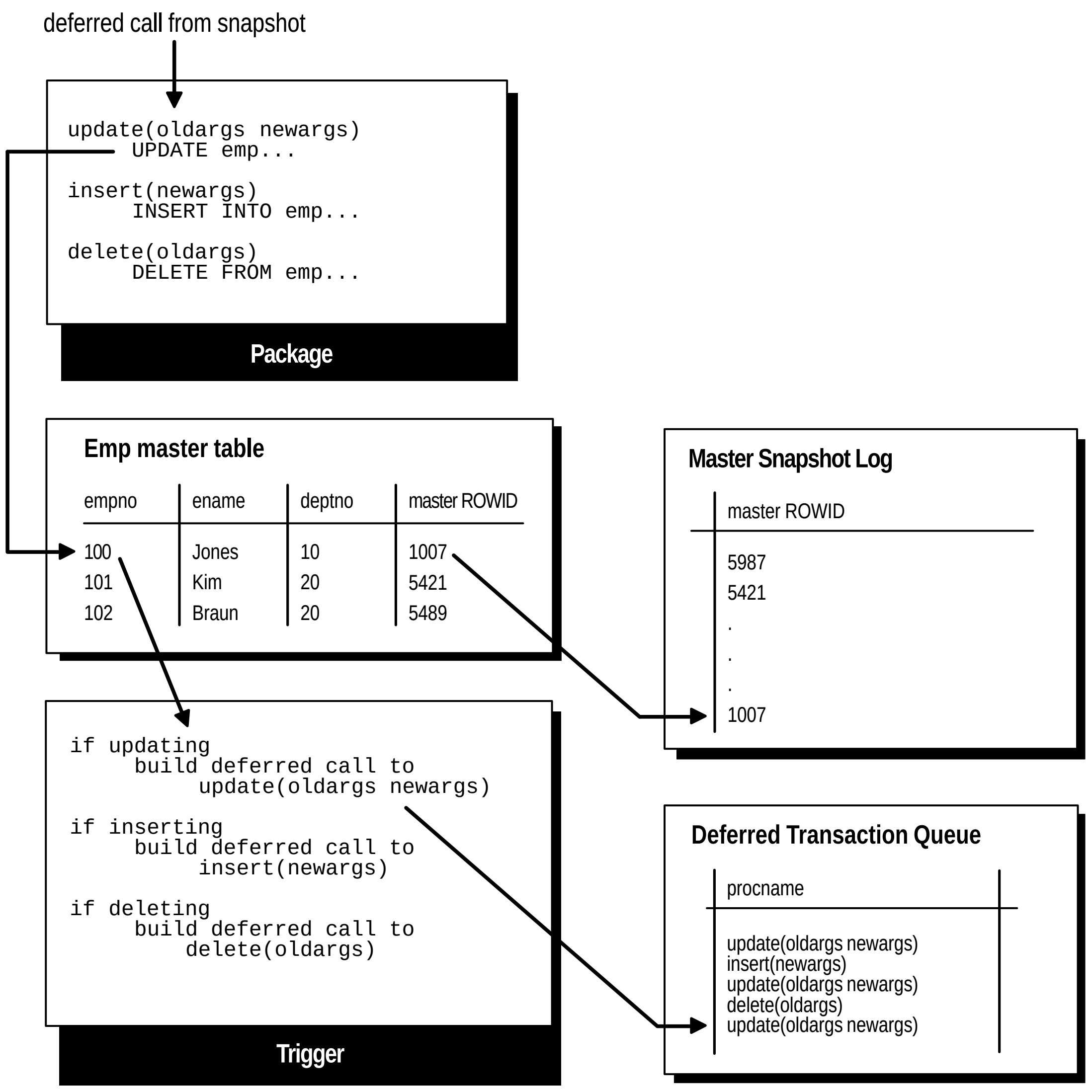


Figure 5 – 2 Applying Changes to a Master Table 3

Two additional steps occur when changes are propagated from an updatable snapshot to the master table:

- The ROWID of any rows inserted, updated, or deleted is logged in the master snapshot log. This information is used during a fast snapshot refresh to determine what changes to apply to the snapshot.

- A call is added to the deferred transaction queue at the master site. These calls are pushed to the other master tables in the replicated environment. It is this final step that ultimately allows changes from one snapshot site to propagate to all other sites in the replicated environment.

Because changes from the master site are applied at the snapshot site using the refresh mechanism, there is no need for conflict detection or resolution at the snapshot site. All conflict detection and resolution occurs at the master site.

Refreshing a Snapshot

As shown in Figure 5 – 3, when you refresh an updatable snapshot the following events occur:

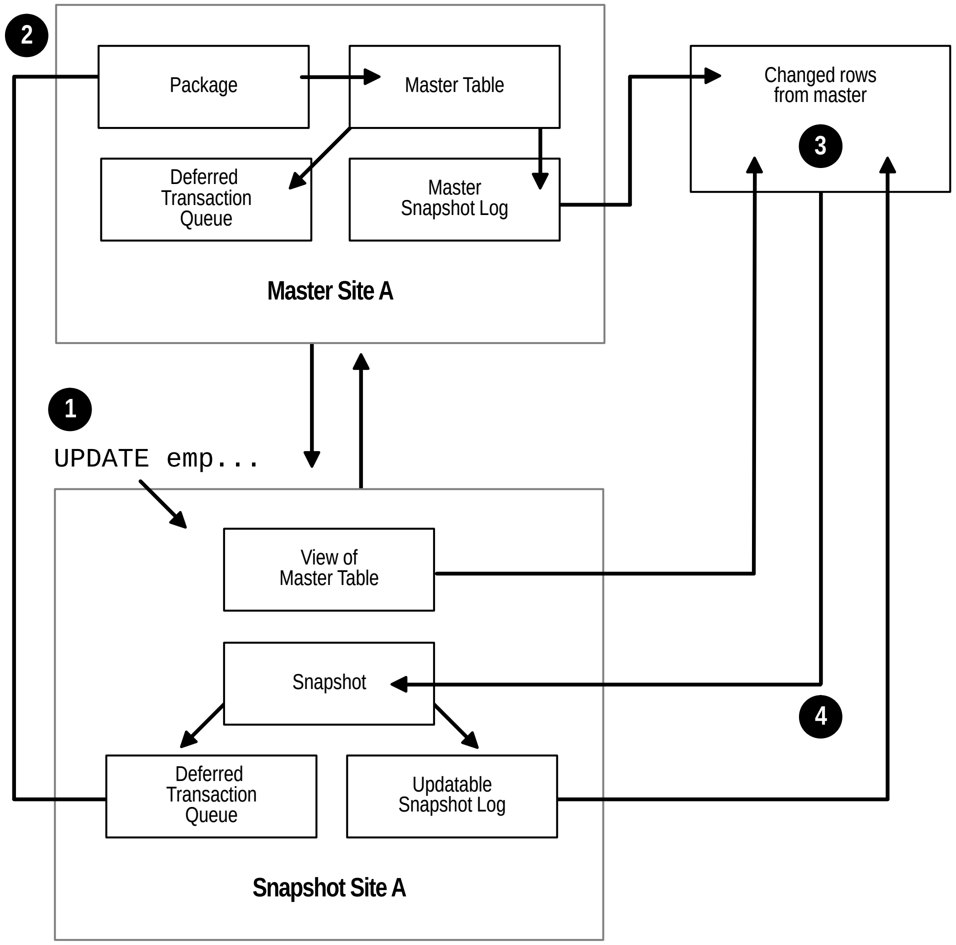


Figure 5 – 3 Snapshot Refresh

1. Any changes that you made to the snapshot site will have been added to the deferred transaction queue associated with the snapshot.

2. Any transactions queued for the snapshot are pushed to its associated master table (assuming PUSH_DEFERRED_RPC is TRUE), where they are applied, using the appropriate replication mechanism. If you used synchronous propagation for the snapshot, there should be not deferred transactions to propagate.

3. Oracle next determines which rows from the master table need to be updated at the snapshot. Conceptually, Oracle uses the view of the master table to create a list of the rows in the master snapshot log and the updatable snapshot log that are relevant for the updatable snapshot.

   Although you might think that all changes in the snapshot log would be recorded in the master log when the changes from the snapshot were pushed to the master, it is possible that a change to the snapshot would not be applied to the master. For example, if a conflict was detected when the change was pushed to the master, application of a conflict resolution routine might have resulted in a change not being applied to the master table.

4. These changes are applied to the snapshot by copying the values of the changed rows in the master table to the snapshot base table.

Propagating Changes Between Snapshot Sites

As shown in Figure 5 – 4, snapshot sites never communicate with one another directly. Instead, they must communicate through their associated master sites. In order for a snapshot to see a change made to a snapshot at another snapshot site, the following series of events illustrates what must occur:

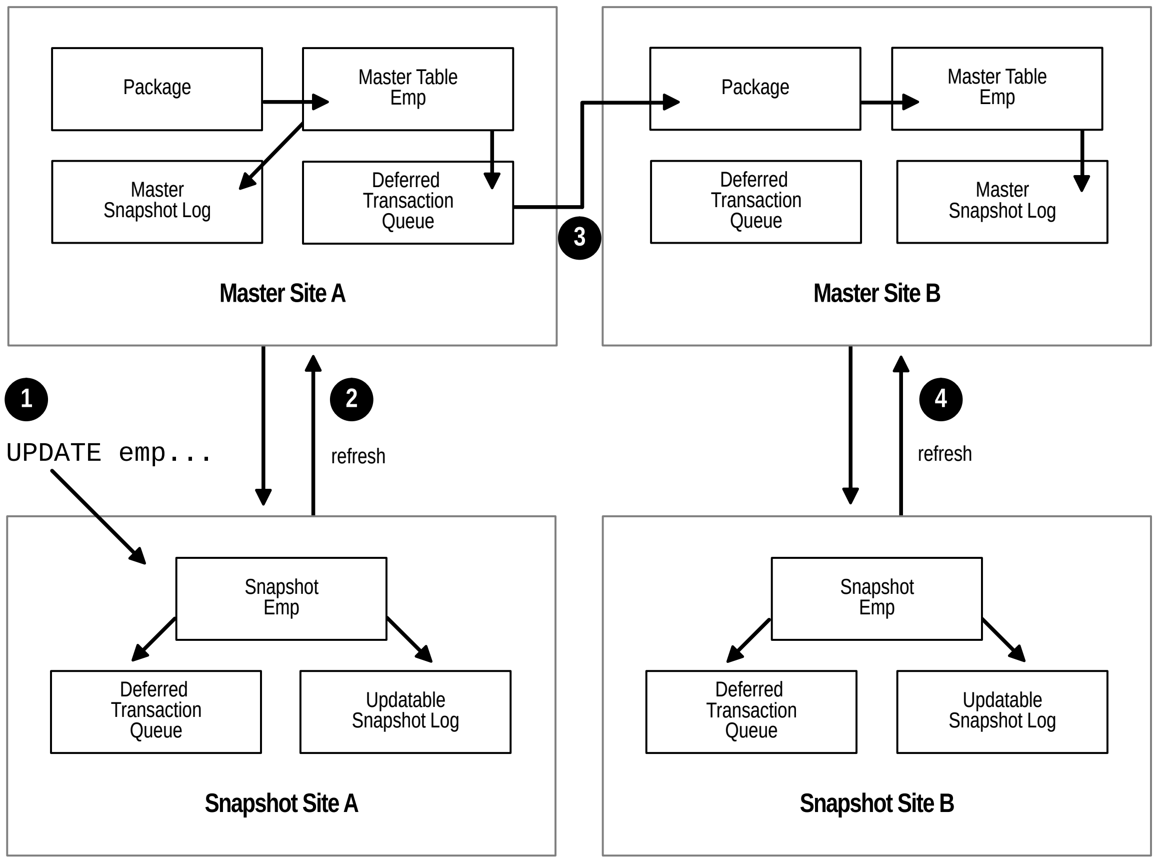


Figure 5 – 4 How Changes Propagate Between Snapshot Sites

1. The change is first applied to snapshot EMP at snapshot site A.
2. This change is either synchronously (using remote procedure calls) or asynchronously (by pushing the deferred transaction queue, for example as the result of a refresh) applied to the master table of snapshot EMP at site A. For changes propagated asynchronously, any conflicts between the snapshot and master table are detected and possibly resolved during this refresh.

3. Later when master site A communicates with master site B, the EMP table at site B is updated with this change. Any conflicts between the EMP table at site A and the EMP table at site B are detected and possibly resolved at this time.
4. Finally, the EMP snapshot at site B is updated with this change when it is refreshed.

Synchronously Replicating Snapshot Updates to the Master Site

If you choose to synchronously propagate your snapshot site changes to the associated master site, Oracle performs the following actions each time you modify an updatable snapshot:

1. Oracle obtains a lock on the local row and performs the update.
2. The AFTER ROW generated trigger for the snapshot fires, and its associated package makes the necessary remote procedure call to the generated package at the master site to apply the change.
3. The generated procedure at the master site locks the row at the master site and performs the update.
4. If the master detects a conflict that it cannot resolve, an error is raised immediately.
5. Using two–phase commit, Oracle commits or rolls–back (if an error occurred), the transaction at the snapshot and master sites and releases the locks.

The snapshot logs at both the updatable snapshot site and the master site, as well as the view and base table at the snapshot site, are updated in the same manner as if you had propagated the changes asynchronously.

Propagating DDL Changes

When you use the procedures in the DBMS_REPCAT package to make changes to your replicated environment, these changes are not visible to a snapshot site until you refresh the snapshot replicated object group. Refreshing a snapshot site updates the objects contained in the replicated object group, whereas refreshing a snapshot updates the values of the rows of a snapshot object in the replicated object group.

Refreshing a Snapshot Site

To refresh a snapshot site with the most recent information from its associated master site, call the REFRESH_SNAPSHOT_REPGROUP procedure in the DBMS_REPCAT package at that snapshot site, as shown in the following example:

```
DBMS_REPCAT.REFRESH_SNAPSHOT_REPGROUP(
    gname                 => 'accts',
    drop_missing_objects  => TRUE,
    refresh_snapshots     => TRUE,
    refresh_other_objects => TRUE
    execute_as_user       => FALSE);
```

In this example, all of the snapshots and other objects in the ACCTS object group of your current snapshot site will be refreshed, and any objects dropped at the associated master site will be dropped from the snapshot site. All snapshots in the replicated object group are consistently refreshed using the FORCE refresh option.

By default, only the RepGroup views for the given replicated object group are updated. Refreshing the RepGroup view implies that any changes made to the replicated environment, such as the addition of a new master site, or the removal of an object from the object group, are made visible to the snapshot site.

You can optionally choose to refresh the replicated snapshots and non–snapshot objects, such as views and procedures. Snapshots are refreshed as described on page 5 – 12. Oracle refreshes the non–snapshot objects, if necessary, by dropping and re–creating them using the definition of the object at the associated master site.

Additional Information: The parameters for the REFRESH_SNAPSHOT_REPGROUP procedure are described in Table 12 – 155, and the exceptions are listed in Table 12 – 156.

☞ **Attention:** During a snapshot refresh, Oracle locks the base table of the snapshot in exclusive mode. Because other users are prevented from *updating* a snapshot during a refresh (queries are still available), you should schedule your refreshes to occur when the expected activity on the snapshot is low.

Altering a Replicated Snapshot

If you must alter the shape of a snapshot as the result of a change to its master, you must drop and recreate the snapshot by calling the DBMS_REPCAT.DROP_SNAPSHOT_REPOBJECT procedure and then the DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT procedure.

Altering the Propagation Method of a Replicated Snapshot

To alter the method used to propagate changes from the snapshot site to its associated master site, use the DBMS_REPCAT.ALTER_SNAPSHOT_PROPAGATION procedure, as shown in the following example:

```
DBMS_REPCAT.ALTER_SNAPSHOT_PROPAGATION(
     gname             => 'accts',
     propagation_mode  => 'synchronous'
     execute_as_user   => 'REPADMIN');
```

In this example, all of the updatable snapshots in the ACCTS object group will now synchronously propagate their changes to their associated master tables.

When you call this procedure from the snapshot site, Oracle pushes the deferred transaction queue at the snapshot site, locks the snapshot base tables, and regenerates any triggers and their associated packages.

> **Additional Information:** The parameters for the ALTER_SNAPSHOT_PROPAGATION procedure are described in Table 12 – 84, and the exceptions are listed in Table 12 – 85.

Forcing Ownership of a Transaction

You can change the ownership of a transaction, usually to the replication administrator who has full privileges at the remote site, by using EXECUTE_AS_USER.

Listing Snapshot Information

Query the DBA_SNAPSHOTS catalog view to obtain a listing of all of the snapshots in a database. For information on snapshot refresh groups, query the USER_REFRESH and USER_REFRESH_CHILDREN views. Query DBA_SNAPSHOT_LOGS at the master site to see master log information.

Sample Application

Suppose that you have the following tables in your INVENTORY database: CUSTOMER, ORDERS, ORDER_LINE, ITEM, and STOCK.

Now suppose that you decide to replicate these tables to multiple sites. Because you have chosen to asynchronously propagate your changes between sites, you decide to avoid possible update conflicts by partitioning the ownership of the data based on workflow.

To partition ownership, you add a STATUS column to the ORDERS table. The status of an order can be: S (shippable), B (billable), O (outstanding bill), or C (complete).

This example has two order entry sites, so you must take steps to ensure either that conflicts do not occur, or that they can be resolved. In this example, all orders have a unique order ID.

The sequence used to generate this ID is partitioned between the master definition site and this snapshot site. Even if the same customer places one order at the master definition site and another at the snapshot site, each order will have a unique ID and will be treated separately. In this example, only one of the sites is allowed to update the CUSTOMER table, therefore no conflict resolution is required for this table.

This example assumes that the appropriate database links have been created at each site, and that the appropriate privileges have been granted to the replication administrator at each site.

To create your replicated environment, perform the following steps:

1. Start by creating your master definition site, which consists of the following steps:

   - Create the replicated object group by issuing the following command:

   ```
   DBMS_REPCAT.CREATE_MASTER_REPGROUP('inventory')
   ```

   - Then select the objects that you want to replicate. For each table, you should also provide a conflict resolution routine if necessary, and then generate the desired form of replication support.

```
 -- replicate customer table
DBMS_REPCAT.CREATE_MASTER_REPOBJECT('acct','customer',
                                    'table','inventory',);
-- insert appropriate calls to conflict resolution methods
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT('acct',
                                         'customer','table');

 -- replicate orders table
DBMS_REPCAT.CREATE_MASTER_REPOBJECT('acct','orders',
                                    'table','inventory');
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT('acct',
                                         'orders','table');

 -- replicate order_line table
DBMS_REPCAT.CREATE_MASTER_REPOBJECT('acct','order_line',
                                    'table','inventory');
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT('acct',
                                         'order_line','table');

 -- replicate item table
DBMS_REPCAT.CREATE_MASTER_REPOBJECT('acct',
                                    'item','table','inventory');
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT('acct','item',
                                         'table');
```

```
 -- replicate stock table
DBMS_REPCAT.CREATE_MASTER_REPOBJECT('acct','stock','table',
                                    'inventory');
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT('acct','stock',
                                         'table');
```

2. Now you can create another master site with this command:

```
DBMS_REPCAT.ADD_MASTER_DATABASE('inventory', 'dbs2');
```

   Oracle creates a replica of the master definition site at the database associated with the DBS2 link. This master site will be used by the shipping department only. Because no orders will be placed from this site, you do not have to worry about conflicts between this site and the master definition site, which is used for order entry.

3. After confirming that there are no errors in the replication log, you can now begin normal replication activity at these sites by issuing the following command:

```
DBMS_REPCAT.RESUME_MASTER_ACTIVITY('inventory');
```

4. You are now ready to begin creating your snapshot sites. This example assumes that you have already used the CREATE SNAPSHOT LOG command to create logs on your master tables. This allows you to perform a fast refresh on the snapshots you need to create. To create your billing server snapshot replication site, you would need to issue the following commands at the new snapshot site:

   - First, you need to indicate what object group you are replicating, and which master site you will be refreshing your snapshot site from.

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPGROUP('inventory', 'dbs2',
                                     'asynchronous')
```

   - Now you can populate your snapshot replication object group with a subset of the tables at its associated master site. Remember to use the FOR UPDATE clause of the CREATE SNAPSHOT command when creating updatable snapshots. Oracle automatically generates the appropriate replication support for these updatable snapshots based on the type of replication support that you generated for their associated master tables.

```
 -- create read-only snapshot of customer table
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
      'acct', 'customer', 'snapshot',
      'CREATE SNAPSHOT customer' ||
      'AS SELECT * FROM customer@dbs2','','inventory');
 –– create updatable snapshot of orders table
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
      ’acct’,’orders’,’snapshot’,
      ’CREATE SNAPSHOT orders FOR UPDATE AS’ ||
      ’SELECT * FROM orders@dbs2 WHERE status = ’’B’’’,
      ’’,’inventory’);

 –– create updatable snapshot of order_line table
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
      ’acct’, ’order_line’, ’snapshot’,
      ’CREATE SNAPSHOT order_line FOR UPDATE AS’ ||
      ’SELECT * FROM order_line@dbs2’,’’,’inventory’);

 –– create updatable snapshot of item table
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
      ’acct’, ’item’, ’snapshot’,
      ’CREATE SNAPSHOT item FOR UPDATE AS’ ||
      ’SELECT * FROM item@dbs2’,’’,’inventory’)
```

CHAPTER

6 Conflict Resolution

Oracle always detects and logs update conflicts, uniqueness conflicts, and delete conflicts. In addition, Oracle provides system–defined conflict resolution routines that enable an environment using asynchronous row–level replication to resolve update conflicts and uniqueness conflicts. You can also write your own conflict resolution routines.

This chapter covers the following topics:

- how conflicts are detected
- how column groups are used in conflict detection
- Oracle–supplied conflict resolution methods
- how to designate a conflict resolution method
- conflict notification
- selecting a conflict resolution strategy
- using column groups
- using priority groups
- using site priority

Additional Information: For an overview of conflict resolution routines, see Writing a Conflict Resolution Routine on page 8 – 2.

When to Use Conflict Resolution Methods

The purposes of conflict resolution are

- to ensure data convergence
- to avoid cascading errors

Convergence ensures that all sites in your replicated environment agree and have the same data. Avoiding cascading errors ensures that your system will run smoothly.

☞ **Attention:** If one (or more) rows in a transaction causes a conflict that remains unresolved, the *entire* transaction is written to the error log. Subsequent transactions that depend on the original transaction can now conflict, and in turn, be written to the error log.

If one or more sites in your replicated environment propagate changes *asynchronously,* conflicts can occur if two or more sites update the same replicated data. Even if your environment is designed to avoid conflicts (for example, by partitioning data ownership), it is prudent to

- monitor the occurrence of any unresolved conflicts
- use a notification method to learn of any unexpected conflicts

Note: If you do not designate a conflict resolution method, Oracle logs any unresolved conflicts as error transactions in the DefError view of the receiving site.

If all of your sites propagate changes *synchronously* and you have no updatable snapshot sites, conflicting updates cannot occur, and you do not need to designate a conflict resolution method.

Detecting Conflicts

When you propagate changes in a replicated environment by pushing the deferred transaction queue, Oracle's symmetric replication facility calls a remote procedure in the generated package at the receiving site. Oracle uses the remote procedure to detect conflicts, if any. If, for example, two sites modify the same row before propagating their updates to each other, a conflict occurs. (If you have set up a conflict resolution method, Oracle attempts to resolve the conflict.)

For each changed row, Oracle forwards

- the old value of each column in the row (the value before you modified the row)
- the new value of each column

> **Note:** If you are inserting a row, it has no *old* value. If you are deleting a row, it has no *new* value.

As shown in Figure 6 – 1, Oracle at the receiving site compares the old and current values of the row. Oracle detects a conflict if there are any differences between these values for any column in the row.

> **Note:** Because a row can have different ROWIDs at different sites, Oracle uses the row's primary key to determine which rows to compare. If you do not want to use the primary key, designate one or more different columns by calling DBMS_REPCAT.SET_COLUMNS, as described on page 4 – 16.

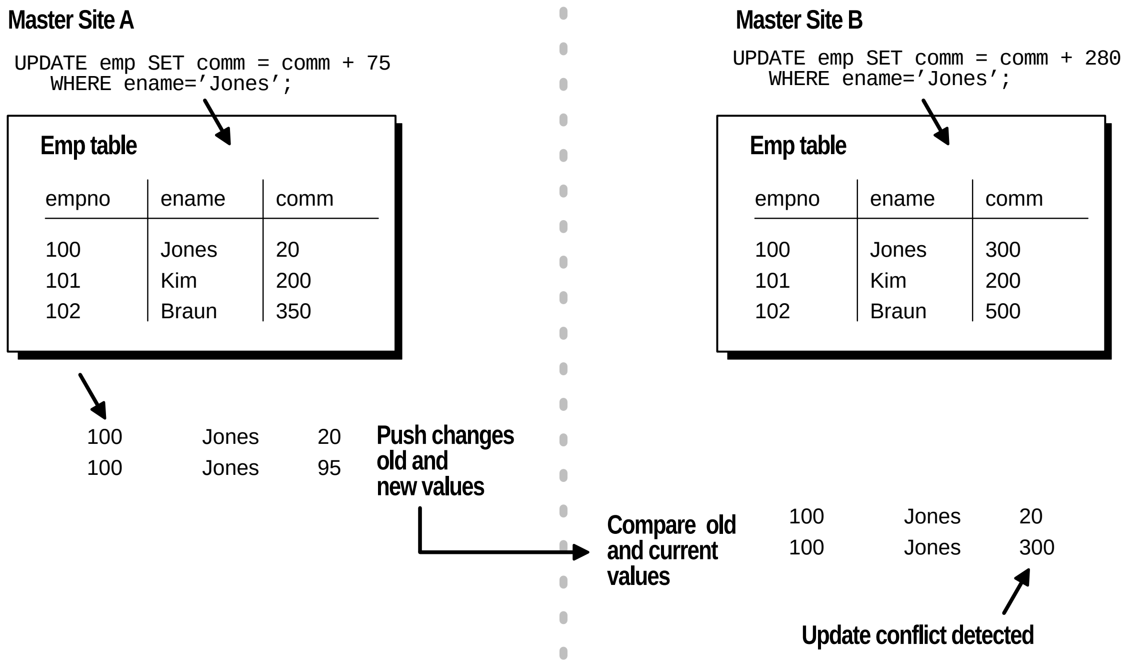


Figure 6 – 1 Detecting Conflicts

If the procedure at the receiving site detects no conflict, the server at the receiving site writes the new value(s).

If a conflict is detected, Oracle applies the appropriate conflict resolution routine, if one is available. Any unresolved conflicts are logged in the DefError view at the receiving site.

When you replicate a table using *row–level replication*, you can designate one or more conflict resolution methods. Oracle applies these methods in the priority order you define until the conflict is resolved, or no more routines are available.

> **Note:** For *procedural replication*, you must supply a conflict resolution method as part of your replicated procedure.

Types of Conflicts

The three types of conflicts that the symmetric replication facility detects are

- update conflicts
- uniqueness conflicts
- delete conflicts

The procedures at the receiving site detect an *update conflict* if there is any difference between the old values of the replicated row and the current values of the same row at the receiving site.

A *uniqueness conflict* is detected if a unique constraint is violated during an INSERT or UPDATE of the replicated row.

A *delete conflict* is detected if you change a row at a remote site after you delete that row from the local site. The delete conflict occurs because the old values of the deleted row at the local site do not match the current values of the same row at the remote site.

> **Warning:** Because the primary key is used to determine which rows to compare, allowing modifications to the primary key is extremely risky to the integrity of your data.

Understanding Column Groups

The symmetric replication facility uses a *column group* to detect and resolve update conflicts. A column group links a collection of columns in a table to a single "logical column". A column group can consist of a single column, any number of columns, or all of the columns in a table. Each column, however, can belong to only one column group.

The conflict detection mechanism detects update conflict *column group by column group*, so all columns must be a part of some column group.

You do not have to assign all of the columns in a table to a column group. However, you can only designate a conflict resolution method for the columns you assign to a column group.

Shadow Column Groups

Any column that you do not assign to a column group is automatically assigned to a “shadow” column group for conflict detection. A shadow column group is not visible to the user. You cannot assign a conflict resolution method to the columns in a shadow group. *Do not use a shadow group for columns if you expect conflicts to occur on those columns.*

Designating a Conflict Resolution Method

Having column groups allows you to designate different methods of resolving conflicts for different types of data. For example, numeric data is often suited for an arithmetical resolution method, and character data is often suited for a timestamp resolution method.

Data Convergence vs Data Integrity

Oracle evaluates each column group individually, so some portions of a row can be updated using the data from the originating site, while other portions can maintain the values of the data at the destination site. When you use multiple column groups, a conflict resolution mechanism can result in *data convergence* (all sites having the same values for a given row) without necessarily resulting in *data integrity* (data convergence on the appropriate value). For example, if the zip code column uses the *latest timestamp* resolution method while the city column uses the *site priority* resolution method, all sites could converge on a zip code that does match the city.

☞ **Attention:** If two or more columns in a table must remain *consistent with respect to each other,* place these columns within the same column group.

Detecting Update Conflicts in a Column Group

When examining a row to determine if an update conflict has occurred, the replication facility uses the following algorithm:

- Starting with the first column group, examine each field to determine if it has changed and, if so, if there is a conflict between the old, new, and current values.
- If no conflict occurred, continue with the next column group. If a conflict occurred, call the conflict resolution routine with the lowest assigned sequence number for that column group.
- If the conflict resolution routine successfully resolves the conflict, hold the appropriate values for the columns pending determination of status.

- If the routine cannot resolve the conflict, the replication facility continues with the next priority routine in the sequence you assign until the conflict is resolved or no more routines are available.
- After evaluating all column groups (including the shadow column group) and successfully resolving any errors, the symmetric replication facility stores the new values for the columns.
- If the replication facility is unable to resolve the conflict for a column group (including the shadow column group, which has no designated conflict resolution method), it logs an error in the DefError view and does not change the local row.

Selecting a Conflict Resolution Method

The symmetric replication facility's declarative conflict resolution mechanism provides system–defined routines for resolving update and uniqueness conflicts.

These system–defined routines do not support the following situations:

- delete conflicts
- changes to primary key columns
- NULLs in the columns that you designate to resolve the conflict
- referential integrity constraint violations

For these situations, either provide your own conflict resolution routine or determine a method of resolving these errors after they are logged in the DefError view.

System–Defined Resolution Routines for Update Conflicts

The table below specifies the system–defined resolution methods for update conflicts that guarantee convergence in three types of replication environments.

Any Number of Master Sites (4 methods)	**One or Two Master Sites (9 methods)**	**One Master Site and Multiple Snapshots Sites (12 methods)**
latest timestamp		
additive		
minimum value (always decreasing)		
maximum value (always increasing)		
	earliest timestamp	
	minimum value	
	maximum value	
	highest priority site	
	highest priority value	
		average
		discard from snapshot sites
		overwrite master site

Table 6 – 1 System Defined Methods to Resolve Update Conflicts

> **Note:** The conflict resolution methods you assign need to ensure data convergence and provide results that are appropriate for how your business uses the data.

System–Defined Resolution Routines for Unique Constraint Conflicts

The symmetric replication facility provides three methods for resolving uniqueness conflicts:

- append the global name of the originating site to the column value from the originating site
- append a generated sequence number to the column value from the originating site
- discard the row value from the originating site

Avoiding Ordering Conflicts

If you have more than one master site, none of these routines result in convergence, and these routines should only be used in conjunction with a notification facility. See the Conflict Notification section on page 6 – 18.

If your replicated environment has more than two masters:

- the following conflict resolution methods cannot guarantee convergence: *earliest timestamp, minimum value, maximum value, highest priority site, highest priority value*
- network failures and infrequent pushing of the deferred remote procedure call (RPC) queue increase the likelihood of non–convergence for these methods

Table 6 – 2 shows how having three master sites can lead to ordering conflicts. Master Site A has priority 30; Master Site B has priority 25; and Master Site C has priority 10; *x* is a column of a particular row in a column group that is assigned the *site–priority* conflict resolution method.

> **Note:** The highest priority is given to the site with the highest priority value. Priority values can be any Oracle number and do not have to be consecutive integers.

Time	Action	Site A	Site B	Site C
1	All sites are up and agree that x = 2.	2	2	2
2	Site A updates x = 5.	5	2	2
3	Site C becomes unavailable.	5	2	down
4	Site A pushes update to Site B. Site A and Site B agree that x = 5. Site C is still unavailable. The update transaction remains in the queue at Site A.	5	5	down
5	Site C becomes available with x = 2. Sites A and B agree that x = 5.	5	5	2
6	Site B updates x = 5 to x = 7.	5	7	2
7	Site B pushes the transaction to Site A. Sites A and B agree that x = 7. Site C still says x = 2.	7	7	2
8	Site B pushes the transaction to Site C. Site C says the old value of x = 2; Site B says the old value of x = 5. Oracle detects a conflict and resolves it by applying the update from Site B, which has a higher priority level (25) than Site C (10). All site agree that x = 7.	7	7	7
9	Site A successfully pushes its transaction (x = 5) to Site C. Oracle detects a conflict because the current value at Site C (x = 7) does not match the old value at Site A (x = 2). Site A has a higher priority (30) than Site C (10). Oracle resolves the conflict by applying the outdated update from Site A (x = 5). Because of this ordering conflict, the sites no longer converge.	7	7	5

Table 6 – 2 Ordering Conflicts With Site Priority ***– More Than Two Masters***

You can guarantee convergence when using priority groups if you require that the flow of ownership be ordered. For example, the workflow model dictates that information flow one–way through a three–step sequence:

1. From the ORDERING site
2. to the SHIPPING site
3. to the BILLING site.

If the billing site receives a change to a row from the ordering site after the billing site received a change to that row from the shipping site, the billing site ignores the out–of–order change because the change from shipping has a higher priority.

Suggestion: To help determine which conflict resolution method to use, make a diagram or time–action table (such as Table 6 – 2) to help uncover any potential loopholes in your conflict resolution methodology.

Why Use Multiple Resolution Methods

The "Understanding Column Groups" section of this chapter explained that multiple column groups provide *multiple conflict resolution methods* for a single row. You can also use *multiple conflict resolution methods* for each column group. Use multiple resolutions methods

- to have one or more backup methods
- to receive notification of conflicts

Multiple resolution methods are applied in the sequence you set.

Your preferred conflict resolution method might not always succeed. You can specify a backup method to have a greater chance of conflict resolution without manual intervention.

Some system–defined resolution methods, such as latest timestamp, occasionally require a backup method to successfully resolve conflicts. (Site priority is a possible backup method.) The latest timestamp method uses a special timestamp column to determine and apply the most recent change. In the unlikely event that the row at the originating site and the row at another site change at precisely the same second, you must provide a backup method.

Note: Oracle stores time to a granularity of one second.

You can also provide a user–defined method that records conflict information or notifies the DBA if the conflict cannot be resolved. You can arrange to receive notification for all conflicts, or for only those conflicts that cannot be resolved. You can mix any number of user–defined and system–defined conflict resolution routines.

Summary of Standard Conflict Resolution Methods

Convergence means that all sites ultimately agree on the same value. Table 6 – 3 summarizes the system–defined conflict resolution methods, and whether they guarantee convergence between multiple master sites and their associated snapshot sites. Following sections describe each of these methods in greater detail.

Conflict Type	Resolution Methods	Converge?	Converge with >2 Masters?
UPDATE	minimum	yes	no, unless always decreasing
	maximum	yes	no, unless always increasing
	earliest timestamp	yes	no
	latest timestamp	yes	yes, with backup method
	priority group	yes	no, unless always increasing
	site priority	yes	no
	overwrite	yes, 1 master only	no
	discard	yes, 1 master only	no
	average (numeric only)	yes, 1 master only	no
	additive (numeric only, additive updates only)	yes	yes
Uniqueness Constraint (UPDATE or INSERT)	append site name	no	no
	append sequence	no	no
	ignore discard	no	no
DELETE	none		

Table 6 – 3 Standard Conflict Resolution Methods

Minimum and Maximum Update Conflict Resolution Methods

When the symmetric replication facility detects a conflict with a column group and calls the *minimum* value conflict resolution routine, it compares the new value from the originating site with the current value from the destination site for a designated column in the column group. You must designate this column when you select the minimum value conflict resolution routine.

If the new value of the designated column is *less than* the current value, the column group values from the originating site are applied at the destination site (assuming that all other errors were successfully resolved for the row). If the new value of the designated column is greater than the current value, the conflict is resolved by leaving the current values of the column group unchanged.

> **Note:** If the two values for the designated column are the same (for example, if the designated column was not the column causing the conflict), the conflict is not resolved, and the values of the columns in the column group remain unchanged. Designate a backup conflict resolution method to be used in this case.

The *maximum* value method is the same as the minimum value method, except that the values from the originating site are only applied if the value of the designated column at the originating site is *greater than* the value of the designated column at the destination site.

There are no restrictions on the datatypes of the columns in the column group. Convergence for more than two master sites is only guaranteed if

- for the maximum method, the column value is always increasing
- for the minimum method, the column value is always decreasing

> **Note:** You should not enforce an always–increasing restriction by using a check constraint because the constraint could interfere with conflict resolution.

Earliest and Latest Timestamp Update Conflict Resolution Methods

The earliest timestamp and latest timestamp methods are variations on the minimum and maximum value methods. For the timestamp method, the designated column must be of type DATE. Whenever any column in a column group is updated, your application should update the value of this timestamp column with the local SYSDATE. For a change applied from another site, the timestamp value should be set to the timestamp value from the originating site.

Consider this sequence of events:

1. A customer in Phoenix calls the local salesperson and updates her address information.
2. After hanging up the phone, the customer realizes that she gave the local salesperson the wrong postal code.
3. The customer tries to call the local salesperson with the correct postal code, but the salesperson cannot be reached.
4. The customer calls the headquarters, which is located in New York. The New York site, rather than the Phoenix site, correctly updates the address information.
5. The network connecting New York headquarters with the local Phoenix sales site goes down temporarily.
6. When the New York / Phoenix network connection comes back up, Oracle sees two updates for the same address, and detects a conflict at each site.
7. Using the *latest timestamp* method, Oracle selects the most recent update, and applies the address with the correct postal code.

> **Note:** If your replicated environment crosses time zones, your application should convert all timestamps to a common time zone. Otherwise, although your data will converge, you may not apply the most *recent* update.

Oracle does not enforce time synchronization, which should be provided by another mechanism.

> **Note:** A sample timestamp and site site maintenance trigger is shown in the Example section (see page 6 – 44).

The earliest timestamp method applies the changes from the site with the earliest timestamp, and the latest timestamp method applies the changes from the site with the latest timestamp.

Suggestion: Designate a backup method, such as *site priority*, to be called if two sites have the same timestamp. Standardize your timestamping mechanism; for example, you can convert the timestamp to a designated time zone, such as Greenwich Mean Time (GMT).

A clock counts seconds as an increasing value. Assuming that you have properly designed your timestamping mechanism and established a backup method in case two sites have the same timestamp, the *latest* timestamp method (like the maximum value method) guarantees convergence. The *earliest* timestamp method, however, *cannot* guarantee convergence for more than two masters.

Additive and Average Update Conflict Resolution Methods

The additive and average routines work with column groups consisting of a single numeric column only.

The additive routine adds the difference between the old and new values at the originating site to the current value at the destination site.

current value = current value + (new value – old value)

The additive conflict resolution method provides convergence for any number of master sites.

The average conflict resolution method averages the new column value from the originating site with the current value at the destination site.

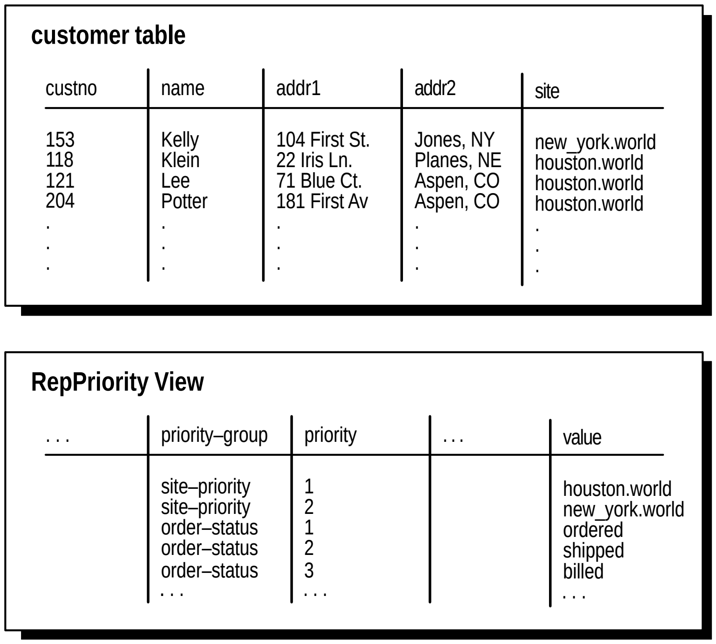
current value = (current value + new value)/2

The average method cannot guarantee convergence if your replicated environment has more than one master. This method is useful for an environment with a single master site and multiple updatable snapshots.

Priority Group and Site Priority Update Conflict Resolution Methods

Priority groups allow you to assign a priority level to each possible value of a particular column. If a conflict is detected, the table whose “priority” column has a lower value will be updated using the data from the table with the higher priority value.

As shown in Figure 6 – 2, the RepPriorityView displays the priority level assigned to each value that the “priority” column can contain. You must specify a priority level for all possible values of the “priority” column.



customer table

custno	name	addr1	addr2	site
153	Kelly	104 First St.	Jones, NY	new_york.world
118	Klein	22 Iris Ln.	Planes, NE	houston.world
121	Lee	71 Blue Ct.	Aspen, CO	houston.world
204	Potter	181 First Av	Aspen, CO	houston.world
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

RepPriority View

. . .	priority–group	priority	. . .	value
	site–priority	1		houston.world
	site–priority	2		new_york.world
	order–status	1		ordered
	order–status	2		shipped
	order–status	3		billed
	. . .	. . .		. . .

Figure 6 – 2 Using Priority Groups

When you select the priority group method of conflict resolution, you must designate which column in your table is the “priority” column.

The *RepPriority* view displays the values of all priority groups defined at the current location. In the example shown in Figure 6 – 2, there are two different priority groups, site–priority and order–status. The *customer* table is using the site–priority priority group.

Site priority is a special kind of priority group. With site priority, the “priority” column that you designate is automatically updated with the global database name of the site where the update originated. The RepPriorityView displays the priority level assigned to each database site. Site priority can be useful if one site is considered to be more likely to have the most accurate information.

For example, in Figure 6 – 2, the New York site (priority value = 2) is corporate headquarters, while the Houston site (priority value = 1) is a sales office.

The headquarters office (New York = 2) is considered more likely than the sales office (Houston = 1) to have the most accurate information about the credit that can be extended to each customer.

> **Note:** The priority–group column of the RepPriority view shows both the site–priority group and the order–status group.

When you are using site priority, convergence with more than two masters is not guaranteed. You can guarantee convergence with more than two masters when you are using priority groups, however, if the value of the "priority" column is always increasing. That is, the values in the priority column correspond to an ordered sequence of events; for example: ordered, shipped, billed.

Overwrite and Discard Update Conflict Resolution Methods

The overwrite and discard methods ignore the values from either the originating or destination site and therefore can never guarantee convergence with more than one master site. These methods are designed to be used by a single master site and multiple snapshot sites, or with some form of a user–defined notification facility.

For example, if you have a single master site that you expect to be used primarily for queries, with all updates being performed at the snapshot sites, you might select the overwrite method. The overwrite and discard methods are also useful if:

- your primary concern is data convergence
- you have a single master site
- there is no particular business rule for selecting one update over the other

or if

- you have multiple master sites
- you supply a notification facility to notify the person who ensures that data is correctly applied, instead of logging the conflict in the DefError view and leaving the resolution to your local database administrator

The overwrite routine replaces the current value at the destination site with the new value from the originating site. Conversely, the discard method ignores the new value from the originating site.

Append Site Name/Append Sequence Uniqueness Conflict Resolution Methods

The append site name and append sequence routines work by appending a string to a column that is generating a DUP_VAL_ON_INDEX exception. Although this allows the column to be inserted or updated without violating a unique integrity constraint, it does not provide any form of convergence between multiple master sites. The resulting discrepancies must be manually resolved; therefore, these methods are meant to be used with some form of a notification facility (see Conflict Notification on page 6 – 18). Both append site name and append sequence can be used on character columns only.

These methods can be useful when the availability of the data may be more important than the complete accuracy of the data. To allow data to be available as soon as it is replicated

- select append site name or append sequence
- use a notification scheme to alert the appropriate person to resolve the duplication, instead of logging a conflict

When a uniqueness conflict occurs, the *append site name* routine appends the global database name of the site originating the transaction to the replicated column value. The name is appended to the first period (.). For example, HOUSTON.WORLD becomes HOUSTON.

Note: Similarly, the *append sequence* routine appends a generated sequence number to the column value. The column value is truncated as needed. If the generated portion of the column value exceeds the column length, the conflict routine does not resolve the error.

Discard Uniqueness Conflict Resolution Method

The *discard uniqueness* conflict resolution routine resolves uniqueness conflicts by simply discarding the row from the originating site that caused the error. This method does not guarantees convergence with multiple masters and should be used with a notification facility.

Unlike the append methods, the discard uniqueness method minimizes the propagation of data until data accuracy can be verified.

Conflict Notification

A conflict notification routine is a user–defined conflict resolution routine that provides notification, rather than resolution. You can have conflict information logged in a database view, or you can write a procedure that, for example, sends an e–mail message to the DBA (or dials a beeper).

You can set up notification to occur when you want it:

- as soon as Oracle detects a conflict
- before Oracle attempts a specific resolution routine
- only if Oracle cannot resolve the conflict

Suggestion: To define the conflict notification routine(s) for your site, use the example on page 6 – 38 . Modify and/or expand upon the example with your own conflict resolution routines.

Note: If the conflict cannot ultimately be resolved, the entire transaction, including any updates to a notification table, will be rolled back. You can design your notification mechanism to use the Oracle DBMS_PIPES package or the interface to Oracle Office to ensure that notification occurs.

Declaring a Conflict Resolution Method

As you create a replicated table, you should designate one or more methods to resolve any potential conflicts.

To declare a conflict resolution method, first complete the planning phase:

- Analyze your data to determine which *column groups* are appropriate, and which *conflict resolution methods* are appropriate for each column group.
- Create columns, such as timestamp, and maintenance triggers as needed by the methods that you select.
- If desired, define notification packages.
  - create a table to hold conflict notification information at each master site
  - create the PL/SQL procedure to record conflict notification in the table

  - add user defined conflict resolutions to the package or add routines to automate e–mail notification (optional)

- If any column groups in any table will use site priority or priority groups for conflict resolution, define the priority levels for each site or value.

After planning, call the appropriate procedures in the DBMS_REPCAT package:

1. If you are adding conflict resolution to an existing replicated environment, you must first suspend all replication activity, as described on page 4 – 43

   If you are creating a new replication group, follow the instructions for creating a replication group and creating replicated objects beginning, on page 4 – 13.

2. Define the column groups for each table.

3. Distribute the conflict notification package and notification log table to the remote sites by registering it as a replicated object in the same object groups as the table monitored for conflicts (optional)

   Note: Do not use this option if each remote site is to customize the notification package.

4. Assign one or more conflict resolution methods for each column group.

5. Generate support for the replicated tables, as described on page 4 – 21.

   Caution: Do *not* generate replication support for the conflict notification table. The data in this table must remain specific to its site.

6. Create a timestamp and maintenance trigger (if needed).

7. Resume replication activity, as described on page 4 – 44.

Suggestion: To define the conflict notification routine(s) for your site, use the example on page 6 – 38 . Modify and / or expand upon the example with your own conflict resolution routines.

Developing a Conflict Resolution Strategy

Before selecting or writing a conflict resolution routine, you should first ensure that you have done everything possible to avoid the conflict in the first place. This section outlines how to

- identify and avoid conflicts
- select an appropriate conflict resolution routine

Define Functional Boundaries

When designing a replicated environment, the guidelines to good single–database schema design apply:

- the application should be modular, with functional boundaries and dependencies that are clearly defined (for example, order–entry, shipping, billing)
- data should be normalized to reduce the amount of hidden dependencies between modules.

In addition, to reduce the potential for conflicts, consider using

- a basic primary site model for data shared between modules, which allows only one module to update the data, while other modules read the data
- an advanced primary site model, where ownership of the data is horizontally partitioned: for example, the server in New York owns customers in New York, and the server in California owns customers in California

Use Generated Primary Key

Use generated sequence numbers for the primary key of each table. By using unique sequence numbers at each site, you can avoid uniqueness conflicts and determine ownership of rows based on the primary key. Although you could simply partition the sequence numbers among the sites, this can become problematic as the number of sites, or number of entries, grows. Instead, allow each site to use the full range of sequence values, and include a unique site identifier as part of the primary key.

Conflict Resolution Methods for Dynamic Ownership

If primary site ownership or distributed access to the data is not appropriate, consider dynamic ownership of data. Dynamic ownership permits only one database (the owner) to update the data at a time. Ownership of the data is allowed to move between sites, but only in a way that guarantees that the owner has the most recent data. Non–owners can have out–of–date data, and ordering conflicts can occur, but such conflicts are easily and correctly resolved using a method such as ”priority group” or ”maximum.” Note that single–master methods, such as ”overwrite”, would result in inconsistencies.

Dynamic ownership is most useful in cases in which

- correctness of data is crucial (such as, salary), and
- there is low contention of data, or
- there is reference locality (that is, the site that most recently updated the data is the most likely site to do the next update to the data).

Additional Information: See page 8 – 9 for more information on dynamic ownership.

Using Timestamp Resolution Method

Dynamic ownership is unnecessarily restrictive for many types of data. Data such as ”date–of–birth” or ”address” are rarely crucial to the correct operation of an application. Once this information is inserted into the database, it is rarely updated. Therefore the probability of a conflict is very low. Furthermore, the real world often has checks–and–balances (such as a forwarding address) that can compensate for slightly out–of–date information. Often, you can resolve conflicts with these types of data by using the ”latest timestamp” method. (Designate a backup method, such as site priority, in case of identical timestamps.)

The timestamp method is particularly useful because the data will converge regardless of the number of sites, but special care must be taken:

- synchronize clocks
- use consistent time zones
- ensure increasing timestamps for local updates

User–Defined Methods

The timestamp method is not appropriate for all data with shared ownership. User–provided conflict resolution routines can be used when the semantics of how data is used do not match those provided by Oracle’s predefined conflict resolution routines. User–provided routines can also be used for monitoring and notification in case of conflicts.

Note: The conflict resolution methods you assign need to ensure data convergence and provide results that are appropriate for how your business uses the data.

Avoid Deletes

The replicated application should not overuse deletes. Conflicts involving deletes are difficult to resolve because they require a history about deleted rows. Oracle symmetric replication does not maintain this history.

Instead, the application should mark a row as deleted (for example, by using a timestamp column that is filled in only upon delete). Periodically, the rows marked as deleted can be purged from the system using procedural replication, as described on page 8 – 20.

Setting the Propagation Interval

Where conflicts are possible, define a propagation interval that is less than the average interval between updates to the same row. Use small propagation intervals to minimize the probability of conflicts.

Suggestion: Make a table (or a diagram) similar to that shown on page 6 – 9 to analyze the implications of the conflict resolution methods you select.

Using Column Groups

The procedures available in the DBMS_REPCAT package allow you to create and delete column groups, and to add members to, and remove members from, an existing column group.

Creating a Column Group with Members

To create a new column group with one or more members, call the MAKE_COLUMN_GROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.MAKE_COLUMN_GROUP(
    sname               => 'acct',
    oname               => 'inv',
    column_group        => 'address',
    list_of_column_names => 'addr1, addr2, city, state, zip');
```

This example creates a column group called ADDRESS that consists of the ADDR1, ADDR2, CITY, STATE, and ZIP columns in the INV table.

To create a column group consisting of all of the columns in the table, you simply pass an asterisk (*) as the final argument to the call. You must call this procedure from the master definition site. Your changes take effect when you generate replication support for the table.

Additional Information: The parameters for the MAKE_COLUMN_GROUP procedure are described in Table 12 – 149, and the exceptions are listed in Table 12 – 150.

Adding Members to an Existing Column Group

To add members to an existing column group, call the ADD_GROUPED_COLUMN procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.ADD_GROUPED_COLUMN(
    sname                => 'acct',
    oname                => 'inv',
    column_group         => 'address',
    list_of_column_names => 'phone, fax');
```

This example adds the columns PHONE and FAX to the ADDRESS column group created in a previous example. To add all of the columns in the table to the column group, you could specify '*' as the LIST_OF_COLUMN_NAMES value.

You must call this procedure from the master definition site. Your changes take effect when you generate replication support for the table.

> **Additional Information:** The parameters for the ADD_GROUPED_COLUMN procedure are described in Table 12 – 62, and the exceptions are listed in Table 12 – 63.

Removing Members from a Column Group

To remove members from a column group, call the DROP_GROUPED_COLUMN procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.DROP_GROUPED_COLUMN(
    sname                => 'acct',
    oname                => 'inv',
    column_group         => 'address',
    list_of_column_names => 'phone, fax');
```

This example removes the columns PHONE and FAX from the ADDRESS column group. To remove all of the columns in the table from the column group, you could specify '*' as the LIST_OF_COLUMN_NAMES value.

You must call this procedure from the master definition site. Your changes take effect when you generate replication support for the table.

> **Additional Information:** The parameters for the DROP_GROUPED_COLUMN procedure are described in Table 12 – 119, and the exceptions are listed in Table 12 – 120.

Dropping a Column Group

To drop a column group, call the DROP_COLUMN_GROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.DROP_COLUMN_GROUP(
    sname        => ’acct’,
    oname        => ’inv’,
    column_group => ’address’);
```

This example drops the ADDRESS column group associated with the INV table.

You must call this procedure from the master definition site. Your changes take effect when you generate replication support for the table.

> **Additional Information:** The parameters for the DROP_COLUMN_GROUP procedure are described in Table 12 – 117, and the exceptions are listed in Table 12 – 118.

Creating an Empty Column Group

To create a new, empty column group, call the DEFINE_COLUMN_GROUP procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.DEFINE_COLUMN_GROUP(
    sname => ’acct’,
    oname => ’inv’,
    gname => ’address’);
```

This example creates the ADDRESS column group associated with the INV table. This column group has no members. You must call the ADD_GROUPED_COLUMN procedure to add members to this group.

You must call this procedure from the master definition site. Your changes take effect when you generate replication support for the table.

> **Additional Information:** The parameters for the DEFINE_COLUMN_GROUP procedure are described in Table 12 – 109, and the exceptions are listed in Table 12 – 110.

Designating a Conflict Resolution Routine for a Table

There are separate procedures in the DBMS_REPCAT package for designating methods to resolve update, delete, and uniqueness conflicts. Use the ADD_UPDATE_RESOLUTION procedure to designate a method for resolving update conflicts for a given column group. Use the ADD_UNIQUE_RESOLUTION procedure to designate a method for resolving uniqueness conflicts involving a given unique constraint. Use the ADD_DELETE_RESOLUTION procedure to designate a method for resolving delete conflicts for a given table. (Recall that delete conflicts can result from an update to a primary key value.)

You must call these procedures from the master definition site. The conflict resolution method that you specify is not actually added until after the next time you generate replication support for the table.

You can designate multiple conflict resolution methods for a single column group, table, or constraint. If you provide multiple methods, they are applied in sequential order until the conflict is resolved or no more methods are available. You must provide a sequence order for each method that you add.

You can either designate one of the standard methods provided with the symmetric replication facility, or you can provide the name of a function that you have written yourself. Instructions for writing your own conflict resolution routine are provided on page 8 – 2. If you write your own conflict resolution routine, you must call CREATE_MASTER_REPOBJECT for this function to ensure that it exists at each master site.

For example, to indicate that you first want to use the TIMESTAMP method to resolve UPDATE conflicts with the ADDRESS column group, you would make a procedure call similar to the following:

```
DBMS_REPCAT.ADD_UPDATE_RESOLUTION(
    sname                 => 'acctg',
    oname                 => 'orders',
    column_group          => 'address',
    sequence_no           => 1,
    method                => 'TIMESTAMP',
    parameter_column_name => 'update_time');
```

Additional Information: The parameters for the ADD_UPDATE_RESOLUTION, ADD_DELETE_RESOLUTION, and ADD_UNIQUE_RESOLUTION procedures are described in Table 12 – 70, and the exceptions are listed in Table 12 – 71.

Changing a Conflict Resolution Routine

To change the conflict resolution method used for a table, you need to complete the following steps:

1. Suspend replication activity for the table's object group by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY.
2. Call the appropriate DBMS_REPCAT.ADD_*conflicttype*_RESOLUTION procedure with the new conflict resolution method.
3. Regenerate replication support for the object by calling either DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT or DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE.

   Because it is not necessary to recreate the generated replication triggers and their associated packages, you can save time by using the DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE procedure.
4. Resume replication activity by calling DBMS_REPCAT.RESUME_MASTER_ACTIVITY.

Generating Replication Procedures

To generate the supporting package for a replicated object at all master sites, as well as the audit tables and conflict resolution packages, call the DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE procedure, as shown in the following example:

```
DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE( sname => 'acct_rec',
                                          oname => 'inventory');
```

You must call this procedure from the master definition site for the given replicated object. Oracle must successfully create the necessary packages at the master definition site for this procedure to complete successfully. These objects are asynchronously created at the other master sites as described on page 4 – 35.

Additional Information: The parameters for the GENERATE_REPLICATION_PACKAGE procedure are described in Table 12 – 143, and the exceptions are listed in Table 12 – 144.

Dropping a Conflict Resolution Routine

There are separate procedures in the DBMS_REPCAT package for removing conflict resolution routines. Use the DROP_UPDATE_RESOLUTION procedure to drop a given routine for resolving update conflicts for a given column group. Use the DROP_DELETE_RESOLUTION procedure to drop a given routine for resolving delete conflicts for a given table. Use the DROP_UNIQUE_RESOLUTION procedure to drop a given routine for resolving uniqueness conflicts involving a given unique constraint.

These procedures must be called from the master definition site. The routine you designate is not actually dropped from usage until after the next time you generate replication support for the table.

The following example drops the TIMESTAMP resolution method for the ADDRESS column group:

```
DBMS_REPCAT.DROP_UPDATE_RESOLUTION(
    sname        => ’acctg’,
    oname        => ’orders’,
    column_group => ’address’,
    sequence_no  => 1);
```

Additional Information: The parameters for the DROP_UPDATE_RESOLUTION, DROP_DELETE_RESOLUTION, and DROP_UNIQUE_RESOLUTION procedures are described in Table 12 – 139, and the exceptions are listed in Table 12 – 140.

Using Priority Groups

To use the priority group method to resolve update conflicts, first create a priority group, then add this conflict resolution method for a column group. To create a priority group, do the following:

1. Define the name of the priority group and the datatype of the values in the group.
2. Define the priority level for each possible value of the “priority” column. This information is stored in the RepPriority view.

A single priority group can be used by multiple tables. Therefore, the name that you select for your priority group must be unique within a replicated object group. The column corresponding to this priority group can have different names in different tables.

You must indicate which column in a table is associated with a particular priority group when you add the priority group conflict resolution routine for the table. The priority group must therefore contain all possible values for all columns associated with that priority group.

For example, suppose that you had a table, INVENTORY, with a column of type VARCHAR2, STATUS, that could have three possible values: ORDERED, SHIPPED, and BILLED. Now suppose that you want to resolve update conflicts based upon the value of this STATUS column.

1. Create a column group that included the STATUS column.
2. Create and populate the status priority group.
3. Designate the PRIORITY GROUP conflict resolution method for the column group.

Your procedure calls would look similar to those shown below.

```
DBMS_REPCAT.MAKE_COLUMN_GROUP(
      sname                => 'acct',
      oname                => 'inventory',
      column_group         => 'all',
      list_of_column_names => '*');
DBMS_REPCAT.DEFINE_PRIORITY_GROUP(
      sname                => 'acct',
      pgroup               => 'status',
      datatype             => 'varchar2');
DBMS_REPCAT.ADD_PRIORITY_VARCHAR2(
      sname                => 'acct',
      pgroup               => 'status',
      value                => 'ordered',
      priority             => 1);
DBMS_REPCAT.ADD_PRIORITY_VARCHAR2(
      sname                => 'acct',
      pgroup               => 'status',
      value                => 'shipped',
      priority             => 2);
DBMS_REPCAT.ADD_PRIORITY_VARCHAR2(
      sname                => 'acct',
      pgroup               => 'status',
      value                => 'billed',
      priority             => 3);
DBMS_REPCAT.ADD_UPDATE_RESOLUTION(
      sname                => 'acct',
      oname                => 'inventory',
      column_group         => 'all',
      sequence_no          => 1,
      method               =>  'PRIORITY_GROUP',
      parameter_column_name => 'status');
```

The next several sections describe how to manage priority groups.

Creating a Priority Group

Use the DEFINE_PRIORITY_GROUP procedure in the DBMS_REPCAT package to create a new priority group for a replicated object group, as shown in the following example:

```
DBMS_REPCAT.DEFINE_PRIORITY_GROUP(
      gname     =>  'acct',
      pgroup    =>  'status',
      datatype  =>  'varchar2');
```

This example creates a priority group called STATUS for the ACCT object group. The members of this priority group will have values of type VARCHAR2.

You must call this procedure from the master definition site. The member is not added to the priority group until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

Additional Information: The parameters for the DEFINE_PRIORITY_GROUP procedure are described in Table 12 – 111, and the exceptions are listed in Table 12 – 112.

Adding Members to a Priority Group

There are several different procedures in the DBMS_REPCAT package for adding members to a priority group. These procedures are of the form ADD_PRIORITY_*type*, where *type* is equivalent to the datatype that you specified when you created the priority group:

- ADD_PRIORITY_CHAR
- ADD_PRIORITY_VARCHAR2
- ADD_PRIORITY_NUMBER
- ADD_PRIORITY_DATE
- ADD_PRIORITY_RAW

The procedure that you must call is determined by the datatype of your "priority" column. You must call this procedure once for each of the possible values of the "priority" column.

You must call this procedure from the master definition site. The value is synchronously available at the master definition site, but is not available at any other master sites until you run GENERATE_REPLICATION_SUPPORT. If you are modifying a priority group that is already in use, call the procedures in the following order to ensure proper resolution of conflicts:

1. DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY
2. DBMS_REPCAT.ADD_PRIORITY_*type*
3. DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT
4. DBMS_REPCAT.RESUME_MASTER_ACTIVITY

The following example adds the value SHIPPED to the STATUS priority group:

```
DBMS_REPCAT.ADD_PRIORITY_VARCHAR2(
      gname                =>  'acct',
      pgroup               =>  'status',
      value                =>  'shipped',
      priority             =>  2);
```

> **Additional Information:** The parameters for the ADD_PRIORITY_*type* procedures are described in Table 12 – 66, and the exceptions are listed in Table 12 – 67.

Altering the Value of a Member

There are several different procedures in the DBMS_REPCAT package for altering the value of a member of a priority group. These procedures are of the form ALTER_PRIORITY_*type*, where *type* is equivalent to the datatype that you specified when you created the priority group:

- ALTER_PRIORITY_CHAR
- ALTER_PRIORITY_VARCHAR2
- ALTER_PRIORITY_NUMBER
- ALTER_PRIORITY_DATE
- ALTER_PRIORITY_RAW

The procedure that you must call is determined by the datatype of your "priority" column. Because a priority group member consists of a priority associated with a particular value, these procedures enable you to change the value associated with a given priority level.

You must call this procedure from the master definition site. The value is synchronously available at the master definition site, but is not available at any other master sites until you run GENERATE_REPLICATION_SUPPORT. If you are modifying a priority group that is already in use, call the procedures in the following order to ensure proper resolution of conflicts:

1. DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY
2. DBMS_REPCAT.ALTER_PRIORITY_*type*
3. DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT
4. DBMS_REPCAT.RESUME_MASTER_ACTIVITY

You must call this procedure from the master definition site. The value of the member is not altered until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

The following example changes the recognized value of items at priority level 2 from SHIPPED to IN_SHIPPING:

```
DBMS_REPCAT.ALTER_PRIORITY_VARCHAR2(
      gname              =>  'acct',
      pgroup             =>  'status',
      old_value          =>  'shipped',
      new_value          =>  'in_shipping');
```

Additional Information: The parameters for the ALTER_PRIORITY_*type* procedures are described in Table 12 – 78, and the exceptions are listed in Table 12 – 79.

Altering the Priority of a Member

Use the ALTER_PRIORITY procedure in the DBMS_REPCAT package to alter the priority level associated with a given priority group member. Because a priority group member consists of a priority associated with a particular value, this procedure lets you raise or lower the priority of a given column value. Members with higher priority values are given higher priority when resolving conflicts.

You must call this procedure from the master definition site. The priority level of the member is not altered until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

The following example changes the priority of items marked as IN_SHIPPING from level 2 to level 4:

```
DBMS_REPCAT.ALTER_PRIORITY(
      gname              =>  'acct',
      pgroup             =>  'status',
      old_priority       =>  2,
      new_priority       =>  4);
```

Additional Information: The parameters for the ALTER_PRIORITY procedure are described in Table 12 – 76, and the exceptions are listed in Table 12 – 77.

Dropping a Member by Value

There are several different procedures in the DBMS_REPCAT package for dropping a member of a priority group by value. These procedures are of the form DROP_PRIORITY_*type*, where *type* is equivalent to the datatype that you specified when you created the priority group:

- DROP_PRIORITY_CHAR
- DROP_PRIORITY_VARCHAR2
- DROP_PRIORITY_NUMBER
- DROP_PRIORITY_DATE
- DROP_PRIORITY_RAW

The procedure that you must call is determined by the datatype of your "priority" column.

You must call this procedure from the master definition site. The member is not actually removed from the priority group until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

In the following example, IN_SHIPPING is no longer a valid state for items in the STATUS priority group:

```
DBMS_REPCAT.DROP_PRIORITY_VARCHAR2(
      gname               =>  'acct',
      pgroup              =>  'status',
      value               =>  'in_shipping');
```

Additional Information: The parameters for the DROP_PRIORITY_*type* procedures are described in Table 12 – 129, and the exceptions are listed in Table 12 – 130.

Dropping a Member by Priority

Use the DROP_PRIORITY procedure in the DBMS_REPCAT package to drop a member of a priority group by priority level.

You must call this procedure from the master definition site. The member is not actually removed from the priority group until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

In the following example, IN_SHIPPING (which was assigned to priority level 4) is no longer a valid state for items in the STATUS priority group:

```
DBMS_REPCAT.DROP_PRIORITY(
      gname               =>  'acct',
      pgroup              =>  'status',
      priority_num        =>  4);
```

Additional Information: The parameters for the DROP_PRIORITY procedure are described in Table 12 – 125, and the exceptions are listed in Table 12 – 126.

Dropping a Priority Group

Use the DROP_PRIORITY_GROUP procedure in the DBMS_REPCAT package to drop a priority group for a given replicated object group, as shown in the following example:

```
DBMS_REPCAT.DROP_PRIORITY_GROUP(
      gname               =>  'acct',
      pgroup              =>  'status');
```

In this example, STATUS is no longer a valid priority group.

☞ **Attention:** Before calling this procedure, you must call the DROP_UPDATE_RESOLUTION procedure for any column groups in the replicated object group that are using the PRIORITY GROUP conflict resolution method with this priority group. You can determine which column groups are affected by querying the RepResolution view.

You must call this procedure from the master definition site. The priority group is not actually dropped until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

Additional Information: The parameters for the DROP_PRIORITY_GROUP procedure are described in Table 12 – 127, and the exceptions are listed in Table 12 – 128.

Using Site Priority

Site priority is a specialized form of priority groups. Thus, many of the procedures associated with site priority behave similarly to the procedures associated with priority groups.

If you have chosen to use the site priority method to resolve update conflicts, you must first create a site priority group before you can add this conflict resolution method for a column group. Creation of a site priority group consists of two steps.

1. Define the name of the site priority group.
2. Add each site to the group and define its priority level. This information is stored in the RepPriority view.

In general, you will need only one site priority group for a replicated object group. This site priority group can be used by any number of replicated tables.

The next several sections describe how to manage site priority groups.

Creating a Site Priority Group

Use the DEFINE_SITE_PRIORITY procedure in the DBMS_REPCAT package to create a new site priority group for a replicated object group, as shown in the following example:

```
DBMS_REPCAT.DEFINE_SITE_PRIORITY(
      gname   =>   ’acct’,
      name    =>   ’site’);
```

This example creates a site priority group called SITE for the ACCT object group.

You must call this procedure from the master definition site. The necessary support for site priority is not generated until you call GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

> **Additional Information:** The parameters for the DEFINE_SITE_PRIORITY procedure are described in Table 12 – 113, and the exceptions are listed in Table 12 – 114.

Adding a Site to the Group

Use the ADD_SITE_PRIORITY_SITE procedure in the DBMS_REPCAT package to add a new site to a site priority group, as shown in the following example:

```
DBMS_REPCAT.ADD_SITE_PRIORITY_SITE(
      gname      =>   ’acct’,
      name       =>   ’site’,
      site       =>   ’hq.widgetek.com’,
      priority   =>   100);
```

This example adds the HQ site to the SITE group and sets its priority level to 100.

> **Note:** The highest priority is given to the site with the highest priority value. Priority values do not have to be consecutive integers.

You must call this procedure from the master definition site. The site is not added to the group until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

> **Additional Information:** The parameters for the ADD_SITE_PRIORITY_SITE procedure are described in Table 12 – 68, and the exceptions are listed in Table 12 – 69.

Altering the Priority Level of a Site

Use the ALTER_SITE_PRIORITY procedure in the DBMS_REPCAT package to alter the priority level associated with a given site, as shown in the following example:

```
DBMS_REPCAT.ALTER_SITE_PRIORITY(
      gname          =>   'acct',
      name           =>   'site',
      old_priority   =>   100,
      new_priority   =>   200);
```

This example changes the priority level of a site in the SITE group from 100 to 200.

> **Note:** The highest priority is given to the site with the highest priority value. Priority values do not have to be consecutive integers.

You must call this procedure from the master definition site. The priority level is not actually updated until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

> **Additional Information:** The parameters for the ALTER_SITE_PRIORITY procedure are described in Table 12 – 80, and the exceptions are listed in Table 12 – 81.

Altering the Site Associated with a Priority Level

Use the ALTER_SITE_PRIORITY_SITE procedure in the DBMS_REPCAT package to alter the site associated with a given priority level, as shown in the following example:

```
DBMS_REPCAT.ALTER_SITE_PRIORITY_SITE(
      gname      =>   'acct',
      name       =>   'site',
      old_site   =>   'hq.widgetek.com',
      new_site   =>   'hq.widgetworld.com);
```

This example changes the global database name of the HQ site to HQ.WIDGETWORLD.COM, while its priority level remains the same.

You must call this procedure from the master definition site. The site name is not actually updated until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

> **Additional Information:** The parameters for the ALTER_SITE_PRIORITY_SITE procedure are described in Table 12 – 82, and the exceptions are listed in Table 12 – 83.

Dropping a Site by Site Name

Use the DROP_SITE_PRIORITY_SITE procedure in the DBMS_REPCAT package to drop a given site, by name, from a site priority group, as shown in the following example:

```
DBMS_REPCAT.DROP_SITE_PRIORITY_SITE(
      gname   =>   'acct',
      name    =>   'site',
      site    =>   'hq.widgetek.com');
```

This example drops the HQ site from the SITE group.

You must call this procedure from the master definition site. The site is not actually removed from the group until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

> **Additional Information:** The parameters for the DROP_SITE_PRIORITY_SITE procedure are described in Table 12 – 133, and the exceptions are listed in Table 12 – 134.

Dropping a Site by Priority Level

Use the DBMS_REPCAT.DROP_PRIORITY procedure described on page 6 – 32 to drop a site from a site priority group by priority level.

Dropping a Site Priority Group

Use the DROP_SITE_PRIORITY procedure in the DBMS_REPCAT package to drop a site priority group for a given replicated object group, as shown in the following example:

```
DBMS_REPCAT.DROP_SITE_PRIORITY(
      gname   =>   'acct',
      name    =>   'site');
```

In this example, SITE is no longer a valid site priority group.

> ☞ **Attention:** Before calling this procedure, you must call the DROP_UPDATE_RESOLUTION procedure for any column groups in the replicated object group that are using the SITE PRIORITY conflict resolution method with this site priority group. You can determine which column groups are affected by querying the RepResolution view.

You must call this procedure from the master definition site. The site priority group is not actually dropped until you call the procedure GENERATE_REPLICATION_SUPPORT for any table in the object group (since there is no group–level equivalent for this command).

Additional Information: The parameters for the DROP_SITE_PRIORITY procedure are described in Table 12 – 131, and the exceptions are listed in Table 12 – 132.

Viewing Conflict Resolution Information

The symmetric replication facility provides several views that you can use to determine what conflict resolution methods are being used by each of the tables and column groups in your replicated environment.

Each view has three versions: USER_*, ALL_*, SYS.DBA_*. The views available include the following:

RepResolution_Method	Lists all of the available conflict resolution methods.
RepColumn_Group	Lists all of the column groups defined for the database.
RepGrouped_Column	Lists all of the columns in each column group in the database.
RepPriority_Group	Lists all of the priority groups and site priority groups defined for the database.
RepPriority	Lists the values and corresponding priority levels for each priority or site priority group.
RepConflict	Lists the types of conflicts (delete, update, or uniqueness) for which you have specified a resolution method, for the tables, column groups, and unique constraints in the database.
RepResolution	Shows more specific information about the conflict resolution method used to resolve conflicts on each object.
RepParameter_Column	Shows which columns are used by the conflict resolution routines to resolve a conflict.

Examples

This section provides examples that you can modify and / or expand with your own conflict resolution routines.

Suppose that you allowed multiple sites to update your CUSTOMER table, and want to set up conflict resolution as follows:

- the ADDITIVE method to resolve update conflicts on the AMOUNT column, which indicates the total merchandise ordered by the customer
- the LATEST TIMESTAMP method on the remaining columns
- SITE PRIORITY as a backup method, because two people could theoretically update the same field at precisely the same time

This example

- uses conflict notification combined with the APPEND SEQUENCE system–defined routine to address uniqueness conflicts
- does not resolve delete conflicts
- assumes that your CUSTOMER table is as follows:

```
     CREATE TABLE customers
     (
     custno     NUMBER(4) PRIMARY KEY,
     last_name  VARCHAR2(10),
     first_name VARCHAR2(10),
     addr1      VARCHAR2(30),
     addr2      VARCHAR2(30),
      amount     NUMBER(7,2),
     timestamp  DATE,
      site       VARCHAR2(128),
     CONSTRAINT c_cust_name UNIQUE (last_name, first_name)
)
```

Conflict Notification Log

This example

- shows how user–defined conflict resolution routines can be used for conflict notification.
- uses a database table to record conflict notifications. This table will be registered as a replicated object so that it will be defined automatically at the other masters. However, this table will NOT have replication support generated for it because its entries are specific to the site that detects a conflict.

```
CREATE TABLE conf_report
(
  line          NUMBER(2),     --- used to order message text
  txt           VARCHAR2(80),  --- conflict notification message
  timestamp     DATE,          --- time of conflict
  table_name    VARCHAR2(30),  --- table in which the conflict occurred
  table_owner   VARCHAR2(30),  --- owner of the table
  conflict_type VARCHAR2(6)    --- INSERT, DELETE or UNIQUE
)
```

Sample Conflict Notification Package and Package Body

The following package and package body perform a simple form of *conflict notification* by logging UNIQUENESS conflicts for the CUSTOMERS table into the CONF_REPORT table. With simple modifications, the user–defined conflict resolution routine can take more active steps. For example, instead of just recording the notification message, the package DBMS_OFFICE can be used to send an Oracle Office e–mail message to the DBA.

Note: This example of *conflict notification* does not resolve any conflicts. You should either provide a method to resolve conflicts (for example, *discard* or *overwrite*), or provide a notification mechanism that will succeed (for example, using e–mail) even if the error is not resolved and the transaction is rolled back.

```
CREATE OR REPLACE PACKAGE notify AS

  --- Report uniqueness constraint violations on customer table
  FUNCTION customer_unique_violation(first_name         IN OUT VARCHAR2,
                                     last_name          IN OUT VARCHAR2,
                                     discard_new_values IN OUT BOOLEAN)
    RETURN BOOLEAN;
END notify;
/
CREATE OR REPLACE PACKAGE BODY notify AS

  --- Define a PL/SQL table to hold the notification message
  TYPE message_table IS TABLE OF VARCHAR2(80) INDEX BY BINARY_INTEGER;

  PROCEDURE report_conflict(conflict_report IN MESSAGE_TABLE,
                            report_length   IN NUMBER,
                            conflict_time   IN DATE,
                            conflict_table  IN VARCHAR2,
                            table_owner     IN VARCHAR2,
                            conflict_type   IN VARCHAR2) IS
  BEGIN
    FOR idx IN 1..report_length LOOP
      BEGIN
        INSERT INTO off_shore_accounts.conf_report
        (line, txt, timestamp, table_name, table_owner, conflict_type)
         VALUES (idx, SUBSTR(conflict_report(idx),1,80), conflict_time,
                  conflict_table, table_owner, conflict_type);
      EXCEPTION WHEN others THEN NULL;
      END;
    END LOOP;
  END report_conflict;

  --- This is the conflict resolution routine that will be called first
  --- when a uniqueness constraint violated is detected in the customer
  --- table.
  FUNCTION customer_unique_violation(first_name         IN OUT VARCHAR2,
                                     last_name          IN OUT VARCHAR2,
                                     discard_new_values IN OUT BOOLEAN)
     RETURN BOOLEAN IS
    local_node  VARCHAR2(128);
    conf_report MESSAGE_TABLE;
    conf_time   DATE := SYSDATE;
  BEGIN
    --- Get the global name of the local site
    BEGIN
      SELECT global_name INTO local_node FROM global_name;
    EXCEPTION WHEN others THEN local_node := '?';
    END;
    --- Generate a message for the DBA
    conf_report(1) := 'UNIQUENESS CONFLICT DETECTED IN TABLE CUSTOMER ON ' ||
                      TO_CHAR(conf_time, 'MM-DD-YYYY HH24:MI:SS');
    conf_report(2) := '  AT NODE ' || local_node;
    conf_report(3) := 'ATTEMPTING TO RESOLVE CONFLICT USING' ||
                      ' APPEND SEQUENCE METHOD';
    conf_report(4) := 'FIRST NAME: ' || first_name;
    conf_report(5) := 'LAST NAME:  ' || last_name;
    conf_report(6) := NULL;
    --- Report the conflict
    report_conflict(conf_report, 5, conf_time, 'CUSTOMER',
                    'OFF_SHORE_ACCOUNTS', 'UNIQUE');

    --- Do not discard the new column values. They are still needed by
    --- other conflict resolution routines
    discard_new_values := FALSE;
    --- Indicate that the conflict was not resolved.
    RETURN FALSE;
  END customer_unique_violation;
END notify;
/
```

Creating the Object Group and Replicated Objects

After defining the tables and the NOTIFY package and package body, you can now *create the object group and replicated objects.* The CUSTOMERS table, CONF_REPORT tables, and the NOTIFY package and package body are defined as replicated objects in the same object group. You can also use the Symmetric Replication facility to distribute these objects to all master sites, but you must generate replication support for the CUSTOMER table.

```
––– Create the replicated object group
dbms_repcat.create_master_repgroup(gname          => ’ORDER_ENTRY’,
                                   group_comment  => ’Foreign Orders’,
                                   master_comment => ’Main Office: NY’);

––– Register customer as a replicated object.
dbms_repcat.create_master_repobject(gname => ’ORDER_ENTRY’,
                                    sname => ’OFF_SHORE_ACCOUNTS’,
                                    oname => ’CUSTOMERS’,
                                    type  => ’TABLE’);

––– Replication only the DDL for the notification table. Do NOT generate
––– replication support on this table
dbms_repcat.create_master_repobject(gname => ’ORDER_ENTRY’,
                                    sname => ’OFF_SHORE_ACCOUNTS’,
                                    oname => ’CONF_REPORT’,
                                    type  => ’TABLE’);

––– Register Notify package and body as a replicated object.
dbms_repcat.create_master_repobject(gname => ’ORDER_ENTRY’,
                                    sname => ’OFF_SHORE_ACCOUNTS’,
                                    oname => ’NOTIFY’,
                                    type  => ’PACKAGE’);
dbms_repcat.create_master_repobject(gname => ’ORDER_ENTRY’,
                                    sname => ’OFF_SHORE_ACCOUNTS’,
                                    oname => ’NOTIFY’,
                                    type  => ’PACKAGE BODY’);
```

Declaring Conflict Resolution Methods for Update Conflicts

After declaring the replicated objects, you can then begin *declaring the conflict resolution methods.* Here, we define two object groups: one for the AMOUNT column and one for the other non–primary key columns. The primary key in this example would be generated by a sequence that has its numbers partitioned among the sites so that this column should avoid being involved in conflicts.

The column group for the AMOUNT column is assigned the ADDITIVE method. The other column group is assigned the TIMESTAMP method with SITE PRIORITY as a back–up method. This example defines the priority of two sites.

```
––– Indicate the columns to resolve if a conflict is detected
––– Primary key is not in a user–defined column group

dbms_repcat.make_column_group(
                     sname                => 'OFF_SHORE_ACCOUNTS',
                     oname                => 'CUSTOMERS',
                     column_group         => 'CG_CUSTOMERS',
                     list_of_column_names => 'LAST_NAME,FIRST_NAME,' ||
                                             'ADDR1,ADDR2,' ||
                                             'TIMESTAMP,SITE');
dbms_repcat.make_column_group(
                      sname                => 'OFF_SHORE_ACCOUNTS',
                      oname                => 'CUSTOMERS',
                      column_group         => 'CG_CUST_AMT',
                      list_of_column_names => 'AMOUNT');

––– make priority group for site priority

dbms_repcat.define_site_priority(
                       gname           => 'ORDER_ENTRY',
                       name            => 'SITE PRIORITY',
                       comment         => 'site priority for customers');

––– add values to site priority group

dbms_repcat.add_site_priority_site(
               gname    => 'ORDER_ENTRY',
               name     => 'SITE PRIORITY',
               site     => 'DBS1.REGRESS.RDBMS.DEV.US.ORACLE.COM',
               priority => 200);

dbms_repcat.add_site_priority_site(
               gname    => 'ORDER_ENTRY',
               name     => 'SITE PRIORITY',
               site     => 'DBS2.REGRESS.RDBMS.DEV.US.ORACLE.COM',
               priority => 100);
```

```
––– define update resolution for Amount column
dbms_repcat.add_update_resolution(
                 sname                 => ’OFF_SHORE_ACCOUNTS’,
                 oname                 => ’CUSTOMERS’,
                 column_group          => ’CG_CUST_AMT’,
                 sequence_no           => 1,
                 method                => ’ADDITIVE’,
                 parameter_column_name => ’AMOUNT’);

––– define timestamp with site–priority backup for other column group
dbms_repcat.add_update_resolution(
                 sname                 => ’OFF_SHORE_ACCOUNTS’,
                 oname                 => ’CUSTOMERS’,
                 column_group          => ’CG_CUSTOMERS’,
                 sequence_no           => 1,
                 method                => ’LATEST TIMESTAMP’,
                 parameter_column_name => ’TIMESTAMP’);

dbms_repcat.add_update_resolution(
                 sname                 => ’OFF_SHORE_ACCOUNTS’,
                 oname                 => ’CUSTOMERS’,
                 column_group          => ’CG_CUSTOMERS’,
                 sequence_no           => 2,
                 method                => ’SITE PRIORITY’,
                 parameter_column_name => ’SITE’,
                 priority_group        => ’SITE PRIORITY’);
```

Declaring Conflict Resolution Methods for Unique Conflicts

You can also *declare methods for handling uniqueness conflicts* for the C_CUST_NAME constraint:

1. Register a user–defined procedure that will perform the conflict notification.
2. Register the APPEND SEQUENCE system–defined method. Because the APPEND SEQUENCE method never converges, you will always want notification to occur. Therefore, the SEQUENCE_NO for the notification method must be smaller than sequence_no for APPEND SEQUENCE method. The notification method only wants the columns associated for the first and last names.

```
  --- register a user-defined resolution routine for notification

dbms_repcat.add_unique_resolution(
  sname                 => 'OFF_SHORE_ACCOUNTS',
  oname                 => 'CUSTOMERS',
  constraint_name       => 'C_CUST_NAME',
  sequence_no           => 1,
  method                => 'USER FUNCTION',
  comment               => 'Notify DBA',
  parameter_column_name => 'FIRST_NAME,LAST_NAME',
  function_name         =>'OFF_SHORE_ACCOUNTS.NOTIFY.CUSTOMER_UNIQUE_VIOLATION');

      --- register a system-defined resolution routine for non-convergent
      --- resolution of the uniqueness conflict.

      dbms_repcat.add_unique_resolution(
                  sname                 => 'OFF_SHORE_ACCOUNTS',
                  oname                 => 'CUSTOMERS',
                  constraint_name       => 'C_CUST_NAME',
                  sequence_no           => 2,
                  method                => 'APPEND SEQUENCE',
                  comment               => 'Resolve Conflict',
                  parameter_column_name => 'LAST_NAME');
```

Sample Timestamp and Site Maintenance Trigger

In either a trigger or in your application, you must implement the logic necessary to maintain the timestamp and site information. The following example trigger considers clock synchronization problems, but needs to be modified if the application crosses time zones. This trigger is specific to the CG_CUSTOMERS column group.

Because the trigger uses one of the generated procedures to check whether or not the trigger should actually be fired, it is necessary to generate replication support for the CUSTOMERS table before creating the trigger. This will also allow transactions on the customer table to be propagated.

```
dbms_repcat.generate_replication_support(sname => 'OFF_SHORE_ACCOUNTS',
                                         oname => 'CUSTOMERS',
                                         type  => 'TABLE');
```

Now you can define the trigger:

```
create or replace trigger "OFF_SHORE_ACCOUNTS"."T_CUSTOMERS"
  before insert or update on "OFF_SHORE_ACCOUNTS"."CUSTOMERS"
  for each row
declare
  TIMESTAMP$X DATE := SYSDATE;
  SITE$X VARCHAR2(128) := dbms_reputil.global_name;
begin
  -- Don't fire if a snapshot refreshing;
  –– Don’t fire if a master and replication is turned off
  if (”OFF_SHORE_ACCOUNTS”.”CUSTOMERS$TP”.active) then
    if not dbms_reputil.from_remote then
      if inserting then
        –– set site and timestamp columns.
        :new.”TIMESTAMP” := TIMESTAMP$X;
        :new.”SITE” := SITE$X;
      elsif updating then
        if (:old.”ADDR1” = :new.”ADDR1” or
           (:old.”ADDR1” is null and :new.”ADDR1” is null)) and
           (:old.”ADDR2” = :new.”ADDR2” or
           (:old.”ADDR2” is null and :new.”ADDR2” is null)) and
           (:old.”FIRST_NAME” = :new.”FIRST_NAME” or
           (:old.”FIRST_NAME” is null and :new.”FIRST_NAME” is null)) and
           (:old.”LAST_NAME” = :new.”LAST_NAME” or
           (:old.”LAST_NAME” is null and :new.”LAST_NAME” is null)) and
           (:old.”SITE” = :new.”SITE” or
           (:old.”SITE” is null and :new.”SITE” is null)) and
           (:old.”TIMESTAMP” = :new.”TIMESTAMP” or
           (:old.”TIMESTAMP” is null and :new.”TIMESTAMP” is null)) then
          –– column group was not changed; do nothing
          NULL;
        else
          –– column group was changed; set site and timestamp columns.
          :new.”SITE” := SITE$X;
          :new.”TIMESTAMP” := TIMESTAMP$X;
          –– consider time synchronization problems;
          –– previous update to this row may have originated from a site
          –– with a clock time ahead of the local clock time.
          if :old.”TIMESTAMP” is not null and
            :old.”TIMESTAMP” > :new.”TIMESTAMP” then
            :new.”TIMESTAMP” := :old.”TIMESTAMP” + 1 / 86400;
          elsif :old.”TIMESTAMP” is not null and
            :old.”TIMESTAMP” = :new.”TIMESTAMP” and
            (:old.”SITE” is null OR :old.”SITE” != :new.”SITE”) then
            :new.”TIMESTAMP” := :old.”TIMESTAMP” + 1 / 86400;
          end if;
        end if;
      end if;
    end if;
  end if;
end;
```

You can propagate the trigger to other masters by registering it with the symmetric replication facility:

```
 dbms_repcat.create_master_repobject(gname => ’ORDER_ENTRY’,
                                      sname => ’OFF_SHORE_ACCOUNTS’,
                                      oname => ’T_CUSTOMERS’,
                                      type  => ’TRIGGER’);
```

After confirming that all replication objects are correctly generated and replicated, start up replication activity:

```
dbms_repcat.resume_master_activity(gname => ’ORDER_ENTRY’);
```

,

CHAPTER

7

Administering a Replicated Environment

This chapter describes how to administer your replicated database environment. The topics include the following:

- determining the cause of an error
- resolving an error
- recovery
- auditing the successful resolution of conflicts
- applying outstanding log information at a site
- importing and exporting replicated data
- determining differences between replicated tables
- updating the comment fields in the views associated with replication

Note that most of the activities described in this chapter can be accomplished much more easily by using Oracle's Replication Manager, a GUI interface for replication. See the documentation for Oracle Replication Manager for more information.

Determining the Cause of an Error

When Oracle pushes a deferred transaction from a snapshot or master site to another master site, it uses two–phase commit to ensure that the transaction is not removed from the local queue until it has been successfully propagated to the remote site. A transaction can be successfully propagated without being successfully applied. An error in applying a deferred transaction may be the result of a database problem, such as a lack of available space in a table that you are attempting to update, or may be the result of an unresolvable update conflict. If an error occurs, Oracle performs the following actions at the destination site:

- rolls back the transaction
- logs the error in the DefError view
- adds the transaction to the local deferred transaction views (since it is no longer queued for this location at the originating site)

Deferred transactions consist of a series of deferred remote procedure calls that must be applied in a given order to maintain transaction consistency. The DefError view provides the ID of the transaction that could not be applied. You can use this ID to locate the queued calls associated with this transaction. These calls are stored in the DefCall view. You can use the procedures in the DBMS_DEFER_QUERY package to determine the arguments to the procedures listed in the DefCall view.

After you successfully resolve the error at the destination site, it is removed from the DefError view, as well as the DefCall and DefTran views at the destination site. Because the transaction queued for this site was removed from the originating queue when the transaction was originally pushed, resolving the transaction at the destination site has no effect on the queue at the originating site.

Determining the Type of an Argument

The DefCall view provides you with the name of each remote procedure call associated with a particular deferred transaction, and the number of arguments to the procedure. Before you can get the value of an argument to a deferred remote procedure call, you must know the type of the argument. Use the GET_ARG_TYPE function in the DBMS_DEFER_QUERY package to determine the type of an argument, as shown in the following example:

```
type_no := DBMS_DEFER_QUERY.GET_ARG_TYPE(
               call_no            =>   234,
               deferred_tran_db   =>   'acct_hq.hq.com',
               arg_no             =>   3,
               deferred_tran_id   =>   '1.7.356');
```

In this example, the GET_ARG_TYPE function returns the datatype of the third argument to the deferred remote procedure call with ID number 234 that originated from the ACCT_HQ database.

> **Additional Information:** The parameters for the GET_ARG_TYPE function are described in Table 12 – 9, the exceptions are listed in Table 12 – 10, and the possible return values are described in Table 12 – 11.

Determining the Value of an Argument

Once you know the type of an argument to a deferred remote procedure call, you can next determine the value of this argument by using the appropriate function in the DBMS_DEFER_QUERY package. For example, if you wanted to get the value of an argument of type VARCHAR2, you would use the GET_VARCHAR2_ARG function, as shown in the following example:

```
val := DBMS_DEFER_QUERY.GET_VARCHAR2_ARG(
           callno             =>   234,
           deferred_tran_db   =>   'acct_hq.hq.com',
           arg_no             =>   3)
```

Whereas the call to GET_ARG_TYPE in the previous example returned only the datatype of the third argument to the procedure with ID number 234, the call to the GET_VARCHAR2_ARG function returns the actual value passed for this argument.

The type of the argument value that you want to retrieve determines the name of the function that you need to call. The supported datatypes are: NUMBER, VARCHAR2, CHAR, DATE, RAW, and ROWID. The datatype of the return value must match the datatype of the function name.

> **Additional Information:** The parameters for all of the GET_*datatype*_ARG functions are described in Table 12 – 12, and the exceptions are listed in Table 12 – 13.

Manually Resolving an Error

Once you have determined the cause of an error, you may need to perform one or more of the following actions at the destination site after fixing the error:

- re–execute the transaction that originally resulted in the update conflict
- remove the entry from the DefError view

There may also be times when you need to delete a transaction from the deferred transaction queue.

Re–executing a Transaction

To re–execute a deferred transaction that did not initially complete successfully, call the EXECUTE_ERROR procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.EXECUTE_ERROR(
      deferred_tran_id  =>  ’234’,
      deferred_tran_db  =>  ’acct_hq.hq.com’,
      destination       =>  ’acct_ny.ny.com’);
```

This example re–executes the transaction with ID number 234 that originated at the ACCT_HQ site and was queued for execution at the ACCT_NY site.

Upon successful execution, the transaction is removed from the DefError view, as well as the local deferred transaction views. Although when you call EXECUTE_ERROR you must always specify the database for which the transaction was originally queued (that is, the site where you want to re–execute a transaction), you can choose to re–execute a single transaction, all transactions originating from a given location, or all transactions, regardless of their originating location.

If you call EXECUTE_ERROR for a single transaction, that transaction is not committed, even if it completes successfully. If you are satisfied with the results of the transaction, you should issue the SQL command COMMIT WORK. If EXECUTE_ERROR re–executes multiple transactions, each transaction is committed as it completes.

Additional Information: The parameters for the EXECUTE_ERROR procedure are described in Table 12 – 24.

Deleting a Transaction from the DefError View

To delete a transaction from the DefError view, call the DELETE_ERROR procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.DELETE_ERROR(
      deferred_tran_id  =>  '234',
      deferred_tran_db  =>  'acct_hq.hq.com',
      destination       =>  'acct_ny.ny.com');
```

This example removes the transaction with ID number 234 that originated at the ACCT_HQ site and was queued for execution at the ACCT_NY site from the DefError view.

Calling DELETE_ERROR removes the specified transaction from the DefError view, as well as the local deferred transaction views. By passing null for selected arguments, you can remove all transactions associated with a particular site from the DefError view. For example, assuming that you have sites A, B, and C, you can choose to remove from the DefError view

- a transaction originating at site A from the queue for site B
- all transactions originating at site A from the queue for site B
- all transactions originating at site A from the queues for all other sites (that is, sites B and C)
- all transactions originating at any site (that is, sites A and C) from the queue for site B

Additional Information: The parameters for the DELETE_ERROR procedure are described in Table 12 – 18.

Deleting a Transaction from the Deferred Transaction Queue

To delete a transaction from the deferred transaction queue, call the DELETE_TRAN procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.DELETE_TRAN(
      deferred_tran_id   =>  '234',
      deferred_tran_db   =>  'acct_hq.hq.com',
      destination        =>  'acct_ny.ny.com');
```

This example deletes the transaction with ID number 234 that originated at the ACCT_HQ site and was queued for execution at the ACCT_NY site.

Calling DELETE_TRAN removes the transaction from the queue for the destination database. If you do not specify a destination database, the transaction is removed from the queues for all destinations. For example, assuming that you have sites A, B, and C you can choose to delete:

- a transaction originating at site A from the queue for site B
- all transactions originating at site A from the queue for site B
- all transactions originating at site A from the queues for all other sites (that is, sites B and C)
- all transactions originating at any site (that is, sites A and C) from the queue for site B

After Oracle deletes a transaction, if the transaction is not queued for any other destinations, Oracle removes the appropriate entries from the DefTran and DefCall views as well.

Additional Information: The parameters for the DELETE_TRAN procedure are described in Table 12 – 19.

Recovery

Databases using symmetric replication are distributed databases. Follow the guidelines for distributed database backups outlined in the *Oracle7 Server Administrator's Guide* when creating backups of symmetric replication databases. Follow the guidelines for coordinated distributed recovery in the *Oracle7 Server Administrator's Guide* when recovering a symmetric replication database.

If you fail to follow the coordinated distributed recovery guidelines, there is no guarantee that your symmetric replication databases will be consistent. For example, a restored master site may have propagated different transactions to different masters. You may need to perform extra steps to correct for an incorrect recovery operation. One such method is as to drop and recreate all replicated objects in the recovered database.

You should consider removing pending deferred transactions and deferred error records from the restored database, and resolve any outstanding distributed transactions before dropping and recreating replicated objects. If the restored database was a master definition site for some replicated environments, you should designate a new master definition site (as described below) before dropping and creating objects. Any snapshots mastered at the restored database should be fully refreshed, as well as any snapshots in the restored database.

To provide continued access to your data, you may need to change master definition sites (assuming the database being recovered was the master definition site), or remaster snapshot sites (assuming their master site is being recovered). These techniques are described in the following sections.

Changing Master Definition Sites

To change your master definition site to another master site in your replicated environment, call the RELOCATE_MASTERDEF procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.RELOCATE_MASTERDEF(
      gname                 =>   'acct',
      old_masterdef         =>   'acct_hq.hq.com',
      new_masterdef         =>   'acct_ny.ny.com',
      notify_masters        =>   TRUE,
      include_old_masterdef =>   TRUE);
```

In this example, the master definition site for the ACCT replicated object group is changed from ACCT_HQ to ACCT_NY. The former master definition site remains as a master site in the replicated environment. By default, all master sites, including the former master definition site, are notified of this change.

Additional Information: The parameters for the RELOCATE_MASTERDEF procedure are described in Table 12 – 159, and the exceptions are listed in Table 12 – 160.

Usage Notes

It is not necessary for either the old or new master definition site to be available when you call RELOCATE_MASTERDEF. In a planned reconfiguration, you should invoke RELOCATE_MASTERDEF with NOTIFY_MASTERS TRUE and INCLUDE_OLD_MASTERDEF TRUE. If just the master definition site fails, you should invoke RELOCATE_MASTERDEF with NOTIFY_MASTERS TRUE and INCLUDE_OLD_MASTERDEF FALSE. If several master sites and the master definition site fail, the replication administrator should invoke RELOCATE_MASTERDEF at each operational master with NOTIFY_MASTERS FALSE.

Changing a Snapshot Site's Master

To change the master database of a snapshot replicated object group to another master site, call the SWITCH_SNAPSHOT_MASTER procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.SWITCH_SNAPSHOT_MASTER(
      gname    =>        'acct',
      master   =>        'acct_ny.ny.com'
      execute_as_user => 'FALSE');
```

In this example, the master site for the ACCT object group is changed to the ACCT_NY database.

You must call this procedure at the snapshot site whose master site you want to change. The new database must be a master site in the replicated environment.

When you call this procedure, Oracle uses the new master to perform a full refresh of each snapshot in the local object group.

The entries in the SYS.SLOG$ table at the old master site for the switched snapshot are not removed. As a result, the MLOG$ table of the switched updatable snapshot at the old master site has the potential to grow indefinitely, unless you purge it by calling DBMS_SNAPSHOT.PURGE_LOG.

Additional Information: The parameters for the SWITCH_SNAPSHOT_MASTER procedure are described in Table 12 – 171, and the exceptions are listed in Table 12 – 172.

Auditing the Successful Resolution of Conflicts

Whenever the symmetric replication facility detects and successfully resolves an update, delete, or uniqueness conflict, you can view information about what method was used to resolve the conflict by querying the RepResolution_Statistics view. This view is only updated if you have chosen to turn on conflict resolution statistics gathering for the table involved in the conflict.

Gathering Conflict Resolution Statistics

Use the REGISTER_STATISTICS procedure in the DBMS_REPCAT package to collect information about the successful resolution of update, delete, and uniqueness conflicts for a table. The following example gathers statistics for the EMP table in the ACCT_REC schema:

```
DBMS_REPCAT.REGISTER_STATISTICS(  sname   =>      'acct_rec',
                                  oname   =>      'emp');
```

Additional Information: The parameters for the REGISTER_STATISTICS procedure are described in Table 12 – 157, and the exceptions are listed in Table 12 – 158

Viewing Conflict Resolution Statistics

After you call REGISTER_STATISTICS for a table, each conflict that is successfully resolved for that table is logged in the RepResolution_Statistics view. Information about unresolved conflicts is always logged to the DefError view, whether the object is registered or not.

Additional Information: The RepResolution_Statistics view is described in Table 13 – 18 and the DefError view is described in Table 13 – 23.

Canceling Conflict Resolution Statistics Gathering

Use the CANCEL_STATISTICS procedure in the DBMS_REPCAT package if you no longer want to collect information about the successful resolution of update, delete, and uniqueness conflicts for a table. The following example cancels statistics gathering on the EMP table in the ACCT_REC schema:

```
DBMS_REPCAT.CANCEL_STATISTICS(    sname   =>     'acct_rec',
                                  oname   =>     'emp');
```

Additional Information: The parameters for the CANCEL_STATISTICS procedure are described in Table 12 – 86, and the exceptions are listed in Table 12 – 87.

Removing Statistics Information

If you registered a table to log information about the successful resolution of update, delete, and uniqueness conflicts, you can remove this information from the RepResolution_Statistics view by calling the PURGE_STATISTICS procedure in the DBMS_REPCAT package.

The following example purges the statistics gathered about conflicts resolved due to inserts, updates, and deletes on the EMP table between January 1 and March 31:

```
DBMS_REPCAT.PURGE_STATISTICS(    sname         =>    'acct_rec',
                                 oname         =>    'emp',
                                 start_date    =>    '01-JAN-95',
                                 end_date      =>    '31-MAR-95);
```

Additional Information: The parameters for the PURGE_STATISTICS procedure are described in Table 12 – 153, and the exceptions are listed in Table 12 – 154.

Determining Outstanding Changes

After making changes to your replicated environment at the master definition site, you can use the WAIT_MASTER_LOG procedure in the DBMS_REPCAT package to determine if these changes have been applied to your current master site.

In the following example, this procedure waits until either 120 seconds have passed or there are at most 5 records in the local RepCatLog view that represent administrative activities for the ACCT replicated object group that have not completed before returning the actual number of incomplete activities.

Activities that have completed with or without an error are not considered. This allows you to determine if changes that were asynchronously propagated to a master site have been applied.

```
incomplete NATURAL;
DBMS_REPCAT.WAIT_MASTER_LOG(  gname          =>     'acct',
                              record_count   =>     5,
                              timeout        =>     120,
                              true_count     =>     incomplete);
```

If there are N master sites and one master definition site for a replicated object group, most asynchronous administrative requests eventually create N+1 log records at the master definition site and one log record at each master. ADD_MASTER_DATABASE is an exception, and may create a log record at the master definition site and a log record at the new master site for each object in the replicated object group.

Additional Information: The parameters for the WAIT_MASTER_LOG procedure are described in Table 12 – 173, and the exceptions are listed in Table 12 – 174.

Applying Outstanding Changes

Whenever you alter a replicated object group, the DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN procedure must run at each master site in order for the changes applied at the master definition site to be visible everywhere. Usually this procedure is periodically invoked at each master site by a background process and no manual intervention is required.

Whenever you add a new master site to your replicated environment, a job is automatically inserted into the job queue. This job periodically executes the procedure DO_DEFERRED_REPCAT_ADMIN. Whenever you alter a replicated object group, Oracle attempts to start this job immediately in order to apply the replicated changes at each master site.

In the following example, the local outstanding deferred administrative procedures for the ACCT replicated object group are executed (with the assistance of job queues) for all master sites. Had ALL_SITES been set to the default value, FALSE, the deferred administrative procedures would have been executed at the current master site only.

```
DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN(   gname      =>   'acct',
                                        all_sites  =>   true);
```

Note: DO_DEFERRED_REPCAT_ADMIN executes only those administrative requests submitted by the connected user that called DO_DEFERRED_REPCAT_ADMIN. Requests submitted by other users are ignored.

Assuming that Oracle does not encounter any errors, DO_DEFERRED_REPCAT_ADMIN will be run whenever a background process is available to execute the job. The initialization parameter JOB_QUEUE_INTERVAL determines how often the background process wakes up.

☞ **Attention:** If the deferred changes involve generating replication support as described on page 4 – 21, it will be necessary to invoke DO_DEFERRED_REPCAT_ADMIN twice to insure that both phases 1 and 2 are executed due to object dependencies.

You can experience a delay if you do not have enough background processes available to execute the outstanding jobs. For more information on scheduling jobs, see Chapter 10.

If the initialization parameter JOB_QUEUE_PROCESSES is set to zero at a master site, you must manually connect to that site and invoke DO_DEFERRED_REPCAT_ADMIN to execute asynchronous requests at that site. Because this procedure may use dynamic SQL to perform DDL, you must never invoke it as a remote procedure call.

Additional Information: The parameters for the DO_DEFERRED_REPCAT_ADMIN procedure are described in Table 12 – 115, and the exceptions are listed in Table 12 – 116.

Removing Log Information

If you resolved an error condition from a replication administration request, you can remove related error messages in the RepCatLog view by calling the PURGE_MASTER_LOG procedure in the DBMS_REPCAT package, as shown in the following example:

```
DBMS_REPCAT.PURGE_MASTER_LOG(     id      =>      1763,
                                  source  =>      'acct_ny.ny.com',
                                  gname   =>      'acct');
```

This procedure removes all local log records corresponding to the request on the ACCT replicated object group that originated at the ACCT_NY master with the identification number 1763. If any parameter is NULL, Oracle treats it as a wildcard.

Like most other procedures in the DBMS_REPCAT package that are executed at a master site, a side effect of PURGE_MASTER_LOG is to perform any local deferred administrative requests for the given replicated object group. This administration is attempted before the log is purged.

Additional Information: The parameters for the PURGE_MASTER_LOG procedure are described in Table 12 – 151, and the exceptions are listed in Table 12 – 152.

Importing and Exporting Replicated Data

The following sections describe how to use the Import, Export, and SQL*Loader utilities with replicated data.

Loading and Import of Replicated Data

When importing or loading data into replicated tables you need to consider whether you want the data to be replicated to other sites by symmetric replication. You might not wish to use symmetric replication to replicate the data if you are loading or importing a large amount of data and are therefore performing the load or import at each site.

If you wish to replicate loaded or imported data using symmetric replication you must ensure that the replication triggers fire as data is loaded or imported. The direct path of SQL*Loader cannot be used. You must be certain that the replication triggers are created and enabled before importing and loading data.

If you do not wish to replicate loaded or imported data using symmetric replication you can use the direct path of SQL*Loader, or you can explicitly disable replication while loading or importing data. To disable the replication of the data, use either of the following methods:

- Call DBMS_REPUTIL.REPLICATION_OFF to disable replication for your session.

  If you use this method, be sure to set DBMS_REPUTIL.REPLICATION_ON before performing other database operations.

- Use the DISABLE clause of the ALTER TRIGGER command to disable the replication triggers.

  If you disable the replication triggers, be sure that other updates to the replicated tables do not occur while data is being loaded or imported because those updates will not be replicated while the triggers are disabled.

If you use this method, be sure to use the ENABLE clause of the ALTER TRIGGER command to re–enable the triggers before performing other database operations.

Using Export/Import to Reorganize Deferred RPC Queues and Replication Catalogs

You may occasionally need to reorganize deferred remote procedure call queues to reclaim fragmented space (you are less likely to need to reorganize replication catalogs, but you can use similar procedures).

⚠ **Warning:** Do not (accidentally) copy the contents of the deferred remote procedure call queues or the replication catalogs between databases. This includes using Export/Import to transfer data, or copying files to create a new database. The only exception is when you replace one database with another database that has the same global name. Error message 23327 is issued during import if you attempt to import the data from a deferred remote procedure call queue into a database with a different global name.

To reorganize deferred remote procedure call queues or replication catalogs using Export/Import, complete the following steps:

1. Place the database in RESTRICTED SESSION mode using the SQL command ALTER SYSTEM with the ENABLE RESTRICTED SESSION option. After placing an instance in restricted mode, you might also want to kill all current user sessions. This prevents concurrent activities from adding rows to the deferred remote procedure call queues.
2. Export the deferred remote procedure call queue tables with indexes and constraints:

   system.def$_destination

   system.def$_call

   system.def$_calldest

   system.def$_error

   system.def$_defaultdest
3. Drop the deferred remote procedure call queue tables.
4. Import the deferred remote procedure call queue tables that you exported in step 2.
5. Lift the instance from restricted mode by using the SQL command ALTER SYSTEM with the DISABLE RESTRICTED SESSION option.

Performing Checks on Imported Data

After performing an export/import of a replicated object or an object used by the symmetric replication facility (for example, the RepSchema view), you should run the REPCAT_IMPORT_CHECK procedure in the DBMS_REPCAT package.

In the following example, the procedure checks the objects in the ACCT replicated object group at a snapshot site to ensure that they have the appropriate object identifiers and status values:

```
DBMS_REPCAT.REPCAT_IMPORT_CHECK(  gname   =>      'acct',
                                  master  =>      FALSE);
```

Additional Information: The parameters for the REPCAT_IMPORT_CHECK procedure are described in Table 12 – 163, and the exceptions are listed in Table 12 – 164.

Supplying Asynchronous DDL

If you need to perform a task for which Oracle does not provide a method, you can supply the DDL that you want to have executed at each master site by calling the EXECUTE_DDL procedure in the DBMS_REPCAT package.

The following example creates an index on the NAME column of the EMP table at the ACCT_NY master site. Had the default, NULL, been used for the MASTER_LIST argument, the index would have been created at all master sites.

```
DBMS_REPCAT.EXECUTE_DDL(
      gname        =>     'acct',
      master_list  =>     'acct_ny.ny.com',
      ddl_text     =>     'CREATE INDEX name_idx ON
                           acct_rec.emp(name)');
```

You can call this procedure only from the master definition site. The DDL is applied asynchronously at each of the designated sites as described on page 4 – 35. The RepCatLog view contains interim status and any asynchronous error messages generated by the request. Although the replicated object group need not be quiesced when you invoke EXECUTE_DDL, you may prefer to quiesce the environment first by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY.

Additional Information: The parameters for the EXECUTE_DDL procedure are described in Table 12 – 141, and the exceptions are listed in Table 12 – 142.

Determining Differences Between Replicated Tables

When administering a replicated environment, you may periodically want to check whether the contents of two replicated tables are identical. The following procedures in the DBMS_RECTIFIER_DIFF package let you identify, and optionally rectify, the differences between two tables when both sites are release 7.3 or higher:

`DIFFERENCES`	The DIFFERENCES procedure compares two replicas of a table, and determines all rows in the first replica that are not in the second and all rows in the second that are not in the first. The output of this procedure is stored in two user–created tables. The first table stores the values of the missing rows, and the second table is used to indicate which site contains each row.
`RECTIFY`	The RECTIFY procedure uses the information generated by the DIFFERENCES procedure to rectify the two tables. Any rows found in the first table and not in the second are inserted into the second table. Any rows found in the second table and not in the first are deleted from the second table.

To restore equivalency between all copies of a replicated table, you should complete the following steps:

1. Select one copy of the table to be the “reference” table. This copy will be used to update all other replicas of the table as needed.
2. Determine if it is necessary to check all rows and columns in the table for differences, or only a subset. For example, it may not be necessary to check rows that have not been updated since the last time that you checked for differences. Although it is not necessary to check all columns, your column list must include all columns that make up the primary key (or that you designated as a substitute primary key by calling DBMS_REPCAT.SET_COLUMNS) for the table.
3. After determining which columns you will be checking in the table, you need to create two tables to hold the results of the comparison.

   You must create one table that can hold the data for the columns being compared. For example, if you decide to compare the EMPNO, SAL, and BONUS columns of the EMPLOYEE table, your CREATE statement would need to be similar to the one shown below.

```
CREATE TABLE missing_rows_data
(
   empno   NUMBER,
   sal     NUMBER,
   bonus   NUMBER
)
```

You must also create a table that indicates where the row is found. This table must contain three columns with the data types shown in the following example:

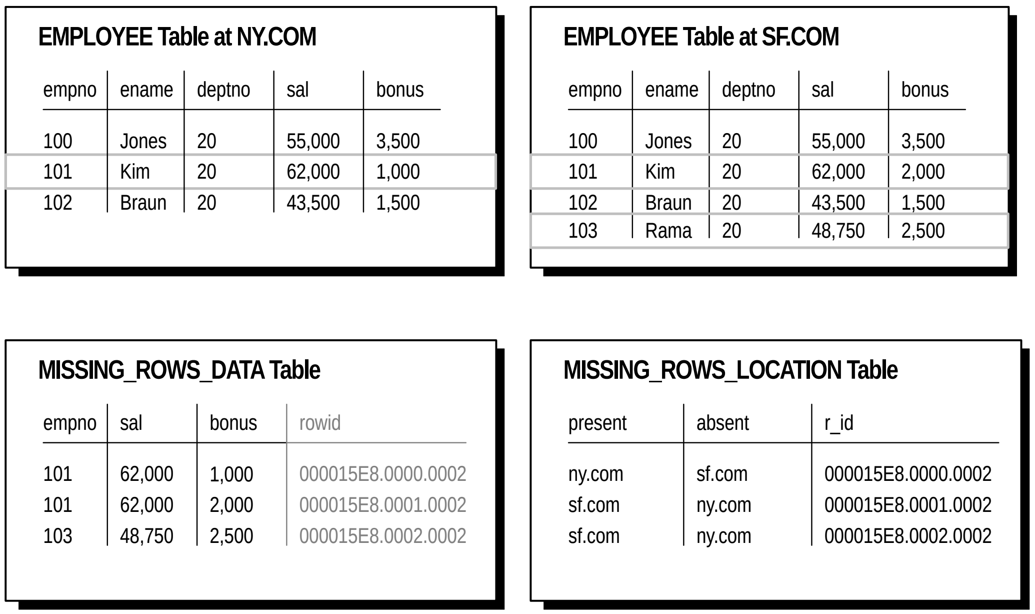
```
CREATE TABLE missing_rows_location
(
   present   VARCHAR2(128),
   absent    VARCHAR2(128),
   r_id      ROWID
)
```

4. Quiesce the object group containing the tables that you want to compare by calling DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY. Although quiescing the replicated object group is not a requirement, rectifying tables that were not quiesced first can result in inconsistencies in your data.

5. At the site containing the “reference” table, call the DBMS_RECTIFIER_DIFF.DIFFERENCES procedure. For example, if you wanted to compare the EMPLOYEE tables at the New York and San Francisco sites, your procedure call would look similar to the following:

```
DBMS_RECTIFIER_DIFF.DIFFERENCES(
    sname1               =>  ’hr’,
    oname1               =>  ’employee’,
    reference_site       =>  ’ny.com’,
    sname2               =>  ’hr’,
    oname2               =>  ’employee’,
    comparison_site      =>  ’sf.com’,
    where_clause         =>  ’’,
    column_list          =>  ’empno,sal,bonus’,
    missing_rows_sname   =>  ’scott’,
    missing_rows_oname1  =>  ’missing_rows_data’,
    missing_rows_oname2  =>  ’missing_rows_location’,
    missing_rows_site    =>  ’ny.com’,
    max_missing          =>  100,
    commit_rows          =>  50);
```

Figure 7 – 1 shows an example of two replicas of the EMPLOYEE table and what the resulting missing rows tables would look like if you executed the DIFFERENCES procedure on these replicas.



EMPLOYEE Table at NY.COM

empno	ename	deptno	sal	bonus
100	Jones	20	55,000	3,500
101	Kim	20	62,000	1,000
102	Braun	20	43,500	1,500

EMPLOYEE Table at SF.COM

empno	ename	deptno	sal	bonus
100	Jones	20	55,000	3,500
101	Kim	20	62,000	2,000
102	Braun	20	43,500	1,500
103	Rama	20	48,750	2,500

MISSING_ROWS_DATA Table

empno	sal	bonus	rowid
101	62,000	1,000	000015E8.0000.0002
101	62,000	2,000	000015E8.0001.0002
103	48,750	2,500	000015E8.0002.0002

MISSING_ROWS_LOCATION Table

present	absent	r_id
ny.com	sf.com	000015E8.0000.0002
sf.com	ny.com	000015E8.0001.0002
sf.com	ny.com	000015E8.0002.0002

Figure 7 – 1 Determining Differences Between Replicas

Notice that the two missing rows tables are related by the ROWID and r_id columns.

6. Now you can rectify the table at the “comparison” site to be equivalent to the table at the “reference” site by calling the DBMS_RECTIFIER_DIFF.RECTIFY procedure as shown in the following example:

```
DBMS_RECTIFIER_DIFF.RECTIFY(
    sname1              =>  'hr',
    oname1              =>  'employee',
    reference_site      =>  'ny.com',
    sname2              =>  'hr',
    oname2              =>  'employee',
    comparison_site     =>  'sf.com',
    column_list         =>  'empno,sal,bonus',
    missing_rows_sname  =>  'scott',
    missing_rows_oname1 =>  'missing_rows_data',
    missing_rows_oname2 =>  'missing_rows_location',
    missing_rows_site   =>  'ny.com',
    commit_rows         =>  50);
```

The RECTIFY procedure temporarily disables replication at the "comparison" site while it performs the necessary insertions and deletions, as you would not want to propagate these changes. RECTIFY first performs all of the necessary DELETEs and then performs all of the INSERTs. This ensures that there are no violations of a PRIMARY KEY constraint.

☞ **Attention:** If you have any additional constraints on the "comparison" table you must ensure that they will not be violated when you call RECTIFY. You may need to update the table directly using the information from the missing rows table. If so, be certain to DELETE the appropriate rows from the missing rows tables.

7. After you have successfully executed the RECTIFY procedure, your missing rows tables should be empty. You can now repeat steps 5 and 6 for the remaining copies of the replicated table. Remember to use the same "reference" table each time to ensure that all copies are identical when you complete this procedure.

8. You may now resume replication activity by calling DBMS_REPCAT.RESUME_MASTER_ACTIVITY.

Updating Comments

There are several procedures in the DBMS_REPCAT package that allow you to update the comment information in the various views associated with replication. Table 7 – 1 lists the appropriate procedure to call for each view.

View	DBMS_REPCAT Procedure	Additional Information

RepGroup	`COMMENT_ON_REPGROUP(`
	`  gname     IN  VARCHAR2,`
	`  comment   IN  VARCHAR2)`
		The parameters for the COMMENT_ON_REPGROUP procedure are described in Table 12 – 92, and the exceptions are listed in Table 12 – 93.

RepObject	`COMMENT_ON_REPOBJECT(`
	`  sname     IN  VARCHAR2,`
	`  oname     IN  VARCHAR2,`
	`  type      IN  VARCHAR2,`
	`  comment   IN  VARCHAR2)`
		The parameters for the COMMENT_ON_REPOBJECT procedure are described in Table 12 – 96, and the exceptions are listed in Table 12 – 97.

RepSites	`COMMENT_ON_REPSITES(`
	`  gname     IN  VARCHAR2,`
	`  master    IN  VARCHAR,`
	`  comment   IN  VARCHAR2)`
		The parameters for the COMMENT_ON_REPSITES procedure are described in Table 12 – 94, and the exceptions are listed in Table 12 – 95.

Table 7 – 1 Updating Comments in Symmetric Replication Facility Views, continued on next page

View	DBMS_REPCAT Procedure	Additional Information
RepColumn_Group	`COMMENT_ON_COLUMN_GROUP( sname IN VARCHAR2, oname IN VARCHAR2, column_group IN VARCHAR2, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_COLUMN_GROUP procedure are described in Table 12 – 88, and the exceptions are listed in Table 12 – 89.
RepPriority_Group	`COMMENT_ON_PRIORITY_GROUP( gname IN VARCHAR2, pgroup IN VARCHAR2, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_PRIORITY_GROUP procedure are described in Table 12 – 90, and the exceptions are listed in Table 12 – 91.
RepPriority_Group (site priority group)	`COMMENT_ON_SITE_PRIORITY( gname IN VARCHAR2, name IN VARCHAR2, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_SITE_PRIORITY procedure are described in Table 12 – 90, and the exceptions are listed in Table 12 – 91.
RepResolution (uniqueness conflicts)	`COMMENT_ON_UNIQUE_RESOLUTION( sname IN VARCHAR2, oname IN VARCHAR2, constraint_nameIN VARCHAR2, sequence_no IN NUMBER, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_UNIQUE_RESOLUTION procedures are described in Table 12 – 98, and the exceptions are listed in Table 12 – 99.
RepResolution (update conflicts)	`COMMENT_ON_UPDATE_RESOLUTION( sname IN VARCHAR2, oname IN VARCHAR2, column_group IN VARCHAR2, sequence_no IN NUMBER, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_UNIQUE_RESOLUTION procedures are described in Table 12 – 98, and the exceptions are listed in Table 12 – 99.
RepResolution (delete conflicts)	`COMMENT_ON_DELETE_RESOLUTION( sname IN VARCHAR2, oname IN VARCHAR2, sequence_no IN NUMBER, comment IN VARCHAR2)`	The parameters for the COMMENT_ON_UNIQUE_RESOLUTION procedures are described in Table 12 – 98, and the exceptions are listed in Table 12 – 99.

Table 7 – 1 Updating Comments in Symmetric Replication Facility Views

CHAPTER

8

Advanced Techniques

This chapter describes advanced techniques that you might want to use in implementing your replicated environment. This chapter describes the following topics:

- how to write your own conflict resolution routine
- how to design a survivable system
- how to employ advanced conflict avoidance techniques
- how to use procedural replication
- enabling and disabling replication in procedures and triggers
- avoiding delete conflicts
- using connection qualifiers

Writing a Conflict Resolution Routine

A conflict resolution routine is a PL/SQL function that returns either TRUE or FALSE. TRUE indicates that the routine has successfully resolved all conflicting modifications for a column group. If the conflict cannot be successfully resolved, the routine should return FALSE. Oracle continues to evaluate available conflict resolution routines, in sequence order, until either a routine returns TRUE or there are no more routines available.

If the conflict resolution routine raises an exception, Oracle stops evaluation of the routine, and, if any other routines were provided to resolve the conflict (with a later sequence number), Oracle does not evaluate them.

Conflict Resolution Routine Parameters

The parameters needed by a conflict resolution routine are determined by the type of conflict being resolved (unique, update, or delete) and the columns of the table being replicated. All conflict resolution routines take some combination of old, new, and current column values for the table.

The old value represents the value of the row at the initiating site before you made the change. The new value represents the value of the row at the initiating site after you made the change. The current value represents the value of the equivalent row at the receiving site. Recall that Oracle uses the primary key (or the key specified by SET_COLUMNS) to determine which rows to compare.

The conflict resolution function should accept as parameters the values for the columns specified in the PARAMETER_COLUMN_NAME argument to the DBMS_REPCAT.ADD_*conflicttype*_CONFLICT procedures. The column parameters are passed to the conflict resolution routine in the order listed in the PARAMETER_COLUMN_NAME argument, or in ascending alphabetical order if you specified '*' for this argument. Where both old and new column values are passed as parameters (for update conflicts), the old value of the column immediately precedes the new value.

☞ **Attention:** Type checking of parameter columns in user–defined conflict resolution routines is not performed until the DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT procedure generates and compiles the packages.

Resolving Update Conflicts

For update conflicts, the function should accept the following values for each column in the column group:

- Old column value from the initiating site. The mode for this parameter is IN. This value should not be changed.
- New column value from the initiating site. The mode for this parameter is IN OUT. If the function can resolve the conflict successfully, it should modify the new column value as needed.
- Current column value from the receiving site. The mode for this parameter is IN.

The old, new, and current values for a column are received consecutively. The final argument to the conflict resolution routine should be a boolean flag. If this flag is FALSE, it indicates that you have updated the value of the IN OUT parameter, new, and that you should update the current column value with this new value. If this flag is TRUE, it indicates that the current column value should not be changed.

Resolving Uniqueness Conflicts

Uniqueness conflicts can occur as the result of an INSERT or UPDATE. Your uniqueness conflict resolution routine should accept the new column value from the initiating site in IN OUT mode for each column in the column group. The final parameter to the conflict resolution routine should be a BOOLEAN flag.

If the routine can resolve the conflict, it should modify the new column values so that the symmetric replication facility can insert or update the current row with the new column values. Your function should set the BOOLEAN flag to TRUE if it wants to discard the new column values, and FALSE otherwise.

Because your conflict resolution routine cannot guarantee convergence for uniqueness conflicts, your routine should include a notification mechanism.

Resolving Delete Conflicts

Delete conflicts occur when you successfully delete from the local site, but the associated row cannot be found at the remote site (for example, because it had been updated). For delete conflicts, the function should accept old column values in IN OUT mode for the entire row. The final parameter to the conflict resolution routine should be a BOOLEAN flag.

If the conflict resolution routine can resolve the conflict, it modifies the old column values so that the symmetric replication facility can delete the current row that matches all old column values. Your function should set the BOOLEAN flag to TRUE if it wants to discard these column values, and FALSE otherwise.

If you perform a delete at the local site and an update at the remote site, the remote site detects the delete conflict, but the local site detects an unresolvable update conflict. This type of conflict cannot be handled automatically. The conflict will raise a NO_DATA_FOUND exception and the transaction will be placed in the error table.

Designing a mechanism to properly handle these types of update/delete conflicts is difficult. It is far easier to avoid these types of conflicts entirely, by simply "marking" deleted rows, and then purging them using procedural replication, as described on page 8 – 20.

Restrictions

You should avoid the following commands in your conflict resolution routines. Use of these commands can result in unpredictable results.

- Data Definition Language
- Transaction Control
- Session Control
- System Control

Example Conflict Resolution Routine

The following examples show variations on the standard MAXIMUM and ADDITIVE conflict resolution methods. Unlike the standard methods, these user functions are designed to handle nulls in the columns used to resolve the conflict.

Maximum User Function

```
-- User function similar to MAXIMUM method.
-- If curr is null or curr < new, use new values.
-- If new is null or new < curr, use current values.
-- If both are null, no resolution.
-- Does not converge with > 2 masters, unless
-- always increasing.

FUNCTION max_null_loses(old                 IN     NUMBER,
                        new                 IN OUT NUMBER,
                        cur                 IN     NUMBER,
                        ignore_discard_flag OUT    BOOLEAN)
  RETURN BOOLEAN IS
BEGIN
    IF (new IS NULL AND cur IS NULL) OR new = cur THEN
        RETURN FALSE;
    END IF;
    IF new IS NULL THEN
      ignore_discard_flag := TRUE;
    ELSIF cur IS NULL THEN
      ignore_discard_flag := FALSE;
    ELSIF new < cur THEN
```

```
      ignore_discard_flag := TRUE;
    ELSE
        ignore_discard_flag := FALSE;
    END IF;
    RETURN TRUE;
END max_null_loses;
```

Additive User Function

```
-- User function similar to ADDITIVE method.
-- If old is null, old = 0.
-- If new is null, new = 0.
-- If curr is null, curr = 0.
-- new = curr + (new - old) -> just like ADDITIVE method.

FUNCTION additive_nulls(old                 IN     NUMBER,
                        new                 IN OUT NUMBER,
                        cur                 IN     NUMBER,
                        ignore_discard_flag OUT    BOOLEAN)
  RETURN BOOLEAN IS
  old_val NUMBER := 0.0;
  new_val NUMBER := 0.0;
  cur_val NUMBER := 0.0;
BEGIN
    IF old IS NOT NULL THEN
      old_val := old;
    END IF;
    IF new IS NOT NULL THEN
      new_val := new;
    END IF;
    IF cur IS NOT NULL THEN
      cur_val := cur;
    END IF;
    new := cur_val + (new_val - old_val);
    ignore_discard_flag := FALSE;
    RETURN TRUE;
END additive_nulls;
```

Survivability

Survivability provides the capability to continue running applications despite system or site failures. It allows applications to be run on a fail–over system, accessing the same, or very nearly the same, data as they were on the primary system when it failed. As shown in Figure 8 – 1, the Oracle Server provides two different technologies for accomplishing survivability: the Oracle Parallel Server and the symmetric replication facility.

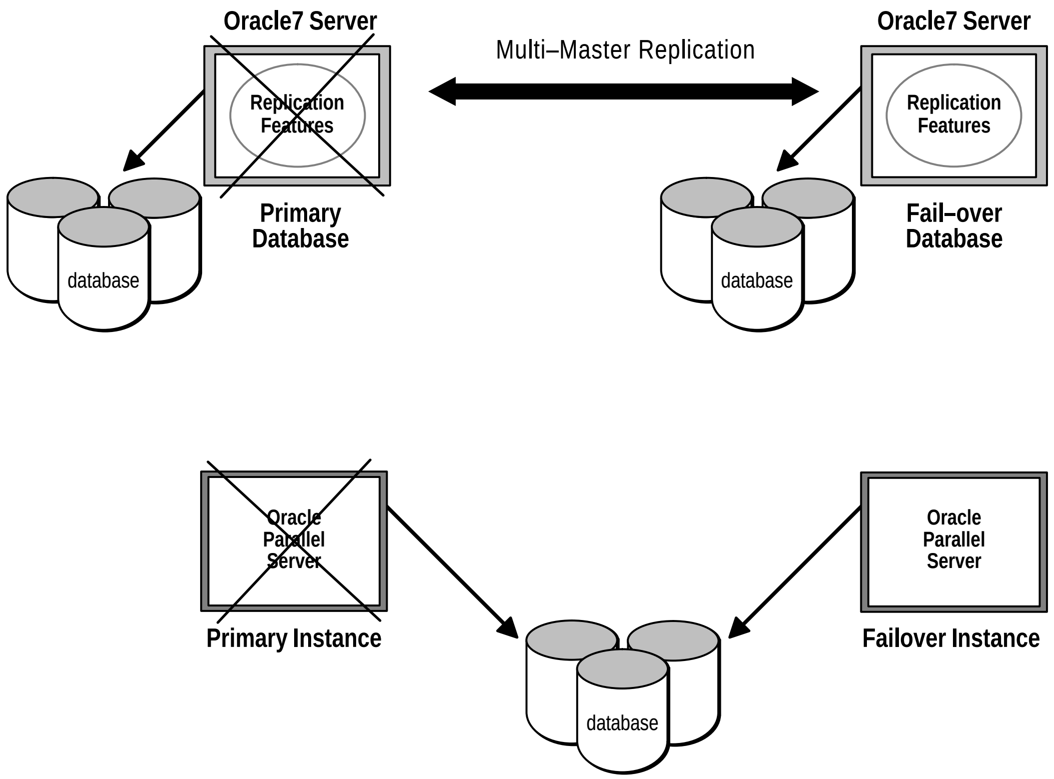


Figure 8 – 1 Survivability Methods: Symmetric Replication vs. Parallel Server

Oracle Parallel Server versus Symmetric Replication

The Oracle Parallel Server supports fail–over to surviving systems when a system supporting an instance of the Oracle Server fails. The Oracle Parallel Server requires a cluster or massively parallel hardware platform, and thus is applicable for protection against processor system failures in the local environment where the cluster or massively parallel system is running.

In these environments, the Oracle Parallel Server is the ideal solution for survivability — supporting high transaction volumes with no lost transactions or data inconsistencies in the event of an instance failure. If an instance fails, a surviving instance of the Oracle Parallel Server automatically recovers any incomplete transactions. Applications running on the failed system can execute on the fail–over system, accessing all of the data in the database.

The Oracle Parallel Server does not, however, provide survivability for site failures (such as flood, fire, or sabotage) that render an entire site, and thus the entire cluster or massively parallel system, inoperable. To provide survivability for site failures, you can use the symmetric replication facility to maintain a replicate of a database at a geographically remote location.

Should the local system fail, the application can continue to execute at the remote site. Symmetric replication, however, cannot guarantee that no transactions will be lost. Also, special care must be taken to prevent data inconsistencies when the primary site is recovered.

Designing for Survivability

If you choose to use the symmetric replication facility for survivability, you should consider the following issues:

- The symmetric replication facility must be able to keep up with the transaction volume of the primary system. This is application specific, but generally much lower than the throughput supported if you are using the Oracle Parallel Server.
- If a failure occurs at the primary site, recently committed transactions at the primary site may not have been asynchronously propagated to the fail–over site yet. These transactions will appear to be lost.
- These "lost" transactions must be dealt with when the primary site is recovered.

  Suppose, for example, you are running an order–entry system that uses replication to maintain a remote fail–over order–entry system, and the primary system fails.

  At the time of the failure, there were two transactions recently executed at the primary site that did not have their changes propagated and applied at the fail–over site. The first of these was a transaction that entered a new order, and the second was a transaction that cancelled an existing order.

In the first case, someone may notice the absence of the new order when processing continues on the fail–over system, and re–enter it. In the second case, the cancellation of the order may not be noticed, and processing of the order may proceed; that is, the canceled item may be shipped and the customer billed.

What happens now, when you restore the primary site? If you simply push all of the changes executed on the fail–over system back to the primary system, you will encounter conflicts.

Specifically, there will be duplicate orders for the item originally ordered at the primary system just before it failed. Additionally, there will be data changes resulting from the transactions to ship and bill the order that was originally canceled on the primary system.

You must carefully design your system, as described in the next section, to deal with these situations.

Implementing a Survivable System

Oracle's symmetric replication facility can be used to provide survivability against site failures by using multiple replicated master sites. You must configure your system using one of the following methods. These methods are listed in order of increasing implementation difficulty.

- The fail–over site is used for read access only. That is, no updates are allowed at the fail–over site, even when the primary site fails.
- After a failure, the primary site is restored from the fail–over site using export / import, or via full backup.
- Full conflict resolution is employed for all data / transactions. This requires careful design and implementation. You must ensure proper resolution of conflicts that can occur when the primary site is restored, such as duplicate transactions.
- Provide your own special applications–level routines and / or procedures to deal with the inconsistencies that occur when the primary site is restored, and the queued transactions from the active fail–over system are propagated and applied to the primary site.

Dynamic Ownership Conflict Avoidance

This section describes a more advanced method of designing your applications to avoid conflicts. This method, known as token passing, is similar to the workflow method described in Chapter 1. Although this section describes how to use this method to control the ownership of an entire row, you can use a modified form of this method to control ownership of the individual column groups within a row.

Both workflow and token passing allow dynamic ownership of data. With dynamic ownership, only one site at a time is allowed to update a row, but ownership of the row can be passed from site to site. Both work flow and token passing use the value of one or more “identifier” columns to determine who is currently allowed to update the row.

Workflow

With workflow partitioning, you can think of data ownership as being ”pushed” from site to site. Only the current owner of the row is allowed to push the ownership of the row to another site, by changing the value of the “identifier” columns.

Take the simple example of separate sites for ordering, shipping, and billing. Here, the identifier columns are used to indicate the status of an order. The status determines which site can update the row. After a user at the ordering site has entered the order, he or she updates the status of this row to SHIP. Users at the ordering site are no longer allowed to modify this row — ownership has been pushed to the shipping site.

After shipping the order, the user at the shipping site will update the status of this row to BILL, thus pushing ownership to the billing site, and so on.

To successfully avoid conflicts, applications implementing dynamic data ownership must ensure that the following conditions are met:

- Only the owner of the row can update the row.
- The row is never owned by more than one site.
- Ordering conflicts can be successfully resolved at all sites.

With workflow partitioning, only the current owner of the row can push the ownership of the row to the next site by updating the “identifier” columns. No site is given ownership unless another site has given up ownership; thus ensuring there is never more than one owner.

Because the flow of work is ordered, ordering conflicts can be resolved by applying the change from the site that occurs latest in the flow of work. Any ordering conflicts can be resolved using a form of the Priority conflict resolution method, where the priority value increases with each step in the work flow process.

The PRIORITY conflict resolution method successfully converges for more than one master as long as the priority value is always increasing.

Token Passing

Token passing uses a more generalized approach to meeting these criteria. To implement token passing, instead of the "identifier" columns, your replicated tables must have owner and epoch columns. The owner column stores the global database name of the site currently believed to own the row.

This column should be used exclusively for establishing ownership and should not otherwise be updated. The epoch column is used to resolve ordering conflicts. This number is updated each time the ownership of the row changes. Thus the change associated with the highest epoch number is the most recent change.

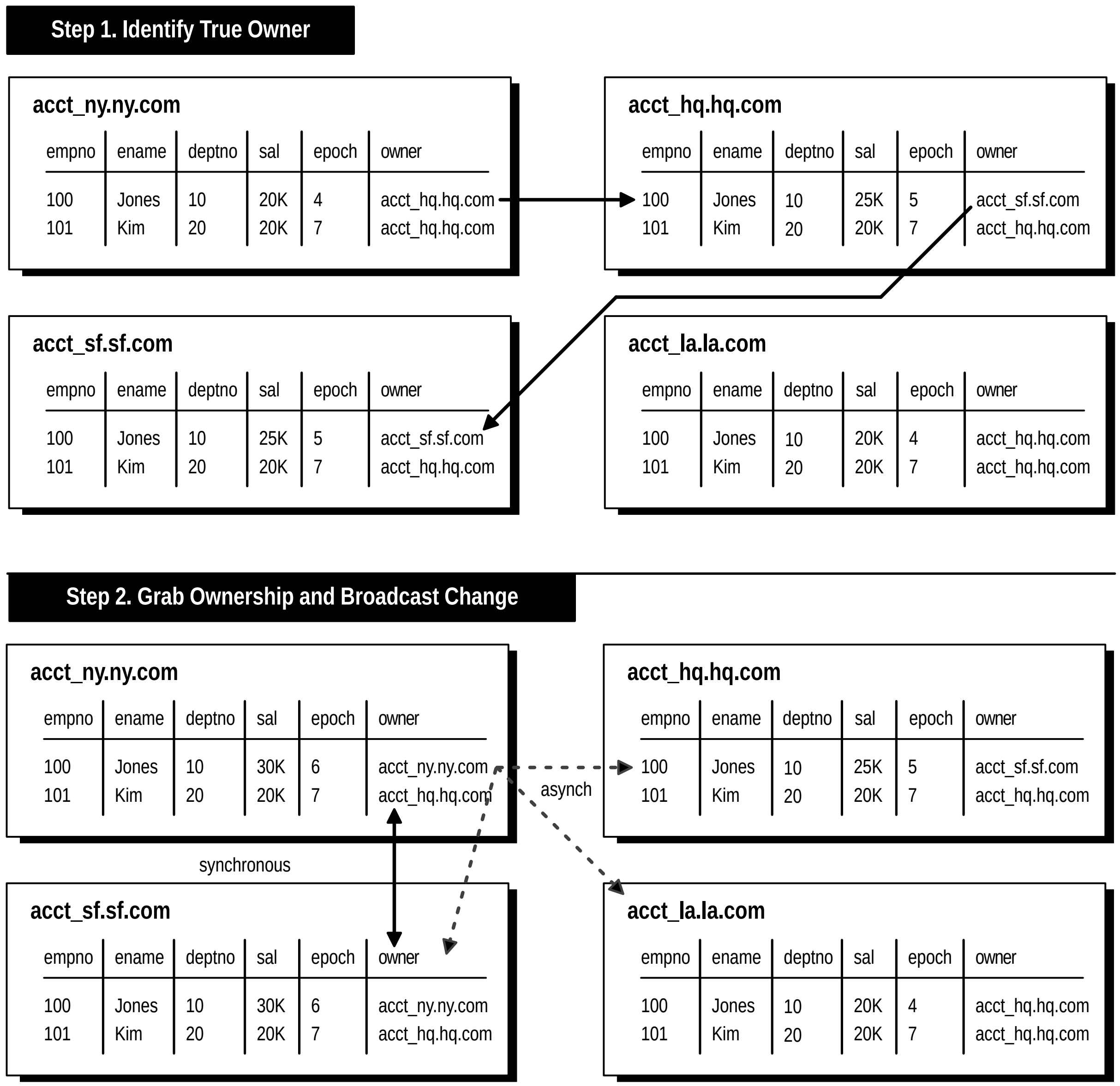
Once you have designed a token passing mechanism, you can use it to implement a variety of forms of dynamic partitioning of data ownership, including workflow.

You should design your application to implement token passing for you automatically. You should not allow the owner or epoch columns to be updated outside this application.

Whenever you attempt to update a row, your application should

1. locate the current owner of the row
2. lock the row to prevent updates while ownership is changing
3. grab ownership of the row
4. perform the update (Oracle releases the lock when you commit your transaction.)

For example, Figure 8 – 2 illustrates how ownership of employee 100 passes from the ACCT_SF database to the ACCT_NY database.



Step 1. Identify True Owner

acct_ny.ny.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	20K	4	acct_hq.hq.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_hq.hq.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	25K	5	acct_sf.sf.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_sf.sf.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	25K	5	acct_sf.sf.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_la.la.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	20K	4	acct_hq.hq.com
101	Kim	20	20K	7	acct_hq.hq.com

Step 2. Grab Ownership and Broadcast Change

acct_ny.ny.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	30K	6	acct_ny.ny.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_hq.hq.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	25K	5	acct_sf.sf.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_sf.sf.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	30K	6	acct_ny.ny.com
101	Kim	20	20K	7	acct_hq.hq.com

acct_la.la.com

empno	ename	deptno	sal	epoch	owner
100	Jones	10	20K	4	acct_hq.hq.com
101	Kim	20	20K	7	acct_hq.hq.com

Figure 8 – 2 Grabbing the Token

Locating the Owner of a Row

To grab ownership, the ACCT_NY database uses a simple recursive algorithm to locate the owner of the row. The pseudo code for this algorithm is shown below:

```
–– Pseudo code for locating the token owner.
–– This is for a table TABLE_NAME with primary key PK.
–– Initial call should initialize loc_epoch to 0 and loc_owner
–– to the local global name.
get_owner(PK IN primary_key_type, loc_epoch IN OUT NUMBER,
          loc_owner IN OUT VARCHAR2)
{
    –– use dynamic SQL (dbms_sql) to perform a select similar to
    –– the following:
    select owner, epoch into rmt_owner, rmt_epoch
         from TABLE_NAME@loc_owner
         where primary_key = PK for update;
    if rmt_owner = loc_owner and rmt_epoch >= loc_epoch then
      loc_owner := rmt_owner;
      loc_epoch := rmt_epoch;
      return;
    elsif rmt_epoch >= loc_epoch then
      get_owner(PK, rmt_epoch, rmt_owner);
      loc_owner := rmt_owner;
      loc_epoch := rmt_epoch;
      return;
    else
      raise_application_error(–20000, ’No owner for row’);
    end if;
}
```

Grabbing Ownership

After locating the owner of the row, the ACCT_NY site grabs ownership from the ACCT_SF site by completing the following steps:

1. Lock the row at the SF site to prevent any changes from occurring while ownership is being exchanged.
2. Synchronously update the owner information at both the SF and NY sites. This ensures that only one site considers itself to be the owner at all times. The update at the SF site should not be replicated using DBMS_REPUTIL.REPLICATION_OFF. The replicated change of ownership at the NY site in step 4 will ultimately be propagated to all other sites in the replicated environment (including the SF site, where it will have no effect).
3. Update the row information at the new owner site, NY, with the information from the current owner site, SF. This data is guaranteed to be the most recent. This time, the change at the NY site should not be replicated. Any queued changes to this data at the SF site will be propagated to all other sites in the usual manner.

When the SF change is propagated to NY, it will be ignored because of the values of the epoch numbers, as described in the next bullet point.

4. Update the epoch number at the new owner site to be one greater than the value at the previous site. Perform this update at the new owner only, and then asynchronously propagate this update to the other master sites. Incrementing the epoch number at the new owner site prevents ordering conflicts.

   When the SF changes (that were in the deferred queue in step 2) are ultimately propagated to the NY site, the NY site will ignore them, because they will have a lower epoch number than the epoch number at the NY site for the same data.

   As another example, suppose the HQ site received the SF changes after receiving the NY changes, the HQ site would ignore the SF changes because the changes applied from the NY site would have the greater epoch number.

Applying the Change

You should design your application to implement this method of token passing for you automatically whenever you perform an update. You should not allow the owner or epoch columns to be updated outside this application. The lock that you grab when you change ownership is released when you apply your actual update. The changed information, along with the updated owner and epoch information, will be asynchronously propagated to the other sites in the usual manner.

Using Procedural Replication

Procedural replication can offer performance advantages for large batch–oriented operations operating on large numbers of rows that can be run serially within your replicated environment.

A good example of an appropriate application is a purge operation (also referred to as an archive operation) run infrequently (for example, once per quarter) during off hours to remove old data, or data that was “logically” deleted from the online database. An example using procedural replication to purge deleted rows is described on page 8 – 20.

Restrictions on Procedural Replication

All parameters for a replicated procedure must be IN parameters; OUT and IN / OUT modes are not supported. The datatypes supported for these parameters are: NUMBER, DATE, VARCHAR2, CHAR, ROWID and RAW.

The symmetric replication facility cannot detect update conflicts produced by replicated procedures. Replicated procedures must detect and resolve conflicts themselves. Because of the difficulties involved in writing your own conflict resolution routines, it is best to simply avoid the possibility of conflicts.

Adhering to the following guidelines will help you ensure that your tables remain consistent at all sites.

- You must disable row–level replication within the deferred procedure as described on page 8 – 18.
- Only one replicated procedure should be executed at a time, as described in the next section, "Serialization of Transactions".
- The replicated procedure must be packaged and the package cannot contain any functions. Standalone deferred procedures and standalone or packaged deferred functions are not currently supported.
- The deferred procedures must reference only locally owned data.
- The procedures should not use locally generated fields or values (for example, the procedure should not call SYSDATE).
- Your data ownership should be statically partitioned (that is, ownership of a row should not change between sites)

Serialization of Transactions

Serial execution ensures that your data remains consistent. The symmetric replication facility propagates and executes replicated transactions one at a time. For example, assume that you have two procedures, A and B, that perform updates on local data. Now assume that you perform the following actions, in order:

1. execute A and B locally
2. queue requests to execute other replicas of A and B on other nodes
3. commit

The replicas of A and B on the other nodes are executed completely serially, in the same order that they were committed at the originating site. If A and B execute concurrently at the originating site, however, they may produce different results locally than they do remotely. Executing A and B serially at the originating site ensures that all sites have identical results. Alternatively, you could write the procedures carefully, to ensure serialization. For example, you could lock any tables being updated in EXCLUSIVE mode to ensure serialization at the originating site.

Generating Support for Replicated Procedures

You must disable row–level replication support at the start of your procedure, and then re–enable support at the end. This ensures that any updates that occur as a result of executing the procedure are not propagated to other sites. Row–level replication is enabled and disabled by calling DBMS_REPUTIL.REPLICATION_ON and DBMS_REPUTIL.REPLICATION_OFF, as described on page 8 – 18.

When you generate replication support for your replicated package, the symmetric replication facility creates a wrapper package. The wrapper package has the same name as the original, but is prefixed with the string that you supplied when you called DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT.

If you did not supply a prefix, the default, "defer_", is used. The wrapper procedure has the same parameters as the original, along with two additional parameters: CALL_LOCAL and CALL_REMOTE. These two boolean parameters determine where the procedure gets executed. When CALL_LOCAL is TRUE, the procedure is executed locally. When CALL_REMOTE is TRUE, the procedure will ultimately be executed at all other sites in the replicated environment.

The remote procedures are called directly if you are propagating changes synchronously, or the calls to these procedures are added to the deferred transaction queue, if you are propagating changes asynchronously. By default, CALL_LOCAL is FALSE, and CALL_REMOTE is TRUE.

Replication support is generated in two phases. The first phase creates the package specification at all sites. Phase two generates the package body at all sites. These two phases are necessary to support synchronous replication.

For example, suppose that you create the package UPDATE containing the procedure UPDATE_EMP, which takes one argument, AMOUNT. You replicate this object to all master sites in your replicated environment by making the following calls:

```
DBMS_REPCAT.CREATE_MASTER_REPOBJECT(
    sname               => 'acct_rec',
    oname               => 'update',
    type                => 'package',
    use_existing_object => FALSE,
    retry               => FALSE,
    copy_rows           => TRUE,
    gname               => 'acct');
```

```
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT(
    sname          => ’acct_rec’,
    oname          => ’update’,
    type           => ’package’,
    package_prefix => ’defer_’,);
```

You would now invoke the replicated procedure as shown below:

```
defer_update.update_emp( amount      => 1.05,
                         call_local  => TRUE,
                         call_remote => TRUE);
```

As shown in Figure 8 – 3, the logic of the wrapper procedure ensures that the procedure is called at the local site and then at all remote sites. The logic of the wrapper procedure also ensures that when the replicated procedure is called at the remote sites, CALL_REMOTE is FALSE, ensuring that the procedure is not further propagated.

If you are operating in a mixed replicated environment with static partitioning of data ownership (that is, if you are not preventing row–level replication), the replication facility will preserve the order of operations at the remote node, since both row–level and procedural replication use the same asynchronous queue.

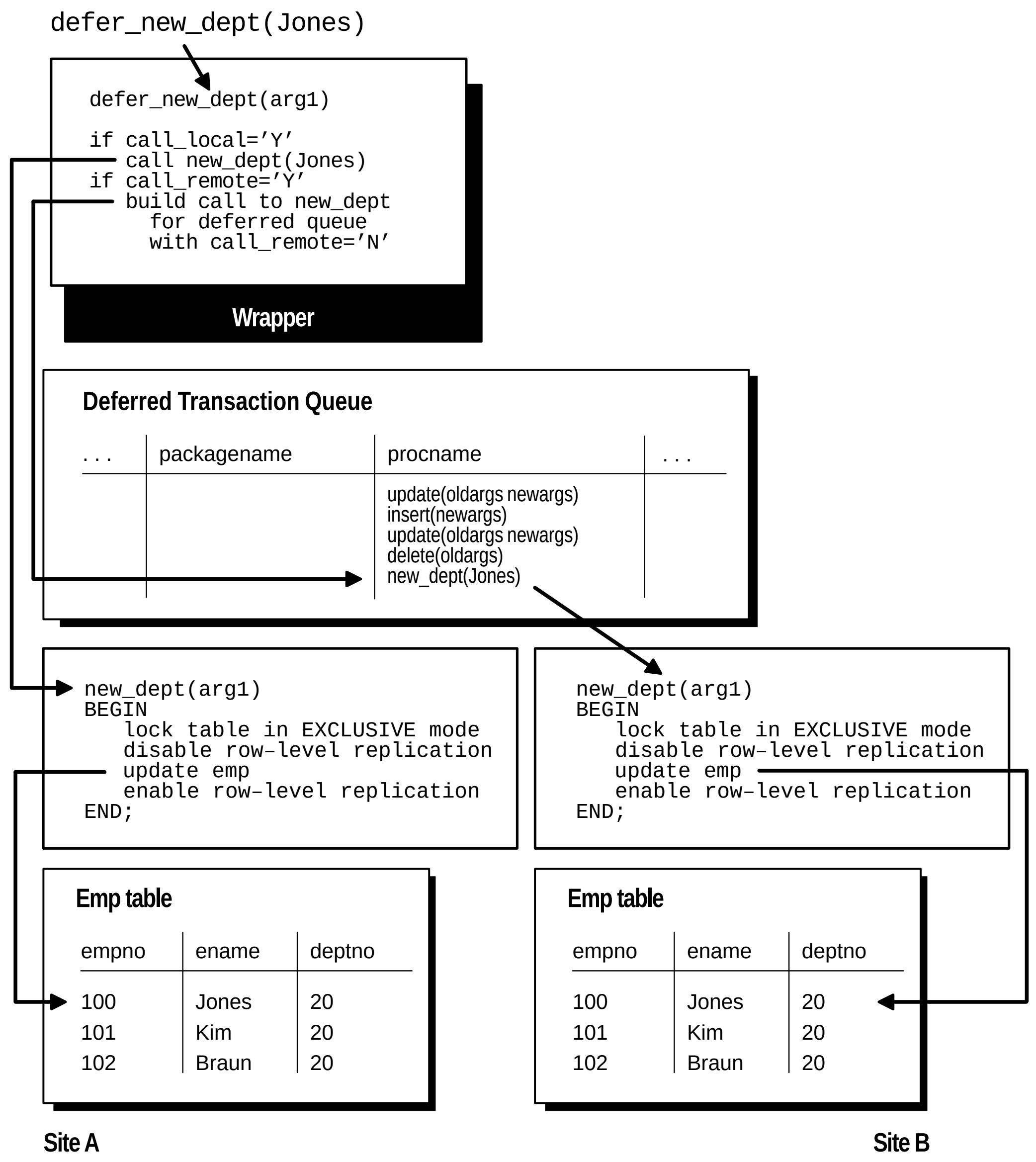


Figure 8 – 3 Asynchronous Procedural Replication

Modifying Tables without Replicating the Modifications

There may be times when you want to make a modification to a replicated object, but you do not want this modification replicated to the other sites in the replicated environment. For example, you might want to disable replication in the following situations:

- When you are using procedural replication to propagate a change, you should always disable row–level replication at the start of your procedure.
- You may need to disable replication in triggers defined on replicated tables to avoid replicating trigger actions multiple times as described on page 8 – 19.
- Sometimes when you manually resolve a conflict, you might not want to replicate this modification to the other copies of the table.

  You might need to do this, for example, if you need to correct the state of a record at one site so that a conflicting replicated update will succeed when you reapply it by calling DBMS_DEFER_SYS.EXECUTE_ERROR. Or you might use an unreplicated modification to undo the effects of a transaction at its origin site because the transaction could not be applied at the destination site. In this example, you would call DBMS_DEFER_SYS.DELETE_ERROR to delete the conflicting transaction from the error tables at the destination site.

To modify tables without replicating the modifications, use the REPLICATION_ON and REPLICATION_OFF procedures in the DBMS_REPUTIL package. These procedures take no arguments and are used as flags by the generated replication triggers.

Note: You must be granted the EXECUTE privilege on the DBMS_REPUTIL package.

Disabling the Symmetric Replication Facility

The DBMS_REPUTIL.REPLICATION_OFF procedure sets the state of the DBMS_REPUTIL.REPLICATION_IS_ON package variable for the current session to FALSE. Because all replicated triggers check the state of this variable before queuing any transactions, modifications made to the replicated tables that use row–level replication do not result in any queued deferred transactions.

☞ **Attention:** Because REPLICATION_IS_ON is a variable in a PL/SQL package specification, its state is session bound. That is, other users logged on to the same schema are not restricted from placing committed changes in the deferred transaction queue.

If you are using procedural replication, you should call REPLICATION_OFF at the start of your procedure, as shown in the following example. This ensures that the symmetric replication facility does not attempt to use row–level replication to propagate the changes that you make.

```
CREATE OR REPLACE PACKAGE update AS
  PROCEDURE update_emp(adjustment IN NUMBER);
END;
/

CREATE OR REPLACE PACKAGE BODY update AS
  PROCEDURE update_emp(adjustment IN NUMBER) IS
  BEGIN
    –– turn off row–level replication for set update
    dbms_reputil.replication_off;
    UPDATE emp . . .;
    –– re–enable replication
    dbms_reputil.replication_on;
  EXCEPTION WHEN OTHERS THEN
     . . .
     dbms_reputil.replication_on;
  END;
END;
```

Re–enabling the Symmetric Replication Facility

After resolving any conflicts, or at the end of your replicated procedure, be certain to call DBMS_REPUTIL.REPLICATION_ON to resume normal replication of changes to your replicated tables or snapshots. This procedure takes no arguments. Calling REPLICATION_ON sets the package variable DBMS_REPUTIL.REPLICATION_IS_ON to TRUE.

Triggers and Replication

If you have defined a replicated trigger on a replicated table, you may need to ensure that the trigger fires only once for each change that you make. Typically, you will only want the trigger to fire when the change is first made, and you will not want the remote trigger to fire when the change is replicated to the remote site.

You should check the value of the DBMS_REPUTIL.FROM_REMOTE package variable at the start of your trigger. The trigger should update the table only if the value of this variable is FALSE.

Alternatively, you can disable replication at the start of the trigger and re–enable it at the end of the trigger when modifying rows other than the one that caused the trigger to fire. Using this method, only the original change is replicated to the remote sites. Then the replicated trigger will fire at each remote site. Any updates performed by the replicated trigger will not be pushed to any other sites.

Using this approach, conflict resolution is not invoked. Therefore, you must ensure that the changes resulting from the trigger do not affect the consistency of the data.

Enabling/Disabling Replication for Snapshots

To disable all local replication triggers for snapshots at your current site, set the I_AM_A_REFRESH package state to TRUE by calling SET_I_AM_A_REFRESH, as shown in the following example:

```
DBMS_SNAPSHOT.SET_I_AM_A_REFRESH(value => TRUE);
```

To re–enable the triggers set the package state to FALSE, as shown below:

```
DBMS_SNAPSHOT.SET_I_AM_A_REFRESH(value => FALSE);
```

To determine the value of the package variable REP$WHAT_AM_I.I_AM_A_SNAPSHOT, call the I_AM_A_REFRESH function as shown below:

```
ref_stat := dbms_snapshot.i_am_a_refresh;
```

To check if a snapshot is refreshing or if a master site has replication turned off, you can also call the ACTIVE function in each table's corresponding $TP package.

Handling Deletes

To avoid encountering delete conflicts, you might find it easiest to mark rows as deleted and purge them later. This section outlines a simple technique for purging these marked rows using procedural replication.

Suppose that your database contains the following MAIL_LIST table:

```
Name                           Null?    Type
------------------------------ -------- --------------
CUSTNO                         NOT NULL NUMBER(4)      PRIMARY KEY
CUSTNAME                                VARCHAR2(10)
ADDR1                                   VARCHAR2(30)
ADDR2                                   VARCHAR2(30)
CITY                                    VARCHAR2(30)
STATE                                   VARCHAR2(2)
ZIP                                     NUMBER(9)
PHONE                                   NUMBER(10)
REMOVE_DATE                             DATE
```

Instead of deleting a customer when he or she requests to be removed from your mailing list, the REMOVE_DATE column would be used to indicate former customers; A NULL value would be used for current customers. After customers request removal from the mailing list, their rows are no longer updated. Such a convention avoids conflicts when the rows are actually deleted sometime later. A view of current customers could be defined as follows:

```
CREATE OR REPLACE VIEW corp.current_mail_list AS
  SELECT custno, custname, addr1, addr2, city, state, zip, phone
    FROM corp.mail_list WHERE remove_date IS NULL;
```

Periodically, perhaps once a year after the holiday sales, the former customers would be purged from the table using the REMOVE_DATE field. Such a delete could be performed using row–level replication just by performing the following delete:

```
DELETE corp.mail_list WHERE remove_date IS NOT NULL AND
                      remove_date < '01–JAN–95';
```

However, for a large company with an extensive mail order business, the number of former customers could be quite large resulting in a lot of undesired network traffic and database overhead. Instead, the procedural replication could be used using the following package:

```
CREATE OR REPLACE PACKAGE corp.purge AS
  PROCEDURE remove_cust(purge_date IN DATE);
END;
/

CREATE OR REPLACE PACKAGE BODY corp.purge AS
  PROCEDURE remove_cust(purge_date IN DATE) IS
  BEGIN
    –– turn off row–level replication for set delete
    dbms_reputil.replication_off;
    –– prevent phantom reads
    LOCK TABLE corp.mail_list IN EXCLUSIVE MODE;
    DELETE corp.mail_list WHERE remove_date IS NOT NULL AND
                                remove_date < purge_date;
    dbms_reputil.replication_on;
  EXCEPTION WHEN other THEN
    dbms_reputil.replication_on;
  END;
END;
```

The DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT procedure would have been used to generate the DEFER_PURGE package during the initial replication setup. Then, the procedural replication package could be called as follows by a single master site:

```
BEGIN
  defer_purge.remove_cust('01–FEB–95','Y');
END;
```

The procedure, PURGE.REMOVE_CUST, would be executed locally and asynchronously executed at each master, resulting in many rows being deleted with only minimal network traffic.

To ensure that there are no outstanding transactions against the rows to be purged, your application should be written to *never* update logically deleted rows and the REMOVE_DATE should be old enough to ensure that the logical delete of the row is propagated before the row is purged. Thus, in the previous example, it is probably not necessary to lock the table in EXCLUSIVE mode; although this is another method of guaranteeing that these rows not be updated during the purge.

Using Connection Qualifiers

Connection qualifiers provide a way to have several database links of the same type (for example, public) that point to the same remote database, yet establish those connections using different communications pathways (for example, an ethernet link or a modem link).

See Chapter 2 of *Oracle7 Server Distributed Systems, Volume I* for information about defining connection qualifiers for a database link.

In a replicated system, after the connection qualifier has been defined as described in *Oracle7 Server Distributed Systems, Volume I*, use the procedure CREATE_MASTER_REPGROUP in the DBMS_REPCAT package to create the master replication object group that will use that connection qualifier.

For example if you have defined the connection qualifier @ETHERNET and want to create the master replicated object group ACCT to use that qualifier:

```
DBMS_REPCAT.CREATE_MASTER_REPGROUP(
        gname          => 'acct',
        group_comment  => 'created by '||user||' on '||SYSDATE,
        master_comment => 'created by '||user||' on '||SYSDATE,
        qualifier      => '@ethernet')
```

When you create the snapshot replication group, use the CREATE_SNAPSHOT_REPGROUP procedure:

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPGROUP(
     gname             => 'acct';
     master            => 'acct_hq.hq.com',
     comment           => 'created on ...',
     propagation_mode  => 'asynchronous',
     qualifier         => 'acct_hq.hq.com@ethernet');
```

After you generate replication support, replication for this site will occur via the ethernet connection specified by the connection qualifier @ETHERNET. The complete database link, with qualifier would look like this in a select statement:

```
SELECT * FROM emp@acct_hq.hq.com@ethernet;
```

If you have a concurrent modem link to ACCT_HQ.HQ.COM, you can also define a connection qualifier, @MODEM, and use the procedures above to create a master replication group and snapshot sites that update via the modem link. A select statement using the @MODEM connection qualifier would look like:

```
SELECT * FROM emp@acct_hq.hq.com@modem;
```

☞ **Attention:** If you plan to use connection qualifiers, you will probably need to increase the value of the INIT.ORA parameter, OPEN_LINKS. The default is four open links per process. You will need to estimate the required value based on your usage. See the *Oracle7 Server Reference* for more information about the parameter OPEN_LINKS.

CHAPTER

9

Troubleshooting

This chapter describes several common problems that you may encounter when using the symmetric replication facility, and suggested resolutions. The topics include:

- using the DBA_REPCATLOG view to diagnose problems.
- disabling the job queue while diagnosing problems.
- diagnosing problems with job queues.
- diagnosing problems with the propagation of changes between sites in a replicated environment.
- diagnosing problems with snapshots.

Note: Often when diagnosing a replication problem, you will need to consult one or more data dictionary views. For the replication catalog views, always use the SYS.DBA_*name* views if you are authorized. Otherwise, use either the ALL_*name* or USER_*name* views. The views for deferred remote procedure calls are all owned by SYS and have no prefix.

Diagnosing Problems with DBA_REPCATLOG Entries

The DBA_REPCATLOG view shows the interim and final status for asynchronous administrative activities. You should examine this table before enabling a replication environment with RESUME_MASTER_ACTIVITY. You should also examine it whenever you suspect replication administration problems. The master definition site uses its DBA_REPCATLOG view to record both local and remote activities. Each of these activities is explained below.

For each local activity, there is a row in the master definition site's DBA_REPCATLOG view. The STATUS column in this row begins with the value READY. If the activity completes normally, the row is deleted from the DBA_REPCATLOG view. If the activity encounters a problem, the Oracle error number is captured in the ERRNUM column and the error message is captured in the MESSAGE column. These columns are helpful when diagnosing symmetric replication problems.

For a remote activity, the symmetric replication facility creates two rows that appear in the DBA_REPCATLOG view: one at the master definition site with a STATUS value of AWAIT_CALLBACK, and one at the remote master with a STATUS value of READY. What happens to these two log rows depends on whether the remote activity completes normally.

- If the remote activity completes normally, the remote master updates its row in the DBA_REPCATLOG view to DO_CALLBACK and commits. When it establishes communication with the master definition site, the remote master deletes the associated row from its DBA_REPCATLOG view and the corresponding row at the master definition site and commits.
- If the remote activity encounters a problem, the remote master updates the ERRNUM and MESSAGE columns and sets the STATUS column to ERROR for the associated row in the DBA_REPCATLOG view for the remote master. When it establishes communication with the master definition site, the remote master updates the row in the DBA_REPCATLOG view at the master definition site with the local ERRNUM, MESSAGE, and STATUS values. Then the remote master deletes the associated row in its DBA_REPCATLOG view and commits.

DO_DEFERRED_REPCAT_ADMIN executes the requests in the local DBA_REPCATLOG submitted by the user that invoked DO_DEFERRED_REPCAT_ADMIN in the order determined by the ID column. When DO_DEFERRED_REPCAT_ADMIN is executed at a master that is not the master definition site, it does as much as possible.

Some asynchronous activities such as populating a replicated table require communication with the master definition site. If this communication is not possible, DO_DEFERRED_REPCAT_ADMIN stops executing rows from DBA_REPCATLOG to avoid executing DBA_REPCATLOG rows out of order. Some communication with the master definition site, such as the final step of updating or deleting a DBA_REPCATLOG row at the master definition site, can be deferred and will not prevent DO_DEFERRED_REPCAT_ADMIN from executing additional rows in the DBA_REPCATLOG.

Entries Not Removed from Log

Occasionally, you may notice that an entry in the DBA_REPCATLOG view is not removed as anticipated, yet the STATUS is not ERROR. Here are some items to check if the symmetric replication facility does not appear to be working properly.

- Ensure that there is not a problem with the execution of the job queue. Job queue errors are described on page 9 – 4.

  Submit a trivial job at the master site to ensure that it runs as expected. In a newly created database, jobs are not automatically enabled until the database is shut down and restarted. You can call DBMS_IJOB.SET_ENABLED(TRUE) to avoid restarting the database. (Note the I, for internal, in DBMS_IJOB.)

  Additionally, check the LOG_USER column in the DBA_JOBS view to ensure that the replication job is being run on behalf of the replication administrator. Check the USERID column of the DBA_REPCATLOG view to ensure that the replication administrator was the user that submitted the request. DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN only performs those administrative requests submitted by the user that calls this procedure.

- Ensure the relevant databases are running and communication is possible.

- Ensure that you have the necessary private database link for the symmetric replication facility, and that the user designated in the CONNECT TO clause (generally the surrogate replication administrator) has the necessary privileges, as described on page 4 – 7. Note that database links are required in both directions, and that privileges must be granted at both sites.

- Ensure that you have the necessary private database link for the replication administrator, and that the replication administrator has been granted the necessary privileges, as described on page 4 – 4. Note that database links are required in both directions, and that privileges must be granted at both sites.

Disabling Job Queues

When diagnosing a replication problem, you may find it useful to disable the job queue at one or more masters. To do this, shut down the master and restart it with a value of zero for JOB_QUEUE_PROCESSES. To avoid restarting the database, you can call DBMS_IJOB.SET_ENABLED(FALSE). (Note the I, for internal, in DBMS_IJOB.) You must then connect to the master site and execute the procedure DO_DEFERRED_REPCAT_ADMIN, to execute asynchronous administrative activities at that master. This lets you have better control over the execution time of administrative activities.

Diagnosing Problems with Job Queues

If a job is not executed as expected, check the following:

- Check the NEXT_DATE value in the DBA_JOBS view to ensure that the job was properly scheduled for execution.
- After determining that the job execution interval has passed, check the FAILURES and BROKEN values in the DBMS_JOBS view.
  - If the job failed to execute, check the alert log and trace files for error information and fix the error. If the job was not broken, it will ultimately be re–executed. If the job was broken, or if you want to force immediate re–execution of the job, call DBMS_JOB.RUN after fixing the job.
  - If the job was never executed, there may be a problem with the availability of background processes. Check the initialization parameter JOB_QUEUE_PROCESSES to determine the maximum number of background processes available and JOB_QUEUE_INTERVAL to determine how frequently each background processes wakes up. The DBA_JOBS_RUNNING view describes what jobs these processes are currently running (you may have a problem with a runaway job), and the alert log and trace file can provide you with additional information about potential problems with the background process.

Diagnosing Problems with Master Sites

Replicated Objects Not Created at New Master Site

If you add a new master site to your replicated environment, and the appropriate replicated objects are not created at the new site, try the following:

- Ensure that the necessary private database links exist between the new master site and the existing master sites, as described on page 4 – 4. You must have links both to the new site from each existing site, and from the new site to each existing site.
- Re–execute DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN at the new master site.

DDL Changes Not Propagated to Master Site

If you call a procedure in the DBMS_REPCAT package to make a schema–level change at the master definition site that is not propagated to a master site, try the following:

- Examine the DBA_REPCATLOG at the master site and at the master definition site.
- Call DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN at the master site to force this change to be applied at the new site.

DDL submitted to repcat executes on behalf of the user who submits the DDL. When a DDL statement applies to an object in a schema other than the submitter's schema, the submitter needs appropriate privileges to execute the statement. In addition, the statement must explicitly name the schema. For example, assume that you, the replication administrator, supply the following as the ddl_text parameter to the DBMS_REPCAT.CREATE_MASTER_REPOBJECT procedure:

```
CREATE TABLE scott.new_emp AS SELECT * FROM hr.emp WHERE...;
```

Because each table name contains a schema name, this statement works whether the replication administrator is SCOTT, HR, or another user—as long as the administrator has the required privileges.

Suggestion: Qualify the name of every schema object with the appropriate schema.

DML Changes Not Asynchronously Propagated to Other Sites

If you make an update to your data at a master site, and that change is not properly asynchronously propagated to the other sites in your replicated environment, try the following:

- Check the NEXT_DATE value in the DefSchedule view to determine if the changes have been scheduled to be propagated.

- Call DBMS_DEFER_SYS.EXECUTE to force the execution of the DML, or call DBMS_DEFER_SYS.SCHEDULE_EXECUTION to schedule execution at a periodic interval. If the scheduled execution interval has passed, check the DefSchedule view for the job number and follow the instructions for diagnosing problems with job queues on page 9 – 4.
- If you determine that the change was not propagated because of an error, you can also check the DefError view at the destination site to determine the cause of the error. You might also check the DefTran and DefCall views for more information. Additional instructions on how to determine the cause of an error logged in the DefError view are included on page 7 – 2.

DML Cannot be Applied to Replicated Table

If you receive the DEFERRED_RPC_QUIESCE exception when you attempt to modify a replicated table, one or more replicated object groups at your local site are "quiescing" or "quiesced". To proceed, your replication administrator must either call DBMS_REPCAT.RESUME_MASTER_ACTIVITY, or DBMS_REPCAT.DROP_MASTER_REPSCHEMA for each quiesced, replicated object group.

Bulk Updates and Constraint Violations

A single update statement applied to a replicated table can update zero or more rows. The update statement causes zero or more update requests to be queued for deferred execution, one for each row updated. This distinction is important when constraints are involved, because Oracle effectively performs constraint checking at the end of each statement. While a bulk update might not violate a uniqueness constraint, for example, some equivalent sequence of individual updates might violate uniqueness.

If the ordering of updates is important, update one row at a time in an appropriate order. This lets you define the order of the update requests in the deferred RPC queue.

Re–creating a Replicated Object

If you replicate an object that already exists at the master definition site with DBMS_REPCAT.CREATE_MASTER_REPOBJECT, the status of the object must be VALID. If the status is INVALID, recompile the object, or drop and recreate the object. Then invoke CREATE_MASTER_REPOBJECT with the RETRY argument set to TRUE.

Unable to Generate Replication Support for a Table

When you call GENERATE_REPLICATION_SUPPORT for a replicated table, Oracle generates a trigger at the local site. If the table will be propagating changes asynchronously, this trigger uses the DBMS_DEFER package to build the calls that are placed in the local deferred transaction queue. EXECUTE privileges for most of the packages involved with symmetric replication, such as DBMS_REPCAT and DBMS_DEFER, need to be granted to replication administrators and users that own replicated objects. The DBMS_REPCAT_ADMIN package performs the grants needed by the replication administrators for many typical replication scenarios. When the owner of a replicated object is not a replication administrator, however, you must explicitly grant EXECUTE privilege on DBMS_DEFER to the object owner.

Problems with Replicated Procedures or Triggers

If you discover an unexpected unresolved conflict, and you were mixing procedural and row–level replication on a table, carefully review the procedure to ensure that the replicated procedure did not cause the conflict. Ensure that ordering conflicts between procedural and row–level updates are not possible. Check if the replicated procedure locks the table in EXCLUSIVE mode before performing updates (or uses some other mechanism of avoiding conflicts with row–level updates). Check that row–level replication is disabled at the start of the replicated procedure and re–enabled at the end. Ensure that row–level replication is re–enabled even if exceptions occur when the procedure executes. In addition, check to be sure that the replicated procedure executed at all master sites. You should perform similar checks on any replicated triggers that you have defined on replicated tables.

Diagnosing Problems with the Deferred Transaction Queue

When you call DBMS_DEFER_SYS.SCHEDULE_EXECUTION, Oracle adds this job to the job queue. If you have scheduled your transaction queue to be pushed at a periodic interval, and you encounter a problem, you should first be certain that you are not experiencing a problem with the job queue. For information on diagnosing job queue problems, see page 9 – 4.

When the symmetric replication facility pushes a deferred transaction to a remote site, it uses a distributed transaction to ensure that the transaction has been properly committed at the remote site before the transaction is removed from the queue at the local site.

For information on diagnosing problems with distributed transactions (two–phase commit), see *Oracle7 Server Distributed Systems, Volume I.*

If you notice that transactions are not being pushed to a given remote site, you may have a problem with how you have specified the destination for the transaction. If you specify a destination database when you call DBMS_DEFER_SYS.SCHEDULE_EXECUTION (using the DBLINK parameter), or DBMS_DEFER_SYS.EXECUTE using the DESTINATION parameter), you must provide the full database link. These procedures do not expand the database link name.

Having the wrong view definitions can lead to erroneous deferred transaction behavior. The DEFCALLDEST and DEFTRANDEST views are defined differently in CATDEFER.SQL and CATREPC.SQL. The definitions in CATREPC.SQL should be used whenever symmetric replication is used. If CATDEFER.SQL is ever (re)loaded, ensure that the view definitions in CATREPC.SQL are subsequently loaded.

Diagnosing Problems with Snapshots

Problems Creating Replicated Objects at Snapshot Site

If you unsuccessfully attempt to create a new replicated object at a snapshot site, try the following:

- For updatable snapshots, check that the associated master table has a master snapshot log.
- Make sure that you have the necessary privileges to create the object. The privileges to create an updatable snapshot as a replicated object are listed on page 5 – 9.
- If you are still unsuccessful, try passing the CREATE SNAPSHOT text to the CREATE_SNAPSHOT_REPOBJECT procedure as the DDL_TEXT parameter, instead of precreating the snapshot.

Problems with Snapshot Refresh

If you have a problem refreshing a snapshot, try the following:

- Check the NEXT value in the DBA_SNAPSHOTS view to determine if the refresh has been scheduled.
- If the refresh interval has passed, check the DBA_REFRESH view for the associated job number for the snapshot refresh and then follow the instructions for diagnosing a problem with job queues on page 9 – 4.

- You may also encounter an error if you attempt to define a master detail relationship between two snapshots. You should define master detail relationships only on the master tables by using declarative referential integrity constraints; the related snapshots should then be placed in the same refresh group to preserve this relationship.
- If you encounter a situation where your snapshots are being continually refreshed, you should check the refresh interval that you specified. This interval is evaluated before the snapshot is refreshed. If the interval that you specify is less than the amount of time it takes to refresh the snapshot, the snapshot will be refreshed each time the SNP background process checks the queue of outstanding jobs.
- If there are any outstanding conflicts recorded in the DefError view at the master site for the snapshots, you can only refresh the snapshots by setting the parameter REFRESH_AFTER_ERRORS to TRUE. This parameter can be set when you call DBMS_SNAPSHOT.REFRESH, DBMS_REFRESH.MAKE, or DBMS_REFRESH.CHANGE.
- If your snapshot logs are growing too large, see the section on managing snapshot space log use on page 3 – 11.
- Snapshots in the same refresh groups will have their rows updated in a single transaction. Such a transaction can be very large, requiring either a large rollback segment at the snapshot site (with the rollback segment specified to be used during refresh) or you will need to use more frequent refreshes to reduce the transaction size.
- If Oracle error ORA–12004 occurs, the master site may have run out of rollback segments when trying to maintain the snapshot log, or the snapshot log may be out of date (for example, it may have been purged or recreated, see page 3 – 9).
- Complete refreshes of a single table internally us the TRUNCATE feature to increase speed and reduce rollback segment requirements. However, until the snapshot refresh is complete, users may temporarily see no data in the snapshot. Refreshes of multiple snapshots (for example, refresh groups) do not use the TRUNCATE feature.

- Reorganization of the master table (for example, to reclaim system resources) should TRUNCATE the master table to force snapshots to do complete refreshes, otherwise, the snapshots will have incorrect references to master table ROWIDs. For more information see page 3 – 6

See also page 3 – 18 for additional troubleshooting information.

CHAPTER

10 Using Job Queues

This chapter describes how to use job queues to schedule periodic execution of PL/SQL code.

This chapter discusses

- the initialization parameters you need to set to use job queues
- the DBMS_JOB package and its procedures that allow you to manage jobs in the job queue
- troubleshooting problems with job execution
- the data dictionary views you can use to examine job queue information

Note: The job queue facility is part of the standard Oracle7 Server product, and requires no special options. The symmetric replication facility makes extensive use of job queues to propagate changes between sites, and this information is provided in this document as a convenience to you.

Attention: Remember to `commit` you job queue changes.

Additional Information: See the *Oracle7 Server Administrator's Guide* for more detailed information about job queues.

Job Queues

Using the job queue, you can schedule routines to be performed periodically. A routine is any PL/SQL code.

To schedule a job, you submit it to the job queue and specify the frequency at which the job is to be run. You can also alter, disable, or delete jobs you have submitted.

SNP Background Processes

Background processes, called *SNP background processes*, execute job queues.

SNP processes periodically wake up and execute any queued jobs that are due to be run. You must have at least one SNP process running to execute your queued jobs in the background.

SNP background processes differ from other Oracle background processes in that the failure of an SNP process does not cause the instance to fail. If an SNP process fails, Oracle restarts it.

Multiple SNP Processes

An instance can have up to thirty–six SNP processes, named SNP0 to SNP9 and SNPA to SNPZ. If an instance has multiple SNP processes, the task of executing queued jobs can be shared across these processes, thus improving performance. Note, however, that each job is run at any point in time by only one process. A single job cannot be shared simultaneously by multiple SNP processes.

Starting Up SNP Processes

The job queue initialization parameters allow you to control the operation of the SNP background processes. Set these parameters in the initialization parameter file for an instance. They take effect the next time you start the instance.

Table 10 – 1 describes the job queue initialization parameters.

Parameter Name	Description	
JOB_QUEUE_PROCESSES	Default:	0
	Range of values:	0...36
	Multiple instances:	can have different values
	Sets the number of SNP background processes per instance.	
JOB_QUEUE_INTERVAL	Default:	60 (seconds)
	Range of values:	1...3600 (seconds)
	Multiple instances:	can have different values
	Sets the interval between wake–ups for the SNP background processes of the instance.	

Table 10 – 1 Job Queue Initialization Parameters

Using Job Queues

To schedule and manage jobs in the job queue, use the procedures in the DBMS_JOB package.

DBMS_JOB Package

Table 10 – 2 lists the job queue procedures in the DBMS_JOB package.

Procedure	Description	Discussed on
SUBMIT	Submits a job to the job queue.	page 10 – 5
REMOVE	Removes specified job from the job queue.	page 10 – 9
CHANGE	Alters a specified job. You can alter the job description, the time at which the job will be run, or the interval between executions of the job.	page 10 – 9
WHAT	Alters the job description for a specified job.	page 10 – 9

Table 10 – 2 Procedures in the DBMS_JOB Package, continued on next page

Procedure	Description	Discussed on
NEXT_DATE	Alters the next execution time for a specified job.	page 10 – 9
INTERVAL	Alters the interval between executions for a specified job.	page 10 – 9
BROKEN	Disables or enables job execution. If a job is marked as broken, Oracle does not attempt to execute it.	page 10 – 10
RUN	Forces a specified job to run.	page 10 – 11

Table 10 – 2 Procedures in the DBMS_JOB Package

Privileges and Job Queues

There are no database privileges associated with using job queues. Any user who can execute the job queue procedures can use the job queue.

Environment of a Job

When you submit a job to the job queue or alter a job's definition, Oracle records certain characteristics of your current environment. These characteristics include the following:

- current user
- user submitting or altering the job
- the current schema
- MAC privileges (if appropriate)
- the following NLS parameters
  - NLS_LANGUAGE
  - NLS_TERRITORY
  - NLS_CURRENCY
  - NLS_ISO_CURRENCY
  - NLS_NUMERIC_CHARACTERS
  - NLS_DATE_FORMAT
  - NLS_DATE_LANGUAGE
  - NLS_SORT

Note that the NLS_LANGUAGE and NLS_TERRITORY parameters are used as the basis for defaults of unspecified NLS parameters.

Every time a job is executed, Oracle restores the job's environment. Thus, jobs run in the same environment in which they were submitted.

A job can change its environment by using the DBMS_SQL package and the SQL command ALTER SESSION. For more information about the DBMS_SQL package, see the *Oracle7 Server Application Developer's Guide*. For more information about the ALTER SESSION command, see the *Oracle7 Server Administrator's Guide*.

Import/Export and Job Queues

Jobs can be exported and imported. Thus, if you define a job in one database, you can transfer it to another database. When exporting and importing jobs, the job's number, environment, and definition remain unchanged.

☞ **Attention:** If the job number of a job you want to import matches the number of a job already existing in the database, you will not be allowed to import that job. Submit the job as a new job in the database.

Submitting a Job to the Job Queue

To submit a new job to the job queue, use the SUBMIT procedure in the DBMS_JOB package. The SUBMIT procedure returns the number of the job you submitted.

Submitting a Job: Example

Submit a new job to the job queue. The job calls the procedure DBMS_DDL.ANALYZE_OBJECT to generate optimizer statistics for the table VALERIE.ACCOUNTS. The statistics are based on a sample of half the rows of the ACCOUNTS table. The job is run every 24 hours.

```
SVRMGR> VARIABLE jobno number;
SVRMGR> begin
     2>          DBMS_JOB.SUBMIT(:jobno,
     3>                  'dbms_ddl.analyze_object(''TABLE'',
     4>                  ''VALERIE'', ''ACCOUNTS'',
     5>                  ''ESTIMATE'', NULL, 50);',
     6>                  SYSDATE, 'SYSDATE + 1');
     7> commit;
     8> end;
     9> /
Statement processed.
SVRMGR> print jobno
JOBNO
----------
     14144
```

Owner of a Job

When a user submits a job to the job queue, Oracle specifies that user as the owner of the job. Only a job's owner can alter the job, force the job to run, or remove the job from the queue. Ideally, the owner of the job will be the replication administrator, however, if it is necessary that a user submit the job, the user must have the appropriate privileges at the remote site.

In a deferred transaction, note that, unless explicitly specified otherwise, the user who initiated a transaction will determine which database link is used. The owner can be specified for a deferred transaction by using the DEFER_SYS.EXECUTE parameter, execute_as_user. See page 12 – 15.

Job Numbers

A queued job is identified by its job number. When you submit a job, its job number is automatically generated from the sequence SYS.JOBSEQ.

Once a job is assigned a job number, that number does not change. Even if the job is exported and imported, its job number remains the same.

Job Definitions

The *job definition* is the PL/SQL code specified in the WHAT parameter of the SUBMIT procedure.

☞ **Attention:** In the job definition, use two single quotation marks around strings. Always include a semicolon at the end of the job definition.

Normally, the job definition is a single call to a procedure. The procedure call can have any number of parameters.

There are special parameter values that Oracle recognizes in a job definition. Table 10 – 3 lists these parameters.

Parameter	Mode	Description
`job`	IN	The number of the current job.
`next_date`	IN/OUT	The date of the next execution of the job.
`broken`	IN/OUT	Status of job, broken or not broken. The IN value is FALSE.

Table 10 – 3 Special Parameter Values for Job Definitions

Job Definitions: Examples

The following examples show valid job definitions:

```
'myproc(''10–JAN–82'', next_date, broken);'
'scott.emppackage.give_raise(''KANE'', 3000.00);'
'dbms_job.remove(job);'
```

Job Execution Interval

Immediately before a job is executed, the INTERVAL date function is evaluated. If the job completes successfully, the date calculated from INTERVAL becomes the new NEXT_DATE. If the INTERVAL date function evaluates to NULL and the job completes successfully, the job is deleted from the queue.

If a job should be executed periodically at a set interval, use a date expression similar to `'SYSDATE + 7'` in the INTERVAL parameter. For example, if you set the execution interval to `'SYSDATE + 7'` on Monday, but for some reason (such as a network failure) the job is not executed until Thursday, `'SYSDATE + 7'` now evaluates to every Thursday, not Monday.

If you always want to automatically execute a job at a specific time, regardless of the last execution (for example, every Monday), the INTERVAL and NEXT_DATE parameters should specify a date expression similar to `'NEXT_DAY(TRUNC(SYSDATE), ''MONDAY'')'`.

Table 10 – 4 lists sample date expressions used for job execution intervals.

Date Expression	Evaluation
`'SYSDATE + 7'`	7 days after the last execution
`'SYSDATE + 1/48'`	30 minutes after the last execution
`'SYSDATE + 1/8640'`	10 seconds after the last execution
`'NEXT_DAY(TRUNC(SYSDATE), ''MONDAY'') + 15/24'`	Every Monday at 3PM
`'NEXT_DAY(ADD_MONTHS (TRUNC(SYSDATE, ''Q''), 3), ''THURSDAY'')'`	First Thursday of each quarter

Table 10 – 4 Common Job Execution Intervals

Attention: When specifying NEXT_DATE or INTERVAL, remember that date literals and date strings must be enclosed in single quotation marks. Also, the value of INTERVAL must be enclosed in single quotes.

Additional Information: The parameters for the DBMS_JOBS.SUBMIT procedure are described in Table 12 – 36.

How Jobs Execute

SNP background processes execute jobs. To execute a job, the process creates a session to run the job.

When an SNP process runs a job, the job is run

- in the same environment in which it was submitted
- with the owner's default privileges

When you force a job to run using the procedure DBMS_JOB.RUN, the job is run by your user process. When your user process runs a job, it is run with your directly granted privileges only. Privileges granted to you through roles are unavailable.

Job Queue Locks

Oracle uses job queue locks to ensure that a job is executed by only one session at a time. When a job is being run, its session acquires a job queue (JQ) lock for that job.

For more information about locking, see *Oracle7 Server Concepts*.

Interpreting Information about JQ Locks

The following query lists the session identifier, lock type, and lock identifiers for all sessions holding JQ locks:

```
SVRMGR> SELECT sid, type, id1, id2
     2>     FROM v$lock
     3>     WHERE type = 'JQ';

SID        TY ID1        ID2
---------- -- ---------- ----------
        12 JQ          0      14144
1 row selected.
```

In the query above, the identifier for the session holding the lock is 12. The ID1 lock identifier is always 0 for JQ locks. The ID2 lock identifier is the job number of the job the session is running.

For more information about the locking views, see the *Oracle7 Server Reference*.

Database Links and Jobs

If the job that you submit uses a database link, the link must include a username and password. Anonymous database links will not succeed.

Removing a Job From the Job Queue

To remove a job from the job queue, use the REMOVE procedure in the DBMS_JOB package.

Removing a Job: Example

Remove job number 14144 from the job queue.

```
DBMS_JOB.REMOVE(14144);
```

Restrictions

You can remove currently executing jobs from the job queue. However, the job will not be interrupted, and the current execution will be completed.

You can remove only jobs you own. If you try to remove a job that you do not own, you receive a message that states the job is not in the job queue.

Additional Information: The parameters for the procedure DBMS_JOB.REMOVE are described in Table 12 – 34.

Altering a Job

To alter a job that has been submitted to the job queue, use the procedures CHANGE, WHAT, NEXT_DATE, or INTERVAL in the DBMS_JOB package.

Restrictions

You can alter only jobs that you own. If you try to alter a job that you do not own, you receive a message that states the job is not in the job queue.

☞ **Attention:** When you change a job's definition using the WHAT parameter in the procedure CHANGE or WHAT, Oracle records your current environment. This becomes the new environment for the job.

Altering a Job: Example

In the following example, the job identified by 14144 is now executed every three days. By specifying NULL for WHAT and NEXT_DATE, these values remain unchanged.

```
DBMS_JOB.CHANGE( job       => 14144,
                 what      => null,
                 next_date => null,
                 interval  => ’SYSDATE + 3’);
```

You could make this same change by calling the INTERVAL procedure, as shown in the following example:

```
DBMS_JOB.INTERVAL( job      => 14144,
                   interval => ’SYSDATE + 3’);
```

Additional Information: The parameters for these procedures are described in the following tables:

CHANGE	Table 12 – 31
WHAT	Table 12 – 37
NEXT_DATE	Table 12 – 33
INTERVAL	Table 12 – 32

Broken Jobs

A job is labeled as either broken or not broken. Oracle does not attempt to run broken jobs. However, you can force a broken job to run by calling the procedure DBMS_JOB.RUN.

When you submit a job, it is considered not broken.

How Jobs Become Broken

There are two ways a job can break:

- Oracle has failed to successfully execute the job after 16 attempts.
- You have marked the job as broken, using the procedure DBMS_JOB.BROKEN.

Marking a Job as Broken or Not Broken

To mark a job as broken or not broken, use the procedure BROKEN in the DBMS_JOB package.

Marking a Job as Not Broken: Example

The following example marks job 14144 as not broken and sets its next execution date to the following Monday:

```
DBMS_JOB.BROKEN( job       => 14144,
                 broken    => FALSE,
                 next_date => NEXT_DAY(SYSDATE, 'MONDAY'));
```

Once a job has been marked as broken, Oracle will not attempt to execute the job until you either mark the job as not broken, or force the job to be executed by calling the procedure DBMS_JOB.RUN.

Restrictions

You can mark only jobs you own as broken. If you try to mark a job you do not own, you receive a message that states the job is not in the job queue.

Additional Information: The parameters for DBMS_JOB.BROKEN are described in Table 12 – 30.

Forcing a Job to Be Executed

There may be times when you would like to manually execute a job. For example, if you have fixed a broken job, you may want to test the job immediately by forcing it to execute.

To force a job to be executed immediately, use the procedure RUN in the DBMS_JOB package. Oracle attempts to run the job, even if the job is marked as broken.

When you run a job using DBMS_JOB.RUN, Oracle recomputes the next execution date. For example, if you create a job on a Monday with a NEXT_DATE value of 'SYSDATE' and an INTERVAL value of 'SYSDATE + 7', the job is run every seven days starting on Monday. However, if you execute RUN on Wednesday, the next execution date will be the next Wednesday.

Forcing a Job to be Executed: Example

The following example runs job 14144 in your session and recomputes the next execution date:

```
DBMS_JOB.RUN( job => 14144);
```

Restrictions

You can run only jobs that you own. If you try to run a job that you do not own, you receive a message that states the job is not in the job queue.

The procedure RUN contains an implicit commit. Once you execute a job using RUN, you cannot rollback.

Running Broken Jobs

When you run a job that has been marked as broken and the job completes successfully, Oracle relabels the job as not broken. Oracle also resets its count of the number of failed executions for the job.

For more information about viewing the number of failures for a job and its status, see "Viewing Job Queue Information" on page 10 – 13.

Additional Information: The parameters for the procedure DBMS_JOB.RUN are described in Table 12 – 35.

Troubleshooting Job Execution Problems

Several factors can prevent the successful execution of queued jobs; for example:

- not having any SNP background processes to run the job
- a network or instance failure
- an exception when executing the job

Note: If a job fails, Oracle rolls back the job's current transaction (if any such transaction exists).

Recording Errors

When a job fails, information about the failure is recorded in a trace file and the alert log. Oracle writes message number ORA–12012 and includes the job number of the failed job.

Job Failure and Execution Times

If a job returns an error while Oracle is attempting to execute it, Oracle tries to execute it again. The first attempt is made after one minute, the second attempt after two minutes, the third after four minutes, and so on, with the interval doubling between each attempt. When the retry interval exceeds the execution interval, Oracle continues to retry the job at the normal execution interval. However, if the job fails 16 times, Oracle automatically marks the job as broken and no longer tries to execute it.

Thus, if you can correct the problem that is preventing a job from running before the job has failed 16 times, Oracle will eventually run that job again.

Running a Job That Oracle Has Marked as Broken

If a problem has caused a job to fail 16 times, Oracle marks the job as broken. Once you have fixed this problem, you can run the job by either

- forcing the job to run by calling DBMS_JOB.RUN, or
- marking the job as not broken by calling DBMS_JOB.BROKEN and waiting for Oracle to execute the job.

If you force the job to run by calling the procedure DBMS_JOB.RUN, Oracle runs the job immediately. If the job succeeds, then Oracle labels the job as not broken and resets its count of the number of failed executions for the job.

Once you reset a job's broken flag (by calling either RUN or BROKEN), job execution resumes according to the scheduled execution intervals set for the job.

Killing a Job

You can kill a running job by marking the job as broken, identifying the session running the job, and disconnecting that session. You should mark the job as broken so that Oracle does not attempt to run the job again.

Once you have identified the session running the job, you can disconnect the session using SQL command ALTER SYSTEM.

For examples of viewing information about jobs and sessions, see "Viewing Job Queue Information" on page 10 – 13.

Viewing Job Queue Information

The following data dictionary views display information about jobs in the job queue:

- DBA_JOBS
- USER_JOBS
- DBA_JOBS_RUNNING

Viewing Information about Job Status and Failed Executions

The following query lists the job number, next execution time, failures, and broken status for each job you have submitted:

```
SVRMGR> SELECT job, next_date, next_sec, failures, broken
     2>     FROM user_jobs;

JOB        NEXT_DATE NEXT_SEC FAILURES   B
---------- --------- -------- ---------- -
      9125 01-NOV-94 00:00:00          4 N
     14144 24-OCT-94 16:35:35          0 N
     41762 01-JAN-00 00:00:00         16 Y
3 rows selected.
```

Viewing Information about Jobs Currently Running

The following query lists the session identifier, job number, user who submitted the job, and the start times for all currently running jobs:

```
SVRMGR> SELECT sid, r.job, log_user, r.this_date, r.this_sec
     2>     FROM dba_jobs_running r, dba_jobs j
     3>     WHERE r.job = j.job;

SID        JOB        LOG_USER             THIS_DATE THIS_SEC
---------- ---------- -------------------- --------- --------
        12      14144 VALERIE              24-OCT-94 17:21:24
        25       8536 SCOTT                24-OCT-94 16:45:12
2 rows selected.
```

CHAPTER

11

Using Deferred Transactions

This chapter describes the following topics:

- how to administer the deferred transaction queue
- how to create your own deferred transactions

Listing Information about Deferred Transactions

Oracle provides several tables and views for you to use in administering deferred transactions. These views provide information about each deferred transaction, such as the transaction destinations, the deferred calls that make up the transaction, and any errors encountered during attempted execution of the transaction. *You should not modify these tables directly; use the procedures provided in the DBMS_DEFER and DBMS_DEFER_SYS packages.*

These views are briefly described below. For more information, see Chapter 13.

DefCall	Records all deferred remote procedure calls.
DefCallDest	Lists the destinations for each deferred remote procedure call.
DefDefaultDest	Lists the default destination for deferred remote procedure calls.
DefError	Provides information about transactions that could not be applied at each destination.
DefSchedule	Displays information about when a job is next scheduled to be executed.
DefTran	Records all deferred transactions.
DefTranDest	Lists the destinations for a deferred transaction.

Creating a Deferred Transaction

Every well formed deferred transaction must consist of zero or one DBMS_DEFER.TRANSACTION calls followed by zero or more well formed deferred remote procedure calls, followed by a SQL COMMIT statement.

☞ **Attention:** The procedures for which you are building deferred calls must be part of a package. Deferred calls to standalone procedures are not supported.

Every well formed deferred remote procedure call must consist of one DBMS_DEFER.CALL call, followed by zero or more DBMS_DEFER.*datatype*_ARG calls. The number of calls to the appropriate *datatype*_ARG procedures is determined by the value of the ARG_COUNT parameter passed to the CALL procedure.

If you do not call DBMS_DEFER.TRANSACTION to indicate the start of a transaction, Oracle considers your first call to DBMS_DEFER.CALL to be the start of a new transaction.

Security

To create your own deferred transactions, you must have the EXECUTE privilege on the DBMS_DEFER package. This package is owned by SYS. Because deferred transactions may be executed in the privilege domain of a more privileged user at the remote site (such as the replication administrator), EXECUTE privileges on the DBMS_DEFER package should not be widely granted. To control access to these procedures, you may prefer to create a cover package, and grant EXECUTE on this cover package. For more information on executing deferred transactions, and the privilege domain used to execute the transaction, refer to page 4 – 26.

Specifying a Destination

In addition to building the calls that make up a deferred transaction, you must also specify the destination for this transaction. Transactions placed into the deferred transaction queue by the symmetric replication facility are queued to all of the asynchronous locations (dblinks) for the replicated object, as listed in the DBA_RepProp view. When you use the procedures in the DBMS_DEFER package to add a deferred transaction to the queue, you must specify a destination using one of the following methods. These methods are listed in order of precedence:

1. If you meet the following conditions, the DBA_RepProp view is used to determine destinations for the deferred transaction:

   - You do not use the NODES parameter to specify a destination in the call to DBMS_DEFER.TRANSACTION.
   - You do not use the NODES parameter to specify a destination for any calls to DBMS_DEFER.CALL.
   - Every call corresponds to a procedure in a $RP package generated for an object in DBA_RepObject.

   Note: This method cannot be combined with any of the following methods.

2. You specify one or more fully qualified database names as the NODES parameter to the DBMS_DEFER.CALL procedure. This value applies to the current deferred remote procedure call only.
3. You specify one or more fully qualified database names as the NODES parameter to the DBMS_DEFER.TRANSACTION procedure. This value applies to all deferred calls that make up the transaction.

4. If you do not use one of the previous mechanisms to specify a destination, Oracle uses the contents of the DefDefaultDest view described on page 13 – 14 to determine the destination for the calls.

Initiating a Deferred Transaction

Indicate the start of a new deferred transaction by calling the TRANSACTION procedure in the DBMS_DEFER package, as shown in the following example:

```
nodes dbms_defer.node_list_t;
node(1) := 'acct_hq.hq.com';
node(2) := 'acct_ny.ny.com';
DBMS_DEFER.TRANSACTION(nodes);
```

In this example, any calls that make up the deferred transaction for which you do not specify a destination when you call DBMS_DEFER.CALL, will be queued for the ACCT_HQ and ACCT_NY databases.

The call to TRANSACTION is optional. If you do not call TRANSACTION, Oracle considers your first call to DBMS_DEFER.CALL to be the start of a new transaction. Calling TRANSACTION is useful if you want to specify a list of nodes to which to forward the deferred calls, and the list is the same for all calls in the deferred transaction.

All deferred transactions are recorded in the DefTran view. Each destination of the transaction is noted in the DefTranDest view.

Additional Information: The parameters for the TRANSACTION procedure are described in Table 12 – 7, and the exceptions are listed in Table 12 – 8.

Deferring a Remote Procedure Call

To build a deferred call to a remote procedure, call the CALL procedure in the DBMS_DEFER package, as shown in the following example:

```
DBMS_DEFER.CALL(
    schema_name  => 'accts_rec',
    package_name => 'hr',
    proc_name    => 'hire_emp',
    arg_count    => 3);
```

This example builds a deferred call to the HR.HIRE_EMP procedure in the ACCTS_REC schema. This HIRE_EMP procedure takes three arguments. No destination is specified for the deferred call, so the destination must have been specified using one of the other methods outlined on page 11 – 3.

All deferred remote procedure calls are recorded in the DefCall view. If you specify a destination for the call, it is noted in the DefCallDest view.

Additional Information: The parameters for the CALL procedure are described in Table 12 – 1, and the exceptions are listed in Table 12 – 2.

Queuing a Parameter Value for a Deferred Call

After deferring a call to a remote procedure, you must provide the data that is passed to this procedure (only IN parameters are supported). There must be one call for each of the arguments that is passed to the remote procedure, and these calls must be made in the order that the arguments must be passed. The type of the data determines which procedure in the DBMS_DEFER package you must call. For example, suppose you deferred a call to the HIRE_EMP procedure, and it took three arguments, as shown below:

```
HIRE_EMP(ename IN VARCHAR2, empno IN NUMBER, salary IN NUMBER)
```

After building the deferred call to HIRE_EMP, you could pass the necessary data to this procedure by making the following three calls:

```
DBMS_DEFER.VARCHAR2_ARG('scott');
DBMS_DEFER.NUMBER_ARG(12345);
DBMS_DEFER.NUMBER_ARG(30000);
```

Depending upon the type of the data that you need to pass to the procedure, you need to call one of the following procedures in the DBMS_DEFER package for each argument to the procedure:

```
DBMS_DEFER.NUMBER_ARG(arg IN NUMBER);
DBMS_DEFER.DATE_ARG(arg IN DATE);
DBMS_DEFER.VARCHAR2_ARG(arg IN VARCHAR2);
DBMS_DEFER.CHAR_ARG(arg IN CHAR);
DBMS_DEFER.ROWID_ARG(arg IN ROWID);
DBMS_DEFER.RAW_ARG(arg IN RAW);
```

The RAW_ARG, CHAR_ARG, and VARCHAR2_ARG procedures can raise an ORA–23323 exception if the argument that you pass to the procedure is too long.

Adding a Destination to the DefDefaultDest View

If you use the DBMS_DEFER package to build a deferred transaction, and you do not supply a destination for a deferred transaction or the calls within that transaction, Oracle uses the DefDefaultDest view to determine the destination databases to which you want to defer a remote procedure call.

To add a destination database to this view call the ADD_DEFAULT_DEST procedure in the DBMS_DEFER_SYS package as shown in the following example:

```
DBMS_DEFER_SYS.ADD_DEFAULT_DEST( dblink => 'acct_ny.ny.com');
```

In this example, any *future* deferred transactions for which no destination has been specified will be queued for the ACCT_NY database. If you want to queue any *existing* deferred transactions to the new destination, you must call the COPY command for each of these transactions, as described in the next section.

> **Additional Information:** The parameter for the ADD_DEFAULT_DEST procedure is described in Table 12 – 14, and the exception is listed in Table 12 – 15.

Copying a Deferred Transaction to a New Destination

When you add a destination to the DefDefaultDest view, you may want to have certain outstanding deferred transactions applied at that location as well. To copy a deferred transaction from an existing destination site to a newly added destination, call the COPY procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
new_nodes dbms_defer.node_list_t;
new_nodes(1) := 'acct_ny.ny.com';
DBMS_DEFER_SYS.COPY(
    deferred_tran_id  => '234',
    deferred_tran_db  => 'acct_hq.hq.com',
    destination_list  => new_nodes,
    destination_count => 1);
```

In this example a copy of the transaction with ID number 234 is queued for the ACCT_NY site.

When you copy a deferred transaction, the new row in the DefTran view has different values for the DEFERRED_TRAN_ID, DEFERRED_TRAN_DB, and DELIVERY_ORDER columns than the source row. Additionally, the DESTINATION_LIST column is set to ‘D’. The destination list for the new transaction is determined by the final two arguments to the COPY procedure: DESTINATION_LIST and DESTINATION_COUNT. Finally, the copies of the DefCall records are assigned new CALLNO values.

> **Additional Information:** The parameters for the COPY procedure are described in Table 12 – 16.

Removing a Destination from the DefDefaultDest View

To remove a destination database from the DefDefaultDest view, call the DELETE_DEFAULT_DEST procedure in the DBMS_DEFER_SYS package, as shown in the following example:

```
DBMS_DEFER_SYS.DELETE_DEFAULT_DEST( dblink => 'acct_ny.ny.com');
```

In this example, any future deferred transactions that you create will no longer be queued for the ACCT_NY database as the default.

To delete a transaction from the deferred transaction queue, call the DELETE_TRAN procedure in the DBMS_DEFER_SYS package as described on page 7 – 5.

Additional Information: The parameter for the DELETE_DEFAULT_DEST procedure is described in Table 12 – 17.

Executing a Deferred Transaction

When you build a deferred transaction, the transaction is added to the deferred transaction queue at your local site. The remote procedures are not executed until this queue is pushed. You can either schedule this queue to be pushed at a periodic interval, by calling DBMS_DEFER_SYS.SCHEDULE_EXECUTION, or you can force the queue to be pushed immediately by calling DBMS_DEFER_SYS.EXECUTE. These transactions are propagated in the same manner as your DML changes are propagated by the symmetric replication facility. For more information on propagating the deferred transaction queue, refer to page 4 – 26.

CHAPTER

12

Reference

This chapter describes the parameters for the packaged procedures that are supplied with the symmetric replication facility. Any possible exceptions raised by these procedures are also described. The procedures are listed alphabetically within each package. The package variables used with replication are described at the end of this chapter on page 12 – 137.

DBMS_DEFER.CALL

Purpose

To build a deferred call to a remote procedure. For additional information, refer to page 11 – 4.

Syntax

The parameters for the CALL procedure are described in Table 12 – 1, and the exceptions are listed in Table 12 – 2. The syntax for this procedure is shown below:

```
DBMS_DEFER.CALL( schema_name   IN     VARCHAR2,
                 package_name  IN     VARCHAR2,
                 proc_name     IN     VARCHAR2,
                 arg_count     IN     NATURAL
                 [, nodes      IN     node_list_t]
                 [, group_name IN     VARCHAR2 :=’’])
```

Parameter	Description
`schema_name`	The name of the schema in which the stored procedure is located.
`package_name`	The name of the package containing the stored procedure. The stored procedure must be part of a package. Deferred calls to standalone procedures are not supported.
`proc_name`	The name of the remote procedure to which you want to defer a call.
`arg_count`	The number of parameters for the procedure. You must have one call to DBMS_DEFER.*datatype*_ARG for each of these parameters.
`nodes`	A PL/SQL table of fully qualified database names to which you want to propagate the deferred call. The table is indexed starting at position one and ending when a NULL entry is found, or the NO_DATA_FOUND exception is raised. The data in the table is case insensitive. This argument is optional.
`group_name`	Reserved for internal use.

Table 12 – 1 Parameters for CALL

Exception	Description
ORA–23304 (malformedcall)	The previous call was not correctly formed.
ORA–23319	Parameter value is not appropriate.
ORA–23352	The destination list (specified by NODES or by a previous DBMS_DEFER.TRANSACTION call) contains duplicates.

Table 12 – 2 Exceptions for CALL

DBMS_DEFER.COMMIT_WORK

Purpose

To perform a transaction commit after checking for well formed deferred remote procedure calls.

Syntax

The parameter for the COMMIT_WORK procedure is described in Table 12 – 3, and the exception is listed in Table 12 – 4. The syntax for this procedure is as follows:

```
DBMS_DEFER.COMMIT_WORK(commit_work_comment IN VARCHAR2)
```

Parameter	Description
`commit_work_ comment`	Up to 50 bytes to describe the transaction in the DEF$_CALL table.

Table 12 – 3 Parameter for COMMIT_WORK

Exception	Description
ORA–23304 (malformedcall)	The transaction was not correctly formed or terminated.

Table 12 – 4 Exception for COMMIT_WORK

DBMS_DEFER.*datatype*_ARG

Purpose

To provide the data that is to be passed to a deferred remote procedure call. For additional information, refer to page 11 – 5.

Syntax

Depending upon the type of the data that you need to pass to the procedure, you need to call one of the following procedures in the DBMS_DEFER package for each argument to the procedure:

```
DBMS_DEFER.NUMBER_ARG(arg IN NUMBER);

DBMS_DEFER.DATE_ARG(arg IN DATE);

DBMS_DEFER.VARCHAR2_ARG(arg IN VARCHAR2);

DBMS_DEFER.CHAR_ARG(arg IN CHAR);

DBMS_DEFER.ROWID_ARG(arg IN ROWID);

DBMS_DEFER.RAW_ARG(arg IN RAW);
```

Parameter	Description
`arg`	The value of the parameter that you want to pass to the remote procedure to which you previously deferred a call.

Table 12 – 5 Parameter for NUMBER/DATE/VARCHAR2/CHAR/ROWID/ RAW_ARG

Exception	Description
ORA–23323	The argument value is too long.

Table 12 – 6 Exception for VARCHAR2/CHAR/RAW_ARG

DBMS_DEFER.TRANSACTION

Purpose

To indicate the start of a new deferred transaction. If you omit this call, Oracle considers your first call to DBMS_DEFER.CALL to be the start of a new transaction. For additional information, refer to page 11 – 4.

Syntax

The parameter for the TRANSACTION procedure is described in Table 12 – 7, and the exceptions are listed in Table 12 – 8. The syntax for this procedure is as follows:

```
DBMS_DEFER.TRANSACTION

DBMS_DEFER.TRANSACTION( nodes         IN     node_list_t)
```

Note: The transaction procedure is overloaded. The behavior of the version without a parameter is similar to that of the version with a parameter, except that the former uses the nodes in the DefDefaultDest view instead of having a nodes parameter.

Parameter	Description
`nodes`	A PL/SQL table of fully qualified database names to which you want to propagate the deferred calls of the transaction. The table is indexed starting at position one until a NULL entry is found, or the NO_DATA_FOUND exception is raised. The data in the table is case insensitive.

Table 12 – 7 Parameter for TRANSACTION

Exception	Description
ORA–23304 (malformedcall)	The previous transaction was not correctly formed or terminated.
ORA–23319	Parameter value is not appropriate.
ORA–23352	Raised by DBMS_DEFER.CALL if the node list contains duplicates.

Table 12 – 8 Exceptions for TRANSACTION

DBMS_DEFER_QUERY.GET_ARG_TYPE

Purpose

To determine the type of an argument in a deferred call. For additional information, refer to page 7 – 2.

Syntax

The parameters for the GET_ARG_TYPE function are described in Table 12 – 9, the exception is listed in Table 12 – 10, and the possible return values are described in Table 12 – 11. The syntax for this procedure is shown below:

```
DBMS_DEFER_QUERY.GET_ARG_TYPE(
          callno             IN     NUMBER,
          deferred_tran_db   IN     VARCHAR2,
          arg_no             IN     NUMBER,
          deferred_tran_id   IN     VARCHAR2)
       RETURN NUMBER
```

Parameter	Description
callno	The ID number from the DefCall view of the deferred remote procedure call.
deferred_tran_db	Fully qualified database name that originated the transaction. This is also stored in the DefCall view.
arg_no	The numerical position of the argument to the call whose type you want to determine. The first argument to a procedure is in position one.
deferred_tran_id	The identifier of the deferred transaction.

Table 12 – 9 Parameters for GET_ARG_TYPE

Exception	Description
NO_DATA_FOUND	The input parameters do not correspond to a parameter of a deferred call.

Table 12 – 10 Exception for GET_ARG_TYPE

Return Value	Corresponding Datatype
1	VARCHAR2
2	NUMBER
11	ROWID
12	DATE
23	RAW
96	CHAR

Table 12 – 11 Return Values for GET_ARG_TYPE

DBMS_DEFER_QUERY.GET_*datatype*_ARG

Purpose

To determine the value of an argument in a deferred call. For additional information, refer to page 7 – 3.

Syntax

Depending upon the type of the argument value that you want to retrieve, the syntax for the appropriate function is as follows. The parameters for these functions are described in Table 12 – 12, and the exceptions are listed in Table 12 – 13. Each of these functions returns the value of the specified argument.

For arguments of type NUMBER:

```
DBMS_DEFER_QUERY.GET_NUMBER_ARG
       (callno             IN       NUMBER,
       deferred_tran_db    IN       VARCHAR2,
       arg_no              IN       NUMBER,
       deferred_tran_id    IN       VARCHAR2 DEFAULT NULL)
       RETURN NUMBER
```

For arguments of type VARCHAR2:

```
DBMS_DEFER_QUERY.GET_VARCHAR2_ARG
       (callno             IN       NUMBER,
       deferred_tran_db    IN       VARCHAR2,
       arg_no              IN       NUMBER,
       deferred_tran_id    IN       VARCHAR2 DEFAULT NULL)
       RETURN VARCHAR2
```

For arguments of type CHAR:

```
DBMS_DEFER_QUERY.GET_CHAR_ARG
       (callno             IN       NUMBER,
       deferred_tran_db    IN       VARCHAR2,
       arg_no              IN       NUMBER,
       deferred_tran_id    IN       VARCHAR2 DEFAULT NULL)
       RETURN CHAR
```

For arguments of type DATE:

```
DBMS_DEFER_QUERY.GET_DATE_ARG
       (callno             IN       NUMBER,
       deferred_tran_db    IN       VARCHAR2,
       arg_no              IN       NUMBER,
       deferred_tran_id    IN       VARCHAR2 DEFAULT NULL)
       RETURN DATE
```

For arguments of type RAW:

```
DBMS_DEFER_QUERY.GET_RAW_ARG
       (callno             IN       NUMBER,
       deferred_tran_db    IN       VARCHAR2,
       arg_no              IN       NUMBER,
       deferred_tran_id    IN        VARCHAR2 DEFAULT NULL)
       RETURN RAW
```

For arguments of type ROWID:

```
DBMS_DEFER_QUERY.GET_ROWID_ARG
        (callno             IN        NUMBER,
        deferred_tran_db    IN        VARCHAR2,
        arg_no              IN        NUMBER,
        deferred_tran_id    IN        VARCHAR2 DEFAULT NULL)
        RETURN ROWID
```

Parameter	Description
callno	The ID number from the DefCall view of the deferred remote procedure call.
deferred_tran_db	Fully qualified database name that originated the transaction. This is also stored in the DefCall view.
arg_no	The numerical position of the argument to the call whose value you want to determine. The first argument to a procedure is in position one.
deferred_tran_id	Default NULL. The identifier of the deferred transaction. Defaults to the last transaction identifier passed to GET_ARG_TYPE.

Table 12 – 12 Parameters for GET_NUMBER/VARCHAR2/CHAR/DATE/RAW/ROWID_ARG

Exception	Description
NO_DATA_FOUND	The input parameters do not correspond to a parameter of a deferred call.
bad_param_type	The argument in this position is not of the specified type.

Table 12 – 13 Exceptions for GET_NUMBER/VARCHAR2/CHAR/DATE/RAW/ROWID_ARG

DBMS_DEFER_SYS.ADD_DEFAULT_DEST

Purpose

To add a destination database to the DefDefaultDest view. For additional information, refer to page 11 – 5.

Syntax

The parameter for the ADD_DEFAULT_DEST procedure is described in Table 12 – 14, and the exception is listed in Table 12 – 15. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.ADD_DEFAULT_DEST(  dblink    IN     VARCHAR2)
```

Parameter	Description
`dblink`	The fully qualified database name of the node that you want to add to the DefDefaultDest view.

Table 12 – 14 Parameter for ADD_DEFAULT_DEST

Exception	Description
ORA–23352	The DBLINK that you specified is already in the default list.

Table 12 – 15 Exception for ADD_DEFAULT_DEST

DBMS_DEFER_SYS.COPY

Purpose

To create a copy of a deferred transaction with a new destination. For additional information, refer to page 11 – 6.

Syntax

The parameters for the COPY procedure are described in Table 12 – 16. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.COPY
 (deferred_tran_id    IN         VARCHAR2,
 deferred_tran_db     IN         VARCHAR2,
 destination_list     IN         dbms_defer.node_list_t,
 destination_count    IN         BINARY_INTEGER)
```

Parameter	Description
`deferred_tran_id`	The ID number from the DefTran view of the deferred transaction that you want to copy.
`deferred_tran_db`	The fully qualified database name from the DefTran view of the database in which the transaction that you want to copy originated.
`destination_list`	A PL/SQL table of fully qualified database names to which you want to propagate the deferred calls of the copied transaction. The table is indexed starting at position one and the data in the table is case insensitive.
`destination_count`	The number of entries in the DESTINATION_LIST table.

Table 12 – 16 Parameters for COPY

DBMS_DEFER_SYS.DELETE_DEFAULT_DEST

Purpose

To remove a destination database from the DefDefaultDest view. For additional information, refer to page 11 – 6.

Syntax

The parameter for the DELETE_DEFAULT_DEST procedure is described in Table 12 – 17. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.DELETE_DEFAULT_DEST(dblink IN VARCHAR2);
```

Parameter	Description
`dblink`	The fully qualified database name of the node that you want to delete from the DefDefaultDest view. If Oracle does not find this dblink in the view, no action is taken.

Table 12 – 17 Parameter for DELETE_DEFAULT_DEST

DBMS_DEFER_SYS.DELETE_ERROR

Purpose

To delete a transaction from the DefError view. If there are not other DefTranDest or DefError entries for the transaction, the transaction is deleted from the DefTran and DefCall views as well. For additional information, refer to page 7 – 5.

Syntax

The parameters for the DELETE_ERROR procedure are described in Table 12 – 18. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.DELETE_ERROR(deferred_tran_id  IN  VARCHAR2,
                            deferred_tran_db  IN  VARCHAR2,
                            destination       IN  VARCHAR2)
```

Parameter	Description
`deferred_tran_id`	The ID number from the DefError view of the deferred transaction that you want to remove from the DefError view. If this parameter is null, all transactions meeting the requirements of the other parameters are removed.
`deferred_tran_db`	The fully qualified database name from the DefError view of the database in which the transaction that you want to remove from the DefError view originated. If this parameter is null, transactions meeting the requirements of the other parameters are removed.
`destination`	The fully qualified database name from the DefError view of the database to which the transaction was originally queued. If this parameter is null, all transactions meeting the requirements of the other parameters are removed from the DefError view.

Table 12 – 18 Parameters for DELETE_ERROR

Exception

Transaction ID and / or node not found.

DBMS_DEFER_SYS.DELETE_TRAN

Purpose

To delete a transaction from the DefTranDest view. If there are not other DefTranDest or DefError entries for the transaction, the transaction is deleted from the DefTran and DefCall views as well. For additional information, refer to page 7 – 5.

Syntax

The parameters for the DELETE_TRAN procedure are described in Table 12 – 19. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.DELETE_TRAN(deferred_tran_id IN     VARCHAR2,
                           deferred_tran_db IN     VARCHAR2,
                           destination      IN     VARCHAR2)
```

Parameter	Description
`deferred_tran_id`	The ID number from the DefTran view of the deferred transaction that you want to delete. If this parameter is null, all transactions meeting the requirements of the other parameters are deleted.
`deferred_tran_db`	The fully qualified database name from the DefTran view of the database in which the transaction that you want to delete originated. If this parameter is null, transactions meeting the requirements of the other parameters are deleted.
`destination`	The fully qualified database name from the DefTranDest view of the database to which the transaction was originally queued. If this parameter is null, all transactions meeting the requirements of the other parameters are deleted.

Table 12 – 19 Parameters for DELETE_TRAN

Exception

Transaction ID and / or node not found.

DBMS_DEFER_SYS.DISABLED

Purpose

To determine if propagation of the deferred transaction queue from the current site to a given site is enabled. The DISABLED function returns TRUE if the deferred remote procedure call (RPC) queue is disabled for the given destination. For additional information, refer to page 4 – 32.

Syntax

The parameter for the DISABLED function is described in Table 12 – 20, the return values are described in Table 12 – 21, and the exception is listed in Table 12 – 22. The syntax for this function is shown below:

```
DBMS_DEFER_SYS.DISABLED(destination  IN  VARCHAR2)
     RETURN BOOLEAN
```

Parameter	Description
destination	The fully qualified database name of the node whose propagation status you want to check.

Table 12 – 20 Parameter for DISABLED

Value	Description
TRUE	Propagation to this site from the current site is disabled.
FALSE	Propagation to this site from the current site is enabled.

Table 12 – 21 Return Values for DISABLED

Exception	Description
NO_DATA_FOUND	DESTINATION does not appear in the DefSchedule view.

Table 12 – 22 Exception for DISABLED

DBMS_DEFER_SYS.EXECUTE

Purpose

To force a deferred remote procedure call queue at your current master or snapshot site to be pushed to another master site. For additional information, refer to page 4 – 31.

Syntax

The parameters for the EXECUTE procedure are shown in Table 12 – 23. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.EXECUTE(
          destination         IN     VARCHAR2,
          stop_on_error       IN     BOOLEAN := FALSE,
          transaction_count   IN     BINARY_INTEGER := 0,
          execution_seconds   IN     BINARY_INTEGER := 0,
          execute_as_user     IN     BOOLEAN := FALSE,
          delay_seconds       IN     NATURAL := 0,
          batch_size          IN     NATURAL := 0)
```

Parameter	Description
`destination`	The fully qualified database name of the master site to which you are forwarding changes.
`stop_on_error`	The default, FALSE, indicates that execution should continue even if errors, such as conflicts are encountered. Set this to TRUE if you want to stop execution when the first error is encountered.
`transaction_count`	Execution stops after TRANSACTION_COUNT (number of transactions) *or* EXECUTION_SECONDS (number of seconds) has occurred. By default, (TRANSACTION_COUNT = 0 and EXECUTION_SECONDS = 0) transactions are executed until there are no more in the queue.
`execution_seconds`	
`execute_as_user`	The default, FALSE, indicates that a deferred call is authenticated at the remote system using the authentication context of the user who originally queued the deferred call (as indicated in the ORIGIN_USER column of the DefTran view). Set this to TRUE if you want the execution of a deferred call to be authenticated at the remote system using the authentication context of the session user.

Table 12 – 23 Parameters for EXECUTE, continued on next page

Parameter	Description
`delay_seconds`	The routine will sleep for this many seconds (default is 0), before returning when it finds no deferred calls queued for the destination. A non–zero value can reduce execution overhead compared to calling DBMS_DEFER_SYS.EXECUTE from a tight loop.
`batch_size`	Indicates that a COMMIT should occur when the total number of deferred calls executed exceeds this number, and a complete transaction has been executed. If this argument is 0 (the default), a COMMIT occurs after each deferred transaction. For transactions originating at a master site, bundling transactions in this manner does not affect how errors are resolved, or how errors are logged in the DefError view. For snapshot sites in a multi–master environment, you should always set this parameter to 0. Otherwise, the transactions are bundled before they are forwarded to other master sites, making conflict resolution more difficult.

Table 12 – 23 Parameters for EXECUTE

DBMS_DEFER_SYS.EXECUTE_ERROR

Purpose

To re–execute a deferred transaction that did not initially complete successfully. This procedure raises an ORA–24275 error when illegal combinations of NULL and non–NULL parameters are used. For additional information, refer to page 7 – 4.

Syntax

The parameters for the EXECUTE_ERROR procedure are described in Table 12 – 24. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.EXECUTE_ERROR(deferred_tran_id IN  VARCHAR2,
                             deferred_tran_db IN  VARCHAR2,
                             destination      IN  VARCHAR2)
```

Parameter	Description
`deferred_tran_id`	The ID number from the DefError view of the deferred transaction that you want to re–execute. If this parameter is null, all transactions queued for DESTINATION that originated from the DEFERRED_TRAN_DB are re–executed.
`deferred_tran_db`	The fully qualified database name from the DefError view of the database in which the transaction that you want to re–execute originated. If both the DEFERRED_TRAN_ID and DEFERRED_TRAN_DB are null, all transactions originating from any site that were queued for the DESTINATION database that were not successfully executed are re–executed.
`destination`	The fully qualified database name from the DefError view of the database to which the transaction was originally queued. This parameter must not be null.

Table 12 – 24 Parameters for EXECUTE_ERROR

Exception

EXECUTE_ERROR raises the last exception that it encountered before execution stopped as a result of the exception.

DBMS_DEFER_SYS.SCHEDULE_EXECUTION

Purpose

To establish communication between a master or snapshot site and another master site. For additional information, refer to page 4 – 30.

Syntax

The parameters for the SCHEDULE_EXECUTION procedure are described in Table 12 – 25. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.SCHEDULE_EXECUTION(
               dblink            IN  VARCHAR2,
               interval          IN  VARCHAR2,
               next_date         IN  DATE,
               reset             IN  BOOLEAN default FALSE,
               stop_on_error     IN  BOOLEAN := NULL,
               transaction_count IN  BINARY_INTEGER := NULL,
               execution_seconds IN  BINARY_INTEGER := NULL,
               execute_as_user   IN  BOOLEAN := NULL,
               delay_seconds     IN  NATURAL := NULL,
               batch_size        IN  NATURAL := NULL)
```

Parameter	Description
dblink	Fully qualified pathname to master database site at which you want to schedule periodic execution of deferred remote procedure calls.
interval	Allows you to provide a function that is used to calculate the next time to apply any changes. This value is stored in the INTERVAL field of the DefSchedule view and is used to calculate the NEXT_DATE field of this view. If you use the default value for this parameter, NULL, the value of this field remains unchanged. If the field had no previous value, it is created with a value of null. If you do not supply a value for this field, you must supply a value for NEXT_DATE.
next_date	Allows you to specify a given time to apply any outstanding changes to the given master site. This value is stored in the NEXT_DATE field of the DefSchedule view. If you use the default value for this parameter, NULL, the value of this field remains unchanged. If this field had no previous value, it is created with a value of null. If you do not supply a value for this field, you must supply a value for INTERVAL. Scheduling a communication interval between master sites is very similar to scheduling a refresh interval for snapshots, as described on page 3 – 16.

Table 12 – 25 Parameters for SCHEDULE_EXECUTION
continued on next page

Parameter	Description
`reset`	Set to TRUE to reset LAST_TXN_COUNT, LST_ERROR, and LAST_MSG to NULL.
`stop_on_error` `transaction_count` `execution_seconds` `execute_as_user` `delay_seconds` `batch_size`	These parameters are passed to the DBMS_DEFER_SYS.EXECUTE call that is scheduled for execution by this call. See Table 12 – 23 for more information on how these parameters are used by EXECUTE.

Table 12 – 25 Parameters for SCHEDULE_EXECUTION

DBMS_DEFER_SYS.SET_DISABLED

Purpose

To disable or enable propagation of the deferred transaction queue from the current site to a given destination site. If the disabled parameter is TRUE, the procedure disables propagation to the given destination and future invocations of DBMS_DEFER_SYS.EXECUTE do not push the deferred remote procedure call (RPC) queue. SET_DISABLED affects neither a session already pushing the queue to the given destination nor sessions appending to the queue with DBMS_DEFER. If the disable parameter is FALSE, the procedure enables propagation to the given destination and, although this does not push the queue, it permits future invocations to DBMS_DEFER_SYS.EXECUTE to push the queue to the given destination. Whether the disabled parameter is TRUE or FALSE, a COMMIT is required for the setting to take effect in other sessions.

Syntax

The parameters for the SET_DISABLED procedure are described in Table 12 – 26 and the exception is listed in Table 12 – 27. The syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.SET_DISABLED(destination IN  VARCHAR2,
                            disabled    IN  BOOLEAN := TRUE)
```

Parameter	Description
`destination`	The fully qualified database name of the node whose propagation status you want to change.
`disabled`	By default, this parameter disables propagation of the deferred transaction queue from your current site to the given destination. Set this parameter to FALSE to enable propagation.

Table 12 – 26 Parameters for SET_DISABLED

Exception	Description
NO_DATA_FOUND	No entry was found in the DefSchedule view for the given DESTINATION.

Table 12 – 27 Exception for SET_DISABLED

DBMS_DEFER_SYS.UNSCHEDULE_EXECUTION

Purpose

To stop automatic communication from a snapshot or master site to another master site. For additional information, refer to page 4 – 32.

Syntax

The parameter for the UNSCHEDULE_EXECUTION procedure is described in Table 12 – 28, and the exception is listed in Table 12 – 29. The complete syntax for this procedure is shown below:

```
DBMS_DEFER_SYS.UNSCHEDULE_EXECUTION(
                    dblink IN     VARCHAR2 NOT NULL)
```

Parameter	Description
dblink	Fully qualified pathname to master database site at which you want to unschedule periodic execution of deferred remote procedure calls.

Table 12 – 28 Parameter for UNSCHEDULE_EXECUTION

Exception	Description
NO_DATA_FOUND	No entry was found in the DefSchedule view for the given DBLINK.

Table 12 – 29 Exception for UNSCHEDULE_EXECUTION

DBMS_JOB.BROKEN

Purpose

To mark a job as broken or not broken. You can mark only jobs you own as broken. For additional information, refer to page 10 – 10.

Syntax

The syntax for the procedure DBMS_JOB.BROKEN is shown below. Table 12 – 30 describes the procedure's parameters.

```
DBMS_JOB.BROKEN(job            IN     BINARY_INTEGER,
                broken         IN     BOOLEAN,
                next_date      IN     DATE DEFAULT SYSDATE)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it.
`broken`	Indicates the state of the job. TRUE indicates that the job is broken and should not be executed. FALSE indicates that the job is not broken and should be executed at its normal execution interval, starting with NEXT_DATE.
`next_date`	The date at which the job will be run next. The default value is SYSDATE.

Table 12 – 30 Parameters for DBMS_JOB.BROKEN

DBMS_JOB.CHANGE

Purpose

To alter any of the user–definable parameters associated with a job that has been submitted to the job queue. For additional information, refer to page 10 – 9.

☞ **Attention:** When you change a job's definition using the WHAT parameter in the procedure CHANGE, Oracle records your current environment. This becomes the new environment for the job.

Syntax

The syntax for the procedure DBMS_JOB.CHANGE is shown below. Table 12 – 31 describes the procedure's parameters.

```
DBMS_JOB.CHANGE(job            IN     BINARY_INTEGER,
                what           IN     VARCHAR2,
                next_date      IN     DATE,
                interval       IN     VARCHAR2)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it.
`what`	The PL/SQL code you want to have executed. Specify NULL to leave the current value unchanged. For more information about defining a job, see "Job Definitions" on page 10 – 6.
`next_date`	The next date when the job will be run. Specify NULL to leave the current value unchanged.
`interval`	The date function that calculates the next time to execute the job. INTERVAL must evaluate to a future point in time or NULL. Specify NULL to leave the current value unchanged. For more information on how to specify an execution interval, see page 10 – 7.

Table 12 – 31 Parameters for DBMS_JOB.CHANGE

DBMS_JOB.INTERVAL

Purpose

To alter the execution interval of a job. For additional information, refer to page 10 – 9.

Syntax

The syntax for the procedure DBMS_JOB.INTERVAL is shown below. Table 12 – 32 describes the procedure's parameters.

```
DBMS_JOB.INTERVAL(job         IN     BINARY_INTEGER,
                  interval    IN     VARCHAR2)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it.
`interval`	The date function that calculates the next time to execute the job. INTERVAL must evaluate to a future point in time or NULL. For more information on how to specify an execution interval, see page 10 – 7.

Table 12 – 32 Parameters for DBMS_JOB.INTERVAL

DBMS_JOB.NEXT_DATE

Purpose

To alter the date that a job will be next executed by Oracle. For additional information, refer to page 10 – 9.

Syntax

The syntax for the procedure DBMS_JOB.NEXT_DATE is shown below. Table 12 – 33 describes the procedure's parameters.

```
DBMS_JOB.NEXT_DATE(job         IN     BINARY_INTEGER,
                   next_date   IN     DATE)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it.
`next_date`	The next date when the job will be run.

Table 12 – 33 Parameters for DBMS_JOB.NEXT_DATE

DBMS_JOB.REMOVE

Purpose

To remove a job from the job queue. For additional information, refer to page 10 – 9.

Syntax

The syntax for the procedure DBMS_JOB.REMOVE is shown below. Table 12 – 34 describes the procedure's parameter.

```
DBMS_JOB.REMOVE(job IN BINARY_INTEGER)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it.

Table 12 – 34 Parameter for DBMS_JOB.REMOVE

DBMS_JOB.RUN

Purpose

To force a job to be executed immediately, even if the job is marked as broken. You can run only jobs that you own. For additional information, refer to page 10 – 11.

Syntax

The syntax for the procedure DBMS_JOB.RUN is shown below. Table 12 – 35 describes the procedure's parameter.

```
DBMS_JOB.RUN( job        IN     BINARY_INTEGER)
```

Parameter	Description
`job`	The identifier that was assigned to the job when you created it. When the job is run, Oracle recomputes the next execution date.

Table 12 – 35 Parameter for DBMS_JOB.RUN

DBMS_JOB.SUBMIT

Purpose

To submit a new job to the job queue. For additional information, refer to page 10 – 5.

Syntax

The syntax for the procedure DBMS_JOB.SUBMIT is shown below. Table 12 – 36 describes the procedure's parameters.

```
DBMS_JOB.SUBMIT( job          OUT    BINARY_INTEGER,
                 what         IN     VARCHAR2,
                 next_date    IN     DATE DEFAULT SYSDATE,
                 interval     IN     VARCHAR2 DEFAULT 'null',
                 no_parse     IN     BOOLEAN DEFAULT FALSE)
```

Parameter	Description
`job` (out parameter)	The identifier assigned to the job you created. You must use the job number whenever you want to alter or remove the job. For more information about job numbers, see "Job Numbers" on page 10 – 6.
`what`	is the PL/SQL code you want to have executed. In the job definition, use two single quotation marks around strings. Always include a semicolon at the end of the job definition. For more information about defining a job, see "Job Definitions" on page 10 – 6.
`next_date`	is the next date when the job will be run. The default value is SYSDATE.
`interval`	is the date function that calculates the next time to execute the job. The default value is NULL. INTERVAL must evaluate to a future point in time or NULL. For more information on how to specify an execution interval, see page 10 – 7.
`no_parse`	is a flag. The default value is FALSE. If NO_PARSE is set to FALSE (the default), Oracle parses the procedure associated with the job. If NO_PARSE is set to TRUE, Oracle parses the procedure associated with the job the first time that the job is executed. If, for example, you want to submit a job before you have created the tables associated with the job, set NO_PARSE to TRUE.

Table 12 – 36 Parameters for DBMS_JOB.SUBMIT

DBMS_JOB.WHAT

Purpose

To alter the definition of a job in the job queue. For additional information, refer to page 10 – 9.

☞ **Attention:** When you execute procedure WHAT, Oracle records your current environment. This becomes the new environment for the job.

Syntax

The syntax for the procedure DBMS_JOB.WHAT is shown below. Table 12 – 37 describes the procedure's parameters.

```
DBMS_JOB.WHAT(job             IN     BINARY_INTEGER,
              what            IN     VARCHAR2)
```

Parameter	Description
job	The identifier that was assigned to the job when you created it.
what	The PL/SQL code you want to have executed. In the job definition, use two single quotation marks around strings. Always include a semicolon at the end of the job definition.

Table 12 – 37 Parameters for DBMS_JOB.WHAT

DBMS_OFFLINE_OG.BEGIN_INSTANTIATION

Purpose

To start offline instantiation of a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 4 – 19.

Syntax

The parameters for the BEGIN_INSTANTIATION procedure are described in Table 12 – 38, and the exceptions are listed in Table 12 – 39. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_OG.BEGIN_INSTANTIATION( gname     IN  VARCHAR2,
                                     new_site  IN  VARCHAR2)
```

Parameter	Description
`gname`	The name of the object group that you want to replicate to the new site.
`new_site`	The fully qualified database name of the new site to which you want to replicate the object group.

Table 12 – 38 Parameters for BEGIN_INSTANTIATION

Exception	Description
23430 badargument	Null or empty string for object group or new master site name.
nonmasterdef	This procedure must be called from the master definition site.
23432 sitealreadyexists	Given site is already a master site for this object group.
23431 wrongstate	Status of master definition site must be QUIESCED.
missingrepgroup	GNAME does not exist as a replicated object group.

Table 12 – 39 Exceptions for BEGIN_INSTANTIATION

DBMS_OFFLINE_OG.BEGIN_LOAD

Purpose

To disable triggers while data is imported to new master site as part of offline instantiation. You must call this procedure from the new master site. For additional information, refer to page 4 – 20.

Syntax

The parameters for the BEGIN_LOAD procedure are described in Table 12 – 40, and the exceptions are listed in Table 12 – 41. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_OG.BEGIN_LOAD( gname     IN  VARCHAR2,
                            new_site  IN  VARCHAR2)
```

Parameter	Description
gname	The name of the object group whose members you are importing.
new_site	The fully qualified database name of the new site at which you will be importing the object group members.

Table 12 – 40 Parameters for BEGIN_LOAD

Exception	Description
23430 badargument	Null or empty string for object group or new master site name.
wrongsite	This procedure must be called from the new master site.
23434 unknownsite	Given site is not recognized by object group.
23431 wrongstate	Status of the new master site must be QUIESCED.
missingrepgroup	GNAME does not exist as a replicated object group.

Table 12 – 41 Exceptions for BEGIN_LOAD

DBMS_OFFLINE_OG.END_INSTANTIATION

Purpose

To complete offline instantiation of a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 4 – 20.

Syntax

The parameters for the END_INSTANTIATION procedure are described in Table 12 – 42, and the exceptions are listed in Table 12 – 43. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_OG.BEGIN_INSTANTIATION( gname     IN  VARCHAR2,
                                     new_site  IN  VARCHAR2)
```

Parameter	Description
gname	The name of the object group that you are replicating to the new site.
new_site	The fully qualified database name of the new site to which you are replicating the object group.

Table 12 – 42 Parameters for END_INSTANTIATION

Exception	Description
23430 badargument	Null or empty string for object group or new master site name.
nonmasterdef	This procedure must be called from the master definition site.
23434 unknownsite	Given site is not recognized by object group.
23431 wrongstate	Status of master definition site must be QUIESCED.
missingrepgroup	GNAME does not exist as a replicated object group.

Table 12 – 43 Exceptions for END_INSTANTIATION

DBMS_OFFLINE_OG.END_LOAD

Purpose

To re–enable triggers after importing data to new master site as part of offline instantiation. You must call this procedure from the new master site. For additional information, refer to page 4 – 20.

Syntax

The parameters for the END_LOAD procedure are described in Table 12 – 44, and the exceptions are listed in Table 12 – 45. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_OG.END_LOAD( gname     IN  VARCHAR2,
                          new_site  IN  VARCHAR2)
```

Parameter	Description
gname	The name of the object group whose members you have finished importing.
new_site	The fully qualified database name of the new site at which you have imported the object group members.

Table 12 – 44 Parameters for END_LOAD

Exception	Description
23430 badargument	Null or empty string for object group or new master site name.
wrongsite	This procedure must be called from the new master site.
23434 unknownsite	Given site is not recognized by object group.
23431 wrongstate	Status of the new master site must be QUIESCED.
missingrepgroup	GNAME does not exist as a replicated object group.

Table 12 – 45 Exceptions for END_LOAD

DBMS_OFFLINE_OG.RESUME_SUBSET_OF_MASTERS

Purpose

To resume replication activity at all existing sites except the new site during offline instantiation of a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 4 – 19.

Syntax

The parameters for the RESUME_SUBSET_OF_MASTERS procedure are described in Table 12 – 46, and the exceptions are listed in Table 12 – 47. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_OG.RESUME_SUBSET_OF_MASTERS(
     gname     IN  VARCHAR2,
     new_site  IN  VARCHAR2)
```

Parameter	Description
`gname`	The name of the object group that you are replicating to the new site.
`new_site`	The fully qualified database name of the new site to which you are replicating the object group.

Table 12 – 46 Parameters for RESUME_SUBSET_OF_MASTERS

Exception	Description
23430 badargument	Null or empty string for object group or new master site name.
nonmasterdef	This procedure must be called from the master definition site.
23434 unknownsite	Given site is not recognized by object group.
23431 wrongstate	Status of master definition site must be QUIESCED.
missingrepgroup	GNAME does not exist as a replicated object group.

Table 12 – 47 Exceptions for RESUME_SUBSET_OF_MASTERS

DBMS_OFFLINE_SNAPSHOT.BEGIN_LOAD

Purpose

To prepare a snapshot site for import of a new snapshot as part of offline instantiation. You must call this procedure from the snapshot site for the new snapshot. For additional information, refer to page 5 – 11.

Syntax

The parameters for the BEGIN_LOAD procedure are described in Table 12 – 48, and the exceptions are listed in Table 12 – 49. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_SNAPSHOT.BEGIN_LOAD(
     gname           IN  VARCHAR2,
     sname           IN  VARCHAR2,
     master_site     IN  VARCHAR2,
     snapshot_oname  IN  VARCHAR2,
     storage_c       IN  VARCHAR2 := '',
     comment         IN  VARCHAR2 :='')
```

Parameter	Description
`gname`	The name of the object group for the snapshot that you are creating using offline instantiation.
`sname`	The name of the schema for the new snapshot.
`master_site`	The fully qualified database name of the snapshot's master site.
`snapshot_oname`	The name of the temporary snapshot created at the master site.
`storage_c`	The storage options to use when creating the new snapshot at the snapshot site.
`comment`	User comment.

Table 12 – 48 Parameters for BEGIN_LOAD

Exception	Description
23430 badargument	Null or empty string for object group, schema, master site, or snapshot name.
missingrepgroup	GNAME does not exist as a replicated object group.
missingremotesnap	Could not locate given snapshot at given master site.
snaptabmismatch	Base table name at snapshot site and master site must be identical.
missingschema	The given schema does not exist.

Table 12 – 49 Exceptions for BEGIN_LOAD

DBMS_OFFLINE_SNAPSHOT.END_LOAD

Purpose To complete offline instantiation of a snapshot. You must call this procedure from the snapshot site for the new snapshot. For additional information, refer to page 5 – 11.

Syntax The parameters for the END_LOAD procedure are described in Table 12 – 50, and the exceptions are listed in Table 12 – 51. The syntax for this procedure is shown below.

```
DBMS_OFFLINE_SNAPSHOT.END_LOAD(
     gname           IN  VARCHAR2,
     sname           IN  VARCHAR2,
     snapshot_oname  IN  VARCHAR2)
```

Parameter	Description
gname	The name of the object group for the snapshot that you are creating using offline instantiation.
sname	The name of the schema for the new snapshot.
snapshot_oname	The name of the snapshot.

Table 12 – 50 Parameters for END_LOAD

Exception	Description
23430 badargument	Null or empty string for object group, schema, or snapshot name.
missingrepgroup	GNAME does not exist as a replicated object group.
nonsnapshot	This procedure must be called from the snapshot site.

Table 12 – 51 Exceptions for END_LOAD

DBMS_RECTIFIER_DIFF.DIFFERENCES

Purpose

To determine the differences between two tables. For additional information, refer to page 7 – 15.

Syntax

The parameters for the DIFFERENCES procedure are described in Table 12 – 52, and the exceptions are listed in Table 12 – 53. The syntax for this procedure is shown below.

```
DBMS_RECTIFIER_DIFF.DIFFERENCES(
            sname1               IN  VARCHAR2,
            oname1               IN  VARCHAR2,
            reference_site       IN  VARCHAR2 := '',
            sname2               IN  VARCHAR2,
            oname2               IN  VARCHAR2,
            comparison_site      IN  VARCHAR2 := '',
            where_clause         IN  VARCHAR2 := '',
            column_list          IN  VARCHAR2 := '', |
            array_columns        IN  dbms_utility.name_array
            missing_rows_sname   IN  VARCHAR2,
            missing_rows_oname1  IN  VARCHAR2,
            missing_rows_oname2  IN  VARCHAR2,
            missing_rows_site    IN  VARCHAR2 := '',
            max_missing          IN  INTEGER,
            commit_rows          IN  INTEGER := 500)
```

Parameter	Description
sname1	The name of the schema at REFERENCE_SITE.
oname1	The name of the table at REFERENCE_SITE.
reference_site	The name of the reference database site. The default, NULL, indicates the current site.
sname2	The name of the schema at COMPARISON_SITE.
oname2	The name of the table at COMPARISON_SITE.
comparison_site	The name of the comparison database site. The default, NULL, indicates the current site.
where_clause	Only rows satisfying this restriction are selected for comparison. The default, NULL or '', indicates that all rows should be compared.
column_list	A comma–separated list of one or more column names in the table that you want to have compared. You must not have any white space before or after the comma. The default, NULL or '', indicates that all columns be compared. The column list must include the primary key (or its SET_COLUMNS equivalent).

Table 12 – 52 Parameters for DBMS_RECTIFIER_DIFF.DIFFERENCES

Parameter	Description
`array_columns`	A PL/SQL table of column names that you want compared for the two tables. Indexing begins at 1, and the final element of the array must be NULL. If position 1 is NULL, all columns are used.
`missing_rows_sname`	The name of the schema in which the missing rows tables are located.
`missing_rows_oname1`	The name of the table at MISSING_ROWS_SITE that stores information about the rows in the table at REFERENCE site missing from the table at COMPARISON site and the rows at COMPARISON site missing from the table at REFERENCE site.
`missing_rows_oname2`	The name of the table at MISSING_ROWS_SITE that stores about the missing rows. This table has three columns: the rowid of the row in the MISSING_ROWS_ONAME1 table, the name of the site at which the row is present, and the name of the site from which the row is absent.
`missing_rows_site`	The name of the site where the MISSING_ROWS_ONAME1 and MISSING_ROWS_ONAME2 tables are located. The default, NULL or '', indicates that the tables are located at the current site.
`max_missing`	The maximum number of rows that should be inserted into the MISSING_ROWS_ONAME1 table. Once this number is reached, the routine returns normally, even if more differences exist.
`commit_rows`	The maximum number of rows to insert into MISSING_ROWS_ONAME1 before a COMMIT occurs. By default, a COMMIT occurs after 500 inserts. An empty string ('') or NULL indicates that a COMMIT should only be issued after all rows have been inserted.

Table 12 – 52 Parameters for DBMS_RECTIFIER_DIFF.DIFFERENCES

Exception	Description
23365 nosuchsite	Database site could not be found.
23366 badnumber	COMMIT_ROWS parameter less than 1.
23367 missingprimarykey	Column list must include primary key (or SET_COLUMNS equivalent).
23368 badname	NULL or empty string for table or schema name.

Table 12 – 53 Exceptions for DBMS_RECTIFIER_DIFF.DIFFERENCES

Exception	Description
23369 cannotbenull	Parameter cannot be NULL.
23370 notshapeequivalent	Tables being compared are not shape equivalent. Shape refers to the number of columns, their column names, and the column data types.
23371 unknowncolumn	Column does not exist.
23372 unsupportedtype	Type not supported.
commfailure	Remote site is inaccessible.
missingobject	Table does not exist.

Table 12 – 53 Exceptions for DBMS_RECTIFIER_DIFF.DIFFERENCES

Restrictions

The error ORA–00001: Unique constraint (XXXX.XXXXX) violated is issued when there are any unique or primary key constraints on the MISSING_ROWS_DATA table.

RECTIFIER_DIFF.DIFFERENCES can only be invoked against release 7.3 sites or higher.

DBMS_RECTIFIER_DIFF.RECTIFY

Purpose

To resolve the differences between two tables. For additional information, refer to page 7 – 15.

Syntax

The parameters for the RECTIFY procedure are described in Table 12 – 54, and the exceptions are listed in Table 12 – 55. The syntax for this procedure is shown below.

```
DBMS_RECTIFIER_DIFF.RECTIFY(
            sname1              IN  VARCHAR2,
            oname1              IN  VARCHAR2,
            reference_site      IN  VARCHAR2 := '',
            sname2              IN  VARCHAR2,
            oname2              IN  VARCHAR2,
            comparison_site     IN  VARCHAR2 := '',
            column_list         IN  VARCHAR2 := '', |
            array_columns       IN  dbms_utility.name_array
            missing_rows_sname  IN  VARCHAR2,
            missing_rows_oname1 IN  VARCHAR2,
            missing_rows_oname2 IN  VARCHAR2,
            missing_rows_site   IN  VARCHAR2 := '',
            commit_rows         IN  INTEGER := 500)
```

Parameter	Description
`sname1`	The name of the schema at REFERENCE_SITE.
`oname1`	The name of the table at REFERENCE_SITE.
`reference_site`	The name of the reference database site. The default, NULL, indicates the current site.
`sname2`	The name of the schema at COMPARISON_SITE.
`oname2`	The name of the table at COMPARISON_SITE.
`comparison_site`	The name of the comparison database site. The default, NULL, indicates the current site.
`column_list`	A comma–separated list of one or more column names being compared for the two tables. You must not have any white space before or after the comma. The default, NULL, indicates that all columns be compared.
`array_columns`	A PL/SQL table of column names being compared for the two tables. Indexing begins at 1, and the final element of the array must be NULL. If position 1 is NULL, all columns are used.
`missing_rows_sname`	The name of the schema in which the missing rows tables are located.

Table 12 – 54 Parameters for DBMS_RECTIFIER_DIFF.RECTIFY

Parameter	Description
`missing_rows_oname1`	The name of the table at MISSING_ROWS_SITE that stores information about the rows in the table at REFERENCE site missing from the table at COMPARISON site and the rows at COMPARISON site missing from the table at REFERENCE site.
`missing_rows_oname2`	The name of the table at MISSING_ROWS_SITE that stores about the missing rows. This table has three columns: the rowid of the row in the MISSING_ROWS_ONAME1 table, the name of the site at which the row is present, and the name of the site from which the row is absent.
`missing_rows_site`	The name of the site where the MISSING_ROWS_ONAME1 and MISSING_ROWS_ONAME2 tables are located. The default, NULL, indicates that the tables are located at the current site.
`commit_rows`	The maximum number of rows to insert to or delete from the reference or comparison table before a COMMIT occurs. By default, a COMMIT occurs after 500 inserts or 500 deletes. An empty string ('') or NULL indicates that a COMMIT should only be issued after all rows for a single table have been inserted or deleted.

Table 12 – 54 Parameters for DBMS_RECTIFIER_DIFF.RECTIFY

Exception	Description
23365 nosuchsite	Database site could not be found.
23366 badnumber	COMMIT_ROWS parameter less than 1.
23368 badname	NULL or empty string for table or schema name.
commfailure	Remote site is inaccessible.
missingobject	Table does not exist.

Table 12 – 55 Exceptions for DBMS_RECTIFIER_DIFF.RECTIFY

DBMS_REFRESH.ADD

Purpose

To add snapshots to a refresh group. For additional information, refer to page 3 – 13.

Syntax

The parameters for the ADD procedure are described in Table 12 – 56. The syntax for this procedure is shown below.

```
DBMS_REFRESH.ADD(name   IN     VARCHAR2,
                 list   IN     VARCHAR2,
                 lax    IN     BOOLEAN DEFAULT FALSE)
```

parameter	description
`name`	Name of the refresh group to which you want to add members.
`list`	Comma–separated list of snapshots that you want to add to the refresh group. (Synonyms are not supported.) Alternatively, you can supply a PL/SQL table of type DBMS_UTILITY.UNCL_ARRAY, where each element is the name of a snapshot.
`lax`	A snapshot can belong to only one refresh group at a time. If you are moving a snapshot from one group to another, you must set the LAX flag to TRUE to succeed. Oracle then automatically removes the snapshot from the other refresh group and updates its refresh interval to be that of its new group. Otherwise, the call to ADD generates an error message.

Table 12 – 56 Parameters for DBMS_REFRESH.ADD

DBMS_REFRESH.CHANGE

Purpose

To change the refresh interval for a snapshot group. For additional information, refer to page 3 – 14.

Syntax

The parameters for the CHANGE procedure are described in Table 12 – 57. The syntax for this procedure is shown below:

```
DBMS_REFRESH.CHANGE(
          name                IN     VARCHAR2,
          next_date           IN     DATE DEFAULT NULL,
          interval            IN     VARCHAR2 DEFAULT NULL,
          implicit_destroy    IN     BOOLEAN DEFAULT NULL,
          rollback_segment    IN     VARCHAR2 DEFAULT NULL,
          push_deferred_rpc   IN     BOOLEAN DEFAULT NULL,
          refresh_after_errorsIN     BOOLEAN DEFAULT NULL)
```

parameter	description
`name`	Name of the refresh group for which you want to alter the refresh interval.
`next_date`	Next date that you want a refresh to occur. By default, this date remains unchanged.
`interval`	Function used to calculate the next time to refresh the snapshots in the group. This interval is evaluated immediately before the refresh. Thus, you should select an interval that is greater than the time it takes to perform a refresh. By default, the interval remains unchanged.
`implicit_destroy`	Allows you to reset the value of the IMPLICIT_DESTROY flag. If this flag is set, Oracle automatically deletes the group if it no longer contains any members. By default, this flag remains unchanged.
`rollback_seg`	Allows you to change the rollback segment used. By default, the rollback segment remains unchanged. To reset this parameter to use the default rollback segment, specify 'null', including the quotes. Specifying null, without quotes, indicates that you do not want to change the rollback segment currently being used.

Table 12 – 57 Parameters for DBMS_REFRESH.CHANGE

parameter	description
`push_deferred_rpc`	Used by updatable snapshots only. Set this parameter to TRUE if you want to push changes from the snapshot to its associated master before refreshing the snapshot. Otherwise, these changes may appear to be temporarily lost. By default, this flag remains unchanged
`refresh_after_errors`	Used by updatable snapshots only. Set this parameter to TRUE if you want the refresh to proceed even if there are outstanding conflicts logged in the DefError view for the snapshot's master. By default, this flag remains unchanged

Table 12 – 57 Parameters for DBMS_REFRESH.CHANGE

DBMS_REFRESH.DESTROY

Purpose

To remove all of the snapshots from a refresh group and delete the refresh group. For additional information, refer to page 3 – 14.

Syntax

The parameter for the DESTROY procedure is described in Table 12 – 58. The syntax for this procedure is shown below:

```
DBMS_REFRESH.destroy(   name   IN     VARCHAR2)
```

parameter	description
name	Name of the refresh group that you want to destroy.

Table 12 – 58 Parameter for DBMS_REFRESH.DESTROY

DBMS_REFRESH.MAKE

Purpose

To specify the members of a refresh group and the time interval used to determine when the members of this group should be refreshed. For additional information, refer to page 3 – 13.

Syntax

The parameters for the MAKE procedure are described in Table 12 – 59. The syntax for this procedure is shown below:

```
DBMS_REFRESH.MAKE(
      name                 IN VARCHAR2,
      list                 IN VARCHAR2,    |
                           IN DBMS_UTILITY.UNCL_ARRAY,
      next_date            IN DATE,
      interval             IN VARCHAR2,
      implicit_destroy     IN BOOLEAN DEFAULT FALSE,
      lax                  IN BOOLEAN DEFAULT FALSE,
      job                  IN BINARY INTEGER DEFAULT 0,
      rollback_seg         IN VARCHAR2 DEFAULT NULL,
      push_deferred_rpc    IN BOOLEAN DEFAULT TRUE,
      refresh_after_errors IN BOOLEAN DEFAULT FALSE)
```

parameter	description
`name`	Unique name used to identify the refresh group. Refresh groups must follow the same naming conventions as tables.
`list`	Comma–separated list of snapshots that you want to refresh. (Synonyms are not supported.) These snapshots can be located in different schemas and have different master tables; however, all of the listed snapshots must be in your current database. Instead of a comma separated list, you can supply a PL/SQL table of names of snapshots that you want to refresh using the datatype DBMS_UTILITY.UNCL_ARRAY. If the table contains the names of N snapshots, the first snapshot should be in position 1 and the N + 1 position should be set to null.
`next_date`	Next date that you want a refresh to occur.

Table 12 – 59 Parameters for DBMS_REFRESH.MAKE
continued on next page

parameter	description
`interval`	Function used to calculate the next time to refresh the snapshots in the group. This field is used with the NEXT_DATE value. For example, if you specify NEXT_DAY(SYSDATE+1, ''MONDAY'') as your interval, and your NEXT_DATE evaluates to Monday, Oracle will refresh the snapshots every Monday. This interval is evaluated immediately before the refresh. Thus, you should select an interval that is greater than the time it takes to perform a refresh. For more information on how to specify a refresh interval, see page 3 – 16.
`implicit_destroy`	Set this argument to TRUE if you want to automatically delete the refresh group when it no longer contains any members. This flag is only checked when you call the SUBTRACT procedure. That is, setting this flag still allows you to create an empty refresh group.
`lax`	A snapshot can belong to only one refresh group at a time. If you are moving a snapshot from an existing group to a new refresh group, you must set the LAX flag to TRUE to succeed. Oracle then automatically removes the snapshot from the other refresh group and updates its refresh interval to be that of its new group. Otherwise, the call to MAKE generates an error message.
`job`	This parameter is needed by the Import utility. Use the default value, 0.
`rollback_seg`	Name of the rollback segment to use while refreshing snapshots. The default, null, uses the default rollback segment.
`push_deferred_rpc`	Used by updatable snapshots only. Use the default value, TRUE, if you want to push changes from the snapshot to its associated master before refreshing the snapshot. Otherwise, these changes may appear to be temporarily lost.
`refresh_after_errors`	Used by updatable snapshots only. Set this parameter to TRUE if you want the refresh to proceed even if there are outstanding conflicts logged in the DefError view for the snapshot's master.

Table 12 – 59 Parameters for DBMS_REFRESH.MAKE

DBMS_REFRESH.REFRESH

Purpose

To manually refresh a refresh group. For additional information, refer to page 3 – 19.

Syntax

The parameter for the REFRESH procedure is described in Table 12 – 60. The syntax for this procedure is shown below:

```
DBMS_REFRESH.REFRESH(   name   IN     VARCHAR2)
```

parameter	description
name	Name of the refresh group that you want to refresh manually.

Table 12 – 60 Parameters for DBMS_REFRESH.REFRESH

DBMS_REFRESH.SUBTRACT

Purpose

To remove snapshots from a refresh group. For additional information, refer to page 3 – 14.

Syntax

The parameters for the SUBTRACT procedure are described in Table 12 – 61. The syntax for this procedure is shown below:

```
DBMS_REFRESH.SUBTRACT(    name    IN     VARCHAR2,
                          list    IN     VARCHAR2,    |
                                  IN     DBMS_UTILITY.UNCL_ARRAY,
                          lax     IN     BOOLEAN DEFAULT FALSE)
```

parameter	description
name	Name of the refresh group from which you want to remove members.
list	Comma–separated list of snapshots that you want to remove from the refresh group. (Synonyms are not supported.) Alternatively, you can supply a PL/SQL table of type DBMS_UTILITY.UNCL_ARRAY, where each element is the name of a snapshot.
lax	Set this parameter to FALSE if you want Oracle to generate an error message if the snapshot you are attempting to remove is not a member of the refresh group.

Table 12 – 61 Parameters for DBMS_REFRESH.SUBTRACT

DBMS_REPCAT.ADD_GROUPED_COLUMN

Purpose

To add members to an existing column group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 23.

Syntax

The parameters for the ADD_GROUPED_COLUMN procedure are described in Table 12 – 62, and the exceptions are listed in Table 12 – 63. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_GROUPED_COLUMN(
              sname,              IN     VARCHAR2,
              oname,              IN     VARCHAR2,
              column_group        IN     VARCHAR2,
              list_of_column_names  IN     VARCHAR2,
                                  IN     DBMS_REPCAT.VARCHAR2S)
```

Parameter	Description
`sname`	The schema in which the replicated table is located.
`oname`	The name of the replicated table with which the column group is located.
`column_group`	The name of the column group to which you are adding members.
`list_of_column_names`	The names of the columns that you are adding to the designated column group. This can either be a comma–separated list or a PL/SQL table of column names. The PL/SQL table must be of type dbms_repcat.varchar2s. Use the single value ‘*’ to create a column group that contains all of the columns in your table.

Table 12 – 62 Parameters for ADD_GROUPED_COLUMN

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingobject	The given table does not exist.
missinggroup	The given column group does not exist.
missingcolumn	A given column does not exist in the designated table.
duplicatecolumn	The given column is already a member of another column group.
missingschema	The given schema does not exist

Table 12 – 63 Exceptions for ADD_GROUPED_COLUMN

DBMS_REPCAT.ADD_MASTER_DATABASE

Purpose

To add another master site to your replicated environment. This procedure regenerates all the triggers and their associated packages at existing master sites. You must call this procedure from the master definition site. For additional information, refer to page 4 – 17.

Syntax

The parameters for the ADD_MASTER_DATABASE procedure are described in Table 12 – 64, and the exceptions are listed in Table 12 – 65. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_MASTER_DATABASE(
             gname                 IN  VARCHAR2,
             master                IN  VARCHAR2,
             use_existing_objects  IN  BOOLEAN := TRUE,
             copy_rows             IN  BOOLEAN := TRUE,
             comment               IN  VARCHAR2 := '',
             propagation_mode      IN  VARCHAR2 := 'ASYNCHRONOUS')
```

Parameter	Description
gname	The name of the object group being replicated. This object group must already exist at the master definition site.
master	The fully qualified database name of the new master database.
use_existing_objects	Indicate TRUE if you want to reuse any objects of the same type and shape that already exist in the schema at the new master site. See page 4 – 15 for more information on how these changes are applied.
copy_rows	Indicate TRUE if you want the initial contents of a table at the new master site to match the contents of the table at the master definition site.
comment	This comment is added to the MASTER_COMMENT field of the RepSite view.
propagation_mode	Method of forwarding changes to and receiving changes from new master database. Accepted values are synchronous and asynchronous.

Table 12 – 64 Parameters for ADD_MASTER_DATABASE

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
notquiesced	The replicated object group has not been suspended.

Table 12 – 65 Exceptions for ADD_MASTER_DATABASE

Exception	Description
missingrepgroup	The object group does not exist at the given database site.
missingobject	A member of the replicated object group does not exist at the master definition site with a database status of VALID. If a package, package body, procedure, or function is invalid, Oracle recompiles the object once, in an attempt to make it valid.
commfailure	The new master is not accessible.
typefailure	An incorrect propagation mode was specified.
notcompat	Compatibility mode must be 7.3.0.0 or greater to use synchronous propagation.
repcompatnum	The GNAME is not an existing database schema at the remote master site and the remote master site is a pre–release 7.3 site.
duplrepgrp	The master site already exists.

Table 12 – 65 Exceptions for ADD_MASTER_DATABASE

DBMS_REPCAT.ADD_PRIORITY_*datatype*

Purpose

To add a member to a priority group. You must call this procedure from the master definition site. The procedure that you must call is determined by the datatype of your "priority" column. You must call this procedure once for each of the possible values of the "priority" column.

The available procedures are listed below:

- ADD_PRIORITY_CHAR
- ADD_PRIORITY_VARCHAR2
- ADD_PRIORITY_NUMBER
- ADD_PRIORITY_DATE
- ADD_PRIORITY_RAW

For additional information, refer to page 6 – 29.

Syntax

The parameters for the ADD_PRIORITY_VARCHAR2 procedure are described in Table 12 – 66, and the exceptions are listed in Table 12 – 67. The syntax for the ADD_PRIORITY_VARCHAR2 procedure is shown below. (The syntax for the remaining ADD_PRIORITY_*datatype* procedures is identical, except for the datatype of the value.)

```
DBMS_REPCAT.ADD_PRIORITY_VARCHAR2(
                        gname       IN    VARCHAR2,
                        pgroup      IN    VARCHAR2,
                        value       IN    VARCHAR2,
                        priority    IN    NUMBER)
```

Parameter	Description
`gname`	The replicated object group for which you are creating a priority group.
`pgroup`	The name of the priority group that you are creating.
`value`	The value of the priority group member. This would be one of the possible values of the associated "priority" column of a table using this priority group.
`priority`	The priority of this value. The higher the number, the higher the priority.

Table 12 – 66 Parameters for ADD_PRIORITY_*datatype*

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
duplicatevalue	The given value already exists in the priority group.

Exception	Description
duplicatepriority	The given priority already exists in the priority group.
missingrepgroup	The given replicated object group does not exist.
missingprioritygroup	The given priority group does not exist.
paramtype	The given value has the incorrect datatype for the priority group.

Table 12 – 67 Exceptions for ADD_PRIORITY_*datatype*

DBMS_REPCAT.ADD_SITE_PRIORITY_SITE

Purpose

To add a new site to a site priority group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 34.

Syntax

The parameters for the ADD_SITE_PRIORITY_SITE procedure are described in Table 12 – 68, and the exceptions are listed in Table 12 – 69. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_SITE_PRIORITY_SITE(
                        gname       IN     VARCHAR2,
                        name        IN     VARCHAR2
                        site        IN     VARCHAR2,
                        priority    IN     NUMBER)
```

Parameter	Description
`gname`	The replicated object group for which you are adding a site to a group.
`pgroup`	The name of the site priority group to which you are adding a member.
`site`	The global database name of the site that you are adding.
`priority`	The priority level of the site that you are adding. A higher number indicates a higher priority level.

Table 12 – 68 Parameters for ADD_SITE_PRIORITY_SITE

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingpriority	The given site priority group does not exist.
duplicatepriority	The given priority level already exists for another site in the group.

Table 12 – 69 Exceptions for ADD_SITE_PRIORITY_SITE

DBMS_REPCAT.ADD_*conflicttype*_RESOLUTION

Purpose

To designate a method for resolving an update, delete, or uniqueness conflict. You must call these procedures from the master definition site. The procedure that you need to call is determined by the type of conflict that the routine is used to resolve.

Conflict Type	Procedure Name
update	ADD_UPDATE_RESOLUTION
uniqueness	ADD_UNIQUE_RESOLUTION
delete	ADD_DELETE_RESOLUTION

For additional information, refer to page 6 – 25.

Syntax

The parameters for the ADD_UPDATE_RESOLUTION procedure are described in Table 12 – 70, and the exceptions are listed in Table 12 – 71. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_UPDATE_RESOLUTION(
          sname                 IN     VARCHAR2,
          oname                 IN     VARCHAR2,
          column_group          IN     VARCHAR2,
          sequence_no           IN     NUMBER,
          method                IN     VARCHAR2,
          parameter_column_nameIN      VARCHAR2,
          priority_group        IN     VARCHAR2 := NULL,
          function_name         IN     VARCHAR2 := NULL,
          comment               IN     VARCHAR2 := NULL)
```

The parameters for the ADD_DELETE_RESOLUTION procedure are described in Table 12 – 70, and the exceptions are listed in Table 12 – 71. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_DELETE_RESOLUTION(
       sname                  IN     VARCHAR2,
       oname                  IN     VARCHAR2,
       sequence_no            IN     NUMBER,
       parameter_column_name  IN     VARCHAR2,
       function_name          IN     VARCHAR2 := NULL,
       comment                IN     VARCHAR2 := NULL)
```

The parameters for the ADD_UNIQUE_RESOLUTION procedure are described in Table 12 – 70, and the exceptions are listed in Table 12 – 71. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ADD_UNIQUE_RESOLUTION(
       sname                 IN    VARCHAR2,
       oname                 IN    VARCHAR2,
       constraint_name       IN    VARCHAR2,
       sequence_no           IN    NUMBER,
       method                IN    VARCHAR2,
       parameter_column_name IN    VARCHAR2,
       function_name         IN    VARCHAR2 := NULL,
       comment               IN    VARCHAR2 := NULL)
```

Parameter	Description
sname	The name of the schema containing the table to be replicated.
oname	The name of the table for which you are adding a conflict resolution routine.
column_group	The name of the column group for which you are adding a conflict resolution routine. Column groups are required for update conflict resolution routines only.
constraint_name	The name of the unique constraint or unique index for which you are adding a conflict resolution routine. Use the name of the unique index if it differs from the name of the associated unique constraint. Constraint names are required for uniqueness conflict resolution routines only.
sequence_no	The order in which the designated conflict resolution methods should be applied.

Table 12 – 70 Parameters for ADD_UPDATE/DELETE_RESOLUTION
continued on next page

Parameter	Description
`method`	The type of conflict resolution routine that you want to create. This can be the name of one of the standard routines provided with symmetric replication, or, if you have written your own routine, you should choose USER FUNCTION, and provide the name of your routine as the FUNCTION_NAME argument. The methods supported in this release are: MINIMUM, MAXIMUM, LATEST TIMESTAMP, EARLIEST TIMESTAMP, ADDITIVE, AVERAGE, PRIORITY GROUP, SITE PRIORITY, OVERWRITE, and DISCARD (for update conflicts) and APPEND SITE NAME, APPEND SEQUENCE NUMBER, and DISCARD (for uniqueness conflicts). There are no standard methods for delete conflicts, so this argument is not used.
`parameter_column_name`	The name of the columns used to resolve the conflict. The standard methods operate on a single column. For example, if you are using the LATEST TIMESTAMP method for a column group, you should pass the name of the column containing the timestamp value as this argument. If your are using a USER FUNCTION, you can resolve the conflict using any number of columns. This argument accepts either a comma separated list of column names, or a PL/SQL table of type dbms_repcat.varchar2s. The single value '*' indicates that you want to use all of the columns in the table (or column group, for update conflicts) to resolve the conflict. If you specify '*', the columns will be passed to your function in alphabetical order.
`priority_group`	If you are using the PRIORITY GROUP or SITE PRIORITY update conflict resolution method, you must supply the name of the priority group that you have created. Instructions for creating a priority group are included on page 6 – 29. If you are using a different method, you can use the default value for this argument, NULL. This argument is applicable to update conflicts only.

Table 12 – 70 Parameters for ADD_UPDATE/DELETE_RESOLUTION
continued on next page

Parameter	Description
`function_name`	If you selected the USER FUNCTION method, or if you are adding a delete conflict resolution routine, you must supply the name of the conflict resolution routine that you have written. If you are using one of the standard methods, you can use the default value for this argument, NULL.
`comment`	This user comment is added to the RepResolution view.

Table 12 – 70 Parameters for ADD_UPDATE/DELETE_RESOLUTION

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingobject	The given object does not exist as a table in the given schema using row–level replication.
missingschema	The given schema does not exist.
missingcolumn	The column that you specified as part of the PARAMETER_COLUMN_NAME argument does not exist.
missingprioritygroup	The priority group that you specified does not exist for the table.
invalidmethod	The resolution method that you specified is not recognized.
invalidparameter	The number of columns that you specified for the PARAMETER_COLUMN_NAME argument is invalid. (The standard routines take only one column name.)
missingfunction	The user function that you specified does not exist.
missingconstraint	The constraint that you specified for a uniqueness conflict does not exist.
typefailure	The datatype of one of the values that you specified for the PARAMETER_COLUMN_NAME argument is not one of the types supported for the given method.

Table 12 – 71 Exceptions for ADD_UPDATE/DELETE_RESOLUTION

DBMS_REPCAT.ALTER_MASTER_PROPAGATION

Purpose

To alter the propagation method for a given object group at a given master site. This object group must be quiesced. You must call this procedure from the master definition site. If the master appears in the dblink_list or dblink_table, ALTER_MASTER_PROPAGATION ignores that database link. There is no way to change the propagation mode from a master to itself. For additional information, refer to page 4 – 34.

Syntax

The parameters for the ALTER_MASTER_PROPAGATION procedure are described in Table 12 – 72, and the exceptions are listed in Table 12 – 73. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_MASTER_PROPAGATION(
           gname             IN  VARCHAR2,
           master            IN  VARCHAR2,
           dblink_list       IN  VARCHAR2, |
           dblink_table      IN  dbms_utility.dblink_array,
           propagation_mode  IN  VARCHAR2,
           comment           IN  VARCHAR2 := '')
```

Parameter	Description
`gname`	The name of the object group that you want to alter.
`master`	The name of the master site at which to alter the object group.
`dblink_list`	A comma–separated list of database links for which to alter propagation. If null, all masters except the master site being altered will be used by default.
`dblink_table`	A PL/SQL table, indexed from position 1, of database links for which to alter propagation.
`propagation_mode`	Determines the manner in which changes from the given master site are propagated to the sites identified by the list of database links. Appropriate values are SYNCHRONOUS and ASYNCHRONOUS.
`comment`	This comment is added to the RepProp view.

Table 12 – 72 Parameters for ALTER_MASTER_PROPAGATION

Exception	Description
nonmasterdef	The local site is not the master definition site.
notquiesced	The local site is not quiesced.

Table 12 – 73 Exceptions for ALTER_MASTER_PROPAGATION

Exception	Description
typefailure	The propagation mode specified was not recognized.
nonmaster	The list of database links includes a site that is not a master site.

Table 12 – 73 Exceptions for ALTER_MASTER_PROPAGATION

DBMS_REPCAT.ALTER_MASTER_REPOBJECT

Purpose

To alter an object in your replicated environment. You must call this procedure from the master definition site. For additional information, refer to page 4 – 45.

Syntax

The parameters for the ALTER_MASTER_REPOBJECT procedure are described in Table 12 – 74, and the exceptions are listed in Table 12 – 75. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_MASTER_REPOBJECT(
                 sname        IN     VARCHAR2,
                 oname        IN     VARCHAR2,
                 type         IN     VARCHAR2,
                 ddl_text     IN     VARCHAR2,
                 comment      IN     VARCHAR2 := '',
                 retry        IN     BOOLEAN := FALSE)
```

Note: If the DDL is supplied without specifying a schema, the default schema is the replication administrator's schema. Be sure to specify the schema if it is other than the replication administrator's schema.

Parameter	Description
`sname`	The schema containing the object that you want to alter.
`oname`	The name of the object that you want to alter.
`type`	The type of the object that you are altering. The types supported are: TABLE, INDEX, SYNONYM, TRIGGER, VIEW, PROCEDURE, FUNCTION, PACKAGE, and PACKAGE BODY.
`ddl_text`	The DDL text that you want used to alter the object. Oracle does not parse this DDL before applying it; therefore, you must ensure that your DDL text provides the appropriate schema and object name for the object being altered.

Table 12 – 74 Parameters for ALTER_MASTER_REPOBJECT

Parameter	Description
`comment`	If not null, this comment will be added to the COMMENT field of the RepObject view.
`retry`	If retry is TRUE, ALTER_MASTER_REPOBJECT alters the object only at masters whose object status is not VALID.

Table 12 – 74 Parameters for ALTER_MASTER_REPOBJECT

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
notquiesced	The associated object group has not been suspended.
missingobject	The object identified by SNAME and ONAME does not exist.
typefailure	The given type parameter is not supported.
ddlfailure	DDL at the master definition site did not succeed.
commfailure	At least one master site is not accessible.

Table 12 – 75 Exceptions for ALTER_MASTER_REPOBJECT

DBMS_REPCAT.ALTER_PRIORITY

Purpose

To alter the priority level associated with a given priority group member. You must call this procedure from the master definition site. For additional information, refer to page 6 – 31.

Syntax

The parameters for the ALTER_PRIORITY procedure are described in Table 12 – 76, and the exceptions are listed in Table 12 – 77. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_PRIORITY(
                gname         IN     VARCHAR2,
                pgroup        IN     VARCHAR2,
                old_priority  IN     NUMBER,
                new_priority  IN     NUMBER)
```

Parameter	Description
`gname`	The replicated object group with which the priority group is associated.
`pgroup`	The name of the priority group containing the priority that you want to alter.
`old_priority`	The current priority level of the priority group member.
`new_priority`	The new priority level that you want assigned to the priority group member.

Table 12 – 76 Parameters for ALTER_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
duplicatepriority	The new priority level already exists in the priority group.
missingrepgroup	The given replicated object group does not exist.
missingvalue	The value was not registered by a call to DBMS_REPCAT.ADD_PRIORITY_*datatype.*

Table 12 – 77 Exceptions for ALTER_PRIORITY

DBMS_REPCAT.ALTER_PRIORITY_*datatype*

Purpose

To alter the value of a member to a priority group. You must call this procedure from the master definition site. The procedure that you must call is determined by the datatype of your “priority” column. The available procedures are listed below.

- ALTER_PRIORITY_CHAR
- ALTER_PRIORITY_VARCHAR2
- ALTER_PRIORITY_NUMBER
- ALTER_PRIORITY_DATE
- ALTER_PRIORITY_RAW

For additional information, refer to page 6 – 30.

Syntax

The parameters for the ALTER_PRIORITY_VARCHAR2 procedure are described in Table 12 – 78, and the exceptions are listed in Table 12 – 79. The syntax for this procedure is shown below. (The syntax for the remaining ALTER_PRIORITY_*datatype* procedures is identical, except for the datatype of the old and new values.)

```
DBMS_REPCAT.ALTER_PRIORITY_VARCHAR2(
                       gname        IN     VARCHAR2,
                       pgroup       IN     VARCHAR2,
                       old_value    IN     VARCHAR2,
                       new_value    IN     VARCHAR2)
```

Parameter	Description
`gname`	The replicated object group with which the priority group is associated.
`pgroup`	The name of the priority group containing the value that you want to alter.
`old_value`	The current value of the priority group member.
`new_value`	The new value that you want assigned to the priority group member.

Table 12 – 78 Parameters for ALTER_PRIORITY_VARCHAR2

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
duplicatevalue	The new value already exists in the priority group.
missingrepgroup	The given replicated object group does not exist.

Table 12 – 79 Exceptions for ALTER_PRIORITY_VARCHAR2

continued on next page

Exception	Description
missingprioritygroup	The given priority group does not exist.
missingvalue	The old value does not already exist.
paramtype	The new value has the incorrect datatype for the priority group.

Table 12 – 79 Exceptions for ALTER_PRIORITY_VARCHAR2

DBMS_REPCAT.ALTER_SITE_PRIORITY

Purpose

To alter the priority level associated with a given site. You must call this procedure from the master definition site. For additional information, refer to page 6 – 35.

Syntax

The parameters for the ALTER_SITE_PRIORITY procedure are described in Table 12 – 80, and the exceptions are listed in Table 12 – 81. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_SITE_PRIORITY(gname        IN   VARCHAR2,
                                name         IN   VARCHAR2
                                old_priority IN   NUMBER,
                                new_priority IN   NUMBER)
```

Parameter	Description
`gname`	The replicated object group with which the site priority group is associated.
`name`	The name of the site priority group whose member you are altering.
`old_priority`	The current priority level of the site whose priority level you want to change.
`new_priority`	The new priority level for the site. A higher number indicates a higher priority level.

Table 12 – 80 Parameters for ALTER_SITE_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingpriority	The old priority level is not associated with any group members.
duplicatepriority	The new priority level already exists for another site in the group.
missingvalue	The old value does not already exist.
paramtype	The new value has the incorrect datatype for the priority group.

Table 12 – 81 Exceptions for ALTER_SITE_PRIORITY

DBMS_REPCAT.ALTER_SITE_PRIORITY_SITE

Purpose

To alter the site associated with a given priority level. You must call this procedure from the master definition site. For additional information, refer to page 6 – 35.

Syntax

The parameters for the ALTER_SITE_PRIORITY_SITE procedure are described in Table 12 – 82, and the exceptions are listed in Table 12 – 83. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_SITE_PRIORITY_SITE(gname     IN  VARCHAR2,
                                     name      IN  VARCHAR2
                                     old_site  IN  VARCHAR2,
                                     new_site  IN  VARCHAR2)
```

Parameter	Description
gname	The replicated object group with which the site priority group is associated.
name	The name of the site priority group whose member you are altering.
old_site	The current global database name of the site whose name you want to change.
new_site	The new global database name that you want to associate with the current priority level.

Table 12 – 82 Parameters for ALTER_SITE_PRIORITY_SITE

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingpriority	The given site priority group does not exist
missingvalue	The old site is not a group member.
duplicatesite	The new site already exists in the group with a different priority level.

Table 12 – 83 Exceptions for ALTER_SITE_PRIORITY_SITE

DBMS_REPCAT.ALTER_SNAPSHOT_PROPAGATION

Purpose

To alter the propagation method for a given object group at the current snapshot site. This procedure pushes the deferred transaction queue at the snapshot site, locks the snapshot base tables, and regenerates any triggers and their associated packages. You must call this procedure from the snapshot site. For additional information, refer to page 5 – 21.

Syntax

The parameters for the ALTER_SNAPSHOT_PROPAGATION procedure are described in Table 12 – 84, and the exceptions are listed in Table 12 – 85. The syntax for this procedure is shown below:

```
DBMS_REPCAT.ALTER_SNAPSHOT_PROPAGATION(
           gname             IN  VARCHAR2,
           propagation_mode  IN  VARCHAR2,
           comment           IN  VARCHAR2 := ''
           execute_as_user   IN  BOOLEAN := FALSE)
```

Parameter	Description
`gname`	The name of the object group that you want to alter.
`propagation_mode`	Determines the manner in which changes from the current snapshot site are propagated to its associated master site. Appropriate values are SYNCHRONOUS and ASYNCHRONOUS.
`comment`	This comment is added to the RepProp view.
`execute_as_user`	The default, FALSE, indicates that a deferred call is authenticated at the remote system using the authentication context of the user who originally queued the deferred call (as indicated in the ORIGIN_USER column of the DefTran view). Set this to TRUE if you want the execution of a deferred call to be authenticated at the remote system using the authentication context of the session user.

Table 12 – 84 Parameters for ALTER_SNAPSHOT_PROPAGATION

Exception	Description
notcompat	Only databases operating in 7.3.0 or later mode can use this procedure.
missingrepgroup	The given replicated object group does not exist.
typefailure	The propagation mode was specified incorrectly.

Table 12 – 85 Exceptions for ALTER_SNAPSHOT_PROPAGATION

DBMS_REPCAT.CANCEL_STATISTICS

Purpose

To stop collecting statistics about the successful resolution of update, uniqueness, and delete conflicts for a table. For additional information, refer to page 7 – 9.

Syntax

The parameters for the CANCEL_STATISTICS procedure are described in Table 12 – 86, and the exceptions are listed in Table 12 – 87. The syntax for this procedure is shown below:

```
DBMS_REPCAT.CANCEL_STATISTICS(sname  IN     VARCHAR2,
                              oname  IN     VARCHAR2)
```

Parameter	Description
sname	The name of the schema in which the table is located.
oname	The name of the table for which you do not want to gather conflict resolution statistics.

Table 12 – 86 Parameters for CANCEL_STATISTICS

Exception	Description
missingschema	The given schema does not exist.
missingobject	The given table does not exist.
statnotreg	The given table is not currently registered to collect statistics.

Table 12 – 87 Exceptions for CANCEL_STATISTICS

DBMS_REPCAT.COMMENT_ON_COLUMN_GROUP

Purpose

To update the comment field in the RepColumn_Group view for a column group. This comment is not added at all master sites until the next call to DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT. For additional information, refer to page 7 – 18.

Syntax

The parameters for the COMMENT_ON_COLUMN_GROUP procedure are described in Table 12 – 88, and the exceptions are listed in Table 12 – 89. The syntax for this procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_COLUMN_GROUP(
                              sname        IN     VARCHAR2,
                              oname        IN     VARCHAR2,
                              column_group IN     VARCHAR2,
                              comment      IN     VARCHAR2)
```

Parameter	Description
sname	The name of the schema in which the object is located.
oname	The name of the replicated table with which the column group is associated.
column_group	The name of the column group.
comment	The text of the updated comment that you want included in the GROUP_COMMENT field of the RepColumn_Group view.

Table 12 – 88 Parameters for COMMENT_ON_COLUMN_GROUP

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missinggroup	The given column group does not exist.

Table 12 – 89 Exceptions for COMMENT_ON_COLUMN_GROUP

DBMS_REPCAT.COMMENT_ON_PRIORITY_GROUP/ DBMS_REPCAT.COMMENT_ON_SITE_PRIORITY

Purpose

To update the comment field in the RepPriority_Group view for a priority group. This comment is not added at all master sites until the next call to DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT. For additional information, refer to page 7 – 18.

To update the comment field in the RepPriority_Group view for a site priority group, call the COMMENT_ON_SITE_PRIORITY procedure in the DBMS_REPCAT package. (This procedure is a wrapper for the COMMENT_ON_COLUMN_GROUP procedure and is provided as a convenience only.) This procedure must be issued at the master definition site.

Syntax

The parameters for the COMMENT_ON_PRIORITY_GROUP and COMMENT_ON_SITE_PRIORITY procedures are described in Table 12 – 90, and the exceptions are listed in Table 12 – 91.

The syntax for the COMMENT_ON_PRIORITY_GROUP procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_PRIORITY_GROUP(
                        gname        IN     VARCHAR2,
                        pgroup       IN     VARCHAR2,
                        comment      IN     VARCHAR2)
```

The syntax for the COMMENT_ON_SITE_PRIORITY procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_SITE_PRIORITY(
                        gname        IN     VARCHAR2,
                        name         IN     VARCHAR2,
                        comment      IN      VARCHAR2)
```

Parameter	Description
gname	The name of the replicated object group.
pgroup/name	The name of the priority or site priority group.
comment	The text of the updated comment that you want included in the PRIORITY_COMMENT field of the RepPriority_Group view.

Table 12 – 90 Parameters for COMMENT_ON_PRIORITY_GROUP and COMMENT_ON_SITE_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingrepgroup	The given replicated object group does not exist.

Exception	Description
missingprioritygroup	The given priority group does not exist.
missingpriority	The given site priority group does not exist.

Table 12 – 91 Exceptions for COMMENT_ON_PRIORITY_GROUP and COMMENT_ON_SITE_PRIORITY

DBMS_REPCAT.COMMENT_ON_REPGROUP

Purpose

To update the comment field in the RepGroup view for a replicated object group. For additional information, refer to page 7 – 18. This procedure must be issued at the master definition site.

Syntax

The parameters for the COMMENT_ON_REPGROUP procedure are described in Table 12 – 92, and the exceptions are listed in Table 12 – 93. The syntax for this procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_REPGROUP( gname   IN VARCHAR2,
                                 comment IN VARCHAR2)
```

Parameter	Description
`gname`	The name of the object group that you want to comment on.
`comment`	The updated comment to include in the GROUP_COMMENT field of the RepGroup view.

Table 12 – 92 Parameters for COMMENT_ON_REPGROUP

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
commfailure	At least one master site is not accessible.
missingobjectgroup	The object group does not exist.

Table 12 – 93 Exceptions for COMMENT_ON_REPGROUP Procedure

DBMS_REPCAT.COMMENT_ON_REPSITES

Purpose

To update the comment field in the RepSite view for a replicated site. For additional information, refer to page 7 – 18. This procedure must be issued at the master definition site.

Syntax

The parameters for the COMMENT_ON_REPGROUP procedure are described in Table 12 – 94, and the exceptions are listed in Table 12 – 95. The syntax for this procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_REPSITES(
                           gname   IN VARCHAR2,
                           master  IN VARCHAR,
                           comment IN VARCHAR2)
```

Parameter	Description
gname	The name of the object group. This avoids confusion if a database is a master site in more than one replicated environment.
master	The fully qualified database name of the master site that you want to comment on.
comment	The text of the updated comment that you want to include in the MASTER_COMMENT field of the RepSites view.

Table 12 – 94 Parameters for COMMENT_ON_REPSITES

Exception	Description
nonmasterdef	The invocation site is not the master definition site.

Table 12 – 95 Exceptions for COMMENT_ON_REPSITES

DBMS_REPCAT.COMMENT_ON_REPOBJECT

Purpose

To update the comment field in the RepObject view for a replicated object. For additional information, refer to page 7 – 18. This procedure must be issued at the master definition site.

Syntax

The parameters for the COMMENT_ON_REPOBJECT procedure are described in Table 12 – 96, and the exceptions are listed in Table 12 – 97. The syntax for this procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_REPOBJECT( sname   IN  VARCHAR2,
                                  oname   IN  VARCHAR2,
                                  type    IN  VARCHAR2,
                                  comment IN  VARCHAR2)
```

Parameter	Description
`sname`	The name of the schema in which the object is located.
`oname`	The name of the object that you want to comment on.
`type`	The type of the object.
`comment`	The text of the updated comment that you want to include in the OBJECT_COMMENT field of the RepObject view.

Table 12 – 96 Parameters for COMMENT_ON_REPOBJECT

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist.
typefailure	The given type parameter is not supported.
commfailure	At least one master site is not accessible.

Table 12 – 97 Exceptions for COMMENT_ON_REPOBJECT

DBMS_REPCAT.COMMENT_ON_*conflicttype*_RESOLUTION

Purpose

To update the comment field in the RepResolution view for a conflict resolution routine. The procedure that you need to call is determined by the type of conflict that the routine is used to resolve. These procedures must be issued at the master definition site.

Conflict Type	Procedure Name
update	COMMENT_ON_UPDATE_RESOLUTION
uniqueness	COMMENT_ON_UNIQUE_RESOLUTION
delete	COMMENT_ON_DELETE_RESOLUTION

The comment is not added at all master sites until the next call to DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT. For additional information, refer to page 7 – 18.

Syntax

The parameters for the COMMENT_ON_*conflicttype*_RESOLUTION procedures are described in Table 12 – 98, and the exceptions are listed in Table 12 – 99.

The syntax for the COMMENT_ON_UPDATE_RESOLUTION procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_UPDATE_RESOLUTION(
                              sname        IN    VARCHAR2,
                              oname        IN    VARCHAR2,
                              column_group IN    VARCHAR2,
                              sequence_no  IN    NUMBER,
                              comment      IN    VARCHAR2)
```

The syntax for the COMMENT_ON_UNIQUE_RESOLUTION procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_UNIQUE_RESOLUTION(
                        sname              IN    VARCHAR2,
                        oname              IN    VARCHAR2,
                        constraint_name    IN    VARCHAR2,
                        sequence_no        IN    NUMBER,
                        comment            IN    VARCHAR2)
```

The syntax for the COMMENT_ON_DELETE_RESOLUTION procedure is shown below:

```
DBMS_REPCAT.COMMENT_ON_DELETE_RESOLUTION(
                              sname        IN    VARCHAR2,
                              oname        IN    VARCHAR2,
                              sequence_no  IN    NUMBER,
                              comment      IN    VARCHAR2)
```

Parameter	Description
`sname`	The name of the schema.
`oname`	The name of the replicated table with which the conflict resolution routine is associated.
`column_group`	The name of the column group with which the update conflict resolution routine is associated.
`constraint_name`	The name of the Unique constraint with which the uniqueness conflict resolution routine is associated.
`sequence_no`	The sequence number of the conflict resolution procedure.
`comment`	The text of the updated comment that you want included in the RESOLUTION_COMMENT field of the RepResolution view.

Table 12 – 98 Parameters for COMMENT_ON_UPDATE/UNIQUE/DELETE_RESOLUTION

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given procedure does not exist.
missingresolution	SEQUENCE_NO or COLUMN_GROUP is not registered.

Table 12 – 99 Exceptions for COMMENT_ON_UPDATE/UNIQUE/DELETE_RESOLUTION

DBMS_REPCAT.CREATE_MASTER_REPGROUP

Purpose

To create a new, empty, quiesced master replication object group. For additional information, refer to page 4 – 13.

Syntax

The parameters for the CREATE_MASTER_REPGROUP procedure are described in Table 12 – 100, and the exceptions are listed in Table 12 – 101. The syntax for this procedure is shown below:

```
DBMS_REPCAT.CREATE_MASTER_REPGROUP(
          gname          IN VARCHAR2,
          group_comment  IN VARCHAR2 := '',
          master_comment IN VARCHAR2 := '')
```

Parameter	Description
`gname`	The name of the object group that you want to create.
`group_comment`	This comment is added to the RepCat view.
`master_comment`	This comment is added to the RepGroup view.

Table 12 – 100 Parameters for CREATE_MASTER_REPGROUP

Exception	Description
duplicaterepgroup	The object group already exists.
ddlfailure	There is a problem creating the rep$what_am_i package or package body.
norepopt	The advanced replication option is not installed.
notcompat	The GNAME is not a schema name, and the master definition site is pre–release 7.3.
missingrepgrp	The object group name was not specified.

Table 12 – 101 Exceptions for CREATE_MASTER_REPGROUP

DBMS_REPCAT.CREATE_MASTER_REPOBJECT

Purpose

To indicate that an object is a replicated object. For additional information, refer to page 4 – 14.

Syntax

The parameters for the CREATE_MASTER_REPOBJECT procedure are shown in Table 12 – 102, and the exceptions are listed in Table 12 – 103. The syntax for this procedure is shown below:

```
DBMS_REPCAT.CREATE_MASTER_REPOBJECT(
                  sname                IN VARCHAR2,
                  oname                IN VARCHAR2,
                  type                 IN VARCHAR2,
                  use_existing_object  IN BOOLEAN := TRUE,
                  ddl_text             IN VARCHAR2 := NULL,
                  comment              IN VARCHAR2 := '',
                  retry                IN BOOLEAN := FALSE
                  copy_rows            IN BOOLEAN := TRUE,
                  gname                IN VARCHAR2 := '')
```

Note: If the DDL is supplied without specifying a schema, the default schema is the replication administrator's schema. Be sure to specify the schema if it is other than the replication administrator's schema.

Parameter	Description
`sname`	The name of the schema in which the object that you want to replicate is located.
`oname`	The name of the object that you are replicating. If DDL_TEXT is NULL, this object must already exist in the given schema. To ensure uniqueness, table names should be a maximum of 27 bytes long, and packages should be no more than 24 bytes.

Table 12 – 102 Parameters for CREATE_MASTER_REPOBJECT
continued on next page

Parameter	Description
`type`	The type of the object that you are replicating. The types supported are: TABLE, INDEX, SYNONYM, TRIGGER,VIEW, PROCEDURE, FUNCTION, PACKAGE, and PACKAGE BODY.
`use_existing_object`	Indicate TRUE if you want to reuse any objects of the same type and shape at the current master sites. See Table 12 – 104 for more information on how these changes are applied.
`ddl_text`	If the object does not already exist at the master definition site, you must supply the DDL text necessary to create this object. PL/SQL packages, package bodies, procedures, and functions must have a trailing semicolon. SQL statements do not end with a trailing semicolon. Oracle does not parse this DDL before applying it; therefore, you must ensure that your DDL text provides the appropriate schema and object name for the object being created.
`comment`	This comment will be added to the OBJECT_COMMENT field of the RepObject view.
`retry`	Indicate TRUE if you want Oracle to reattempt to create an object that it was previously unable to create. Use RETRY if the error was transient or has since been rectified; for example, if you previously had insufficient resources. If RETRY is TRUE, Oracle creates the object only at master sites whose object status is not VALID.
`copy_rows`	Indicate TRUE if you want the initial contents of a newly replicated object to match the contents of the object at the master definition site. See Table 12 – 104 for more information.
`gname`	The name of the object group in which you want to create the replicated object. The schema name is used as the default object group name is none is specified.

Table 12 – 102 Parameters for CREATE_MASTER_REPOBJECT

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
notquiesced	The replicated object group has not been suspended.
duplicateobject	The given object already exists in the replicated object group and retry is FALSE, or if a name conflict occurs.
missingobject	The object identified by SNAME and ONAME does not exist and appropriate DDL has not been provided.
typefailure	Objects of the given type cannot be replicated
ddlfailure	DDL at the master definition site did not succeed.
commfailure	At least one master site is not accessible.

Table 12 – 103 Exceptions for CREATE_MASTER_REPOBJECT

Object Already Exists?	COPY_ROWS	USE_EXISTING _OBJECT	Result
yes	TRUE	TRUE	*duplicateobject* message if objects do not match. For tables, use data from master definition site.
yes	FALSE	TRUE	*duplicateobject* message if objects do not match. For tables, Admin must ensure contents are identical.
yes	TRUE/FALSE	FALSE	*duplicateobject* message

Table 12 – 104 Object Creation at Master Sites, continued on next page

Object Already Exists?	COPY_ROWS	USE_EXISTING _OBJECT	Result
no	TRUE	TRUE/FALSE	Object is created. Tables populated using data from master definition site.
no	FALSE	TRUE/FALSE	Object is created. Admin must populate tables and ensure consistency of tables at all sites.

Table 12 – 104 Object Creation at Master Sites

DBMS_REPCAT.CREATE_SNAPSHOT_REPGROUP

Purpose

To create a new, empty, snapshot replication object group in your local database. For additional information, refer to page 5 – 6.

Syntax

The parameters for the CREATE_SNAPSHOT_REPGROUP procedure are described in Table 12 – 105, and the procedures are listed in Table 12 – 106. The syntax for this procedure is shown below:

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPGROUP(
           gname             IN VARCHAR2,
           master            IN VARCHAR2,
           comment           IN  VARCHAR2 := '',
           propagation_mode  IN VARCHAR2 := 'ASYNCHRONOUS')
```

Parameter	Description
`gname`	The name of the replicated object group. This object group must exist at the given master site.
`master`	The fully qualified database name of the database in the replicated environment to use as the master.
`comment`	This comment is added to the GROUP_COMMENT field of the RepCat view.
`propagation_mode`	The method of propagation for all updatable snapshots in the object group. Acceptable values are SYNCHRONOUS and ASYNCHRONOUS.

Table 12 – 105 Parameters for CREATE_SNAPSHOT_REPGROUP

Exception	Description
duplicaterepgroup	The object group already exists at the invocation site.
nonmaster	The given database is not a master site.
commfailure	The given database is not accessible.
norepopt	The advanced replication option is not installed.
typefailure	The propagation mode was specified incorrectly.
notcompatible	Propagation mode must be ASYNCHRONOUS for pre–release 7.3 sites.

Table 12 – 106 Exceptions for CREATE_SNAPSHOT_REPGROUP

DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT

Purpose

To add a replicated object to your snapshot site. For additional information, refer to page 5 – 7.

Syntax

The parameters for the CREATE_SNAPSHOT_REPOBJECT procedure are shown in Table 12 – 107, and the exceptions are listed in Table 12 – 108. The syntax for this procedure is shown below:

```
DBMS_REPCAT.CREATE_SNAPSHOT_REPOBJECT(
                       sname          IN VARCHAR2,
                       oname          IN VARCHAR2,
                       type           IN VARCHAR2,
                       ddl_text       IN VARCHAR2 := NULL,
                       comment        IN VARCHAR2 := '',
                       gname          IN VARCHAR2 := ''
                       gen_obj_owner  IN VARCHAR2 := '')
```

Note: If the DDL is supplied without specifying a schema, the default schema is the replication administrator's schema. Be sure to specify the schema if it is other than the replication administrator's schema.

Parameter	Description
`sname`	The name of the schema in which the object is located.
`oname`	The name of the object that you want to add to the replicated snapshot object group. ONAME must exist at the associated master site.
`type`	The type of the object that you are replicating. The types supported for snapshot sites are: PACKAGE, PACKAGE BODY, PROCEDURE, FUNCTION, SNAPSHOT, SYNONYM, and VIEW.
`ddl_text`	For objects of type SNAPSHOT, the DDL text needed to create the object; for other types, use the default, NULL. If a snapshot with the same name already exists, Oracle ignores the DDL and registers the existing snapshot as a replicated object. If the master table for a snapshot does not exist in the replicated object group of the master site designated for this schema, Oracle raises a *missingobject error.*
`comment`	This comment is added to the OBJECT_COMMENT field of the RepObject view.

Table 12 – 107 Parameters for CREATE_SNAPSHOT_REPOBJECT

Parameter	Description
`gname`	The name of the replicated object group to which you are adding an object. The schema name is used as the default group name if none is specified.
gen_obj_owner	The name of the user you want to assign as owner of the transaction.

Table 12 – 107 Parameters for CREATE_SNAPSHOT_REPOBJECT

Exception	Description
nonsnapshot	The invocation site is not a snapshot site.
nonmaster	The master is no longer a master site.
missingobject	The given object does not exist in the master's replicated object group.
duplicateobject	The given object already exists with a different shape.
typefailure	The type is not an allowable type.
ddlfailure	The DDL did not succeed.
commfailure	The master site is not accessible.
missingschema	The schema does not exist as a database schema.

Table 12 – 108 Exceptions for CREATE_SNAPSHOT_REPOBJECT

DBMS_REPCAT.DEFINE_COLUMN_GROUP

Purpose

To create an empty column group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 24.

Syntax

The parameters for the DEFINE_COLUMN_GROUP procedure are described in Table 12 – 109, and the exceptions are listed in Table 12 – 110. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DEFINE_COLUMN_GROUP(
                 sname        IN    VARCHAR2,
                 oname        IN    VARCHAR2,
                 column_group IN    VARCHAR2,
                 comment      IN    VARCHAR2 := NULL)
```

Parameter	Description
sname	The schema in which the replicated table is located.
oname	The name of the replicated table for which you are creating a column group.
column_group	The name of the column group that you want to create.
comment	This user text is displayed in the RepColumnGroup view.

Table 12 – 109 Parameters for DEFINE_COLUMN_GROUP

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingobject	The given table does not exist.
duplicategroup	The given column group already exists for the table.

Table 12 – 110 Exceptions for DEFINE_COLUMN_GROUP

DBMS_REPCAT.DEFINE_PRIORITY_GROUP

Purpose

To create a new priority group for a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 29.

Syntax

The parameters for the DEFINE_PRIORITY_GROUP procedure are described in Table 12 – 111, and the exceptions are listed in Table 12 – 112. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DEFINE_PRIORITY_GROUP(
                 gname        IN     VARCHAR2,
                 pgroup       IN     VARCHAR2,
                 datatype     IN     VARCHAR2,
                 fixed_length IN     INTEGER := NULL,
                 comment      IN     VARCHAR2 := NULL)
```

Parameter	Description
gname	The replicated object group for which you are creating a priority group.
pgroup	The name of the priority group that you are creating.
datatype	The datatype of the priority group members. The datatypes supported are: CHAR, VARCHAR2, NUMBER, DATE, and RAW.
fixed_length	You must provide a column length for the CHAR datatype. All other types can use the default, NULL.
comment	This user comment is added to the RepPriority view.

Table 12 – 111 Parameters for DEFINE_PRIORITY_GROUP

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
duplicateprioritygroup	The given priority group already exists in the replicated object group.
typefailure	The given datatype is not supported.

Table 12 – 112 Exceptions for DEFINE_PRIORITY_GROUP

DBMS_REPCAT.DEFINE_SITE_PRIORITY

Purpose

To create a new site priority group for a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 34.

Syntax

The parameters for the DEFINE_SITE_PRIORITY procedure are described in Table 12 – 113, and the exceptions are listed in Table 12 – 114. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DEFINE_SITE_PRIORITY(
                 gname        IN     VARCHAR2,
                 name         IN     VARCHAR2,
                 comment      IN     VARCHAR2 := NULL)
```

Parameter	Description
gname	The replicated object group for which you are creating a site priority group.
name	The name of the site priority group that you are creating.
comment	This user comment is added to the RepPriority view.

Table 12 – 113 Parameters for DEFINE_SITE_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
duplicateprioritygroup	The given site priority group already exists in the replicated object group.

Table 12 – 114 Exceptions for DEFINE_SITE_PRIORITY

DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN

Purpose

To execute the local outstanding deferred administrative procedures for the given replicated object group at the current master site, or (with assistance from job queues) for all master sites. For additional information, refer to page 7 – 10.

> **Note:** DO_DEFERRED_REPCAT_ADMIN executes only those administrative requests submitted by the connected user that called DO_DEFERRED_REPCAT_ADMIN. Requests submitted by other users are ignored.

Syntax

The parameters for the DO_DEFERRED_REPCAT_ADMIN procedure are described in Table 12 – 115, and the exceptions are listed in Table 12 – 116. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN(
                  gname       IN    VARCHAR2,
                  all_sites   IN    BOOLEAN := FALSE)
```

Parameter	Description
gname	The name of the replicated object group.
all_sites	If ALL_SITES is TRUE, use a job to execute the local administrative procedures at each master.

Table 12 – 115 Parameters for DO_DEFERRED_REPCAT_ADMIN

Exception	Description
nonmaster	The invocation site is not a master site.
commfailure	At least one master site is not accessible and all_sites is TRUE.

Table 12 – 116 Exceptions for DO_DEFERRED_REPCAT_ADMIN

DBMS_REPCAT.DROP_COLUMN_GROUP

Purpose

To drop a column group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 24.

Syntax

The parameters for the DROP_COLUMN_GROUP procedure are described in Table 12 – 117, and the exceptions are listed in Table 12 – 118. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_COLUMN_GROUP(sname         IN     VARCHAR2,
                              oname         IN     VARCHAR2,
                              column_group  IN     VARCHAR2)
```

Parameter	Description
`sname`	The schema in which the replicated table is located.
`oname`	The name of the replicated table whose column group you are dropping.
`column_group`	The name of the column group that you want to drop.

Table 12 – 117 Parameters for DROP_COLUMN_GROUP

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
referenced	The given column group is being used in conflict detection and resolution.
missingobject	The given table does not exist.
missinggroup	The given column group does not exist.
missingschema	The given schema does not exist.

Table 12 – 118 Exceptions for DROP_COLUMN_GROUP

DBMS_REPCAT.DROP_GROUPED_COLUMN

Purpose

To remove members from a column group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 23.

Syntax

The parameters for the DROP_GROUPED_COLUMN procedure are described in Table 12 – 119, and the exceptions are listed in Table 12 – 120. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_GROUPED_COLUMN(
                    sname                     IN    VARCHAR2,
                    oname                     IN    VARCHAR2,
                    column_group              IN    VARCHAR2,
                    list_of_column_names      IN    VARCHAR2)
```

Parameter	Description
`sname`	The schema in which the replicated table is located.
`oname`	The name of the replicated table in which the column group is located.
`column_group`	The name of the column group from which you are removing members.
`list_of_column_names`	The names of the columns that you are removing from the designated column group. This can either be a comma–separated list or a PL/SQL table of column names. The PL/SQL table must be of type dbms_repcat.varchar2s.

Table 12 – 119 Parameters for DROP_GROUPED_COLUMN

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingobject	The given table does not exist.
missinggroup	The given column group does not exist.
missingschema	The given schema does not exist.

Table 12 – 120 Exceptions for DROP_GROUPED_COLUMN

DBMS_REPCAT.DROP_MASTER_REPGROUP

Purpose

To drop a replicated object group from your current site. To drop the replicated object group from all master sites, including the master definition site, you can call this procedure at the master definition site, and set the final argument to TRUE. For additional information, refer to page 4 – 49.

Syntax

The parameters for the DROP_MASTER_REPGROUP procedure are described in Table 12 – 121, and the exceptions are listed in Table 12 – 122. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_MASTER_REPGROUP(
                  gname         IN     VARCHAR2,
                  drop_contents IN     BOOLEAN := FALSE,
                  all_sites     IN     BOOLEAN := FALSE)
```

Parameter	Description
`gname`	The name of the replicated object group that you want to drop from the current master site.
`drop_contents`	By default, when you drop the object group at a master site, all of the objects remain in the schema. They simply are no longer replicated; that is, the replicated objects in the object group no longer send changes to, or receive changes from, other master sites. If you set this argument to TRUE, any replicated objects in the replicated object group are dropped from their associated schemas.
`all_sites`	If ALL_SITES is TRUE and the invocation site is the master definition site, the procedure synchronously multicasts the request to all masters. In this case, execution is immediate at the master definition site and may be deferred at all other master sites.

Table 12 – 121 Parameters for DROP_MASTER_REPGROUP

Exception	Description
nonmaster	The invocation site is not a master site.
nonmasterdef	The invocation site is not the master definition site and ALL_SITES is TRUE.
commfailure	At least one master site is not accessible and ALL_SITES is TRUE.
fullqueue	The deferred RPC queue has entries for the repschema

Table 12 – 122 Exceptions for DROP_MASTER_REPGROUP

DBMS_REPCAT.DROP_MASTER_REPOBJECT

Purpose

To drop a replicated object from a replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 4 – 46.

Syntax

The parameters for the DROP_MASTER_REPOBJECT procedure are described in Table 12 – 123, and the exceptions are listed in Table 12 – 124. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_MASTER_REPOBJECT(
                  sname         IN     VARCHAR2,
                  oname         IN     VARCHAR2,
                  type          IN     VARCHAR2,
                  drop_objects  IN     BOOLEAN := FALSE)
```

Parameter	Description
`sname`	The name of the schema in which the object is located.
`oname`	The name of the object that you want to remove from the replicated object group.
`type`	The type of object that you want to drop.
`drop_objects`	By default, the object remains in the schema, but is dropped from the replicated object group; that is, any changes to the object are no longer replicated to other master and snapshot sites. To completely remove the object from all master sites in the replicated environment, set this argument to TRUE.

Table 12 – 123 Parameters for DROP_MASTER_REPOBJECT

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist.
typefailure	The given type parameter is not supported.
commfailure	At least one master site is not accessible.

Table 12 – 124 Exceptions for DROP_MASTER_REPOBJECT

DBMS_REPCAT.DROP_PRIORITY

Purpose

To drop a member of a priority group by priority level. You must call this procedure from the master definition site. For additional information, refer to page 6 – 32.

Syntax

The parameters for the DROP_PRIORITY procedure are described in Table 12 – 125, and the exceptions are listed in Table 12 – 126. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_PRIORITY( gname        IN   VARCHAR2,
                           pgroup       IN   VARCHAR2,
                           priority_num IN   NUMBER)
```

Parameter	Description
gname	The replicated object group with which the priority group is associated.
pgroup	The name of the priority group containing the member that you want to drop.
priority_num	The priority level of the priority group member that you want to remove from the group.

Table 12 – 125 Parameters for DROP_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingprioritygroup	The given priority group does not exist.

Table 12 – 126 Exceptions for DROP_PRIORITY

DBMS_REPCAT.DROP_PRIORITY_GROUP

Purpose

To drop a priority group for a given replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 33.

Syntax

The parameters for the DROP_PRIORITY_GROUP procedure are described in Table 12 – 127, and the exceptions are listed in Table 12 – 128. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_PRIORITY_GROUP(gname     IN  VARCHAR2,
                                pgroup    IN  VARCHAR2)
```

Parameter	Description
`gname`	The replicated object group with which the priority group is associated.
`pgroup`	The name of the priority group that you want to drop.

Table 12 – 127 Parameters for DROP_PRIORITY_GROUP

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
referenced	The given priority group is being used in conflict resolution.

Table 12 – 128 Exceptions for DROP_PRIORITY_GROUP

DBMS_REPCAT.DROP_PRIORITY_*datatype*

Purpose

To drop a member of a priority group by value. You must call this procedure from the master definition site. The procedure that you must call is determined by the datatype of your "priority" column. The available procedures are listed below:

- DROP_PRIORITY_CHAR
- DROP_PRIORITY_VARCHAR2
- DROP_PRIORITY_NUMBER
- DROP_PRIORITY_DATE
- DROP_PRIORITY_RAW

For additional information, refer to page 6 – 32.

Syntax

The parameters for the DROP_PRIORITY_PROCEDURE_*datatype* procedure are described in Table 12 – 129, and the exceptions are listed in Table 12 – 130. The syntax for the DROP_PRIORITY_VARCHAR2 procedure is shown below. (The syntax for the remaining DROP_PRIORITY_*datatype* procedures is identical, except for the datatype of the value.)

```
DBMS_REPCAT.DROP_PRIORITY_VARCHAR2(gname   IN  VARCHAR2,
                                   pgroup  IN  VARCHAR2,
                                   value   IN  VARCHAR2)
```

Parameter	Description
`gname`	The replicated object group with which the priority group is associated.
`pgroup`	The name of the priority group containing the member that you want to drop.
`value`	The value of the priority group member that you want to remove from the group.

Table 12 – 129 Parameters for DROP_PRIORITY_VARCHAR2

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingprioritygroup	The given priority group does not exist.
paramtype	The value has the incorrect datatype for the priority group.

Table 12 – 130 Exceptions for DROP_PRIORITY_VARCHAR2

DBMS_REPCAT.DROP_SITE_PRIORITY

Purpose

To drop a site priority group for a given replicated object group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 36.

Syntax

The parameters for the DROP_SITE_PRIORITY procedure are described in Table 12 – 131, and the exceptions are listed in Table 12 – 132. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_SITE_PRIORITY( gname   IN  VARCHAR2,
                                name    IN  VARCHAR2)
```

Parameter	Description
gname	The replicated object group with which the site priority group is associated.
name	The name of the site priority group that you want to drop.

Table 12 – 131 Parameters for DROP_SITE_PRIORITY

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
referenced	The given site priority group is being used in conflict resolution.

Table 12 – 132 Exceptions for DROP_SITE_PRIORITY

DBMS_REPCAT.DROP_SITE_PRIORITY_SITE

Purpose

To drop a given site, by name, from a site priority group. You must call this procedure from the master definition site. For additional information, refer to page 6 – 36.

Syntax

The parameters for the DROP_SITE_PRIORITY_SITE procedure are described in Table 12 – 133, and the exceptions are listed in Table 12 – 134. The syntax for this procedure is shown below:

```
drop_site_priority_site(gname   IN  VARCHAR2,
                        name    IN  VARCHAR2
                        site    IN  VARCHAR2)
```

Parameter	Description
gname	The replicated object group with which the site priority group is associated.
name	The name of the site priority group whose member you are dropping.
site	The global database name of the site you are removing from the group.

Table 12 – 133 Parameters for DROP_SITE_PRIORITY_SITE

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingrepgroup	The given replicated object group does not exist.
missingpriority	The given site priority group does not exist.
missingsite	The given site does not exist.

Table 12 – 134 Exceptions for DROP_SITE_PRIORITY_SITE

DBMS_REPCAT.DROP_SNAPSHOT_REPGROUP

Purpose

To drop a snapshot site from your replicated environment. For additional information, refer to page 5 – 10.

Syntax

The parameters for the DROP_SNAPSHOT_REPGROUP procedure are described in Table 12 – 135, and the exceptions are listed in Table 12 – 136. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_SNAPSHOT_REPGROUP(
                    gname          IN      VARCHAR2,
                    drop_contents  IN      BOOLEAN := FALSE)
```

Parameter	Description
`gname`	The name of the replicated object group that you want to drop from the current snapshot site. All objects generated to support replication, such as triggers and packages are dropped.
`drop_contents`	By default, when you drop the replicated object group at a snapshot site, all of the objects remain in their associated schemas; they simply are no longer replicated. If you set this argument to TRUE, any replicated objects in the replicated object group are dropped from their schemas.

Table 12 – 135 Parameters for DROP_SNAPSHOT_REPSCHEMA

Exception	Description
nonsnapshot	The invocation site is not a snapshot site.
missrepgrp	The specified object group does not exist.

Table 12 – 136 Exception for DROP_SNAPSHOT_REPSCHEMA

DBMS_REPCAT.DROP_SNAPSHOT_REPOBJECT

Purpose

To drop a replicated object from a snapshot site. For additional information, refer to page 5 – 9.

Syntax

The parameters for the DROP_SNAPSHOT_REPOBJECT procedure are described in Table 12 – 137, and the exceptions are listed in Table 12 – 138. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_SNAPSHOT_REPOBJECT(
                 sname        IN     VARCHAR2,
                 oname        IN     VARCHAR2,
                 type         IN     VARCHAR2,
                 drop_objects IN     BOOLEAN := FALSE)
```

Parameter	Description
`sname`	The name of the schema in which the object is located.
`oname`	The name of the object that you want to drop from the replicated object group.
`type`	The type of the object that you want to drop.
`drop_objects`	By default, the object remains in its associated schema, but is dropped from its associated object group. To completely remove the object from its schema at the current snapshot site, set this argument to TRUE.

Table 12 – 137 Parameters for DROP_SNAPSHOT_REPOBJECT

Exception	Description
nonsnapshot	The invocation site is not a snapshot site.
missingobject	The given object does not exist.
typefailure	The given type parameter is not supported.

Table 12 – 138 Exceptions for DROP_SNAPSHOT_REPOBJECT

DBMS_REPCAT.DROP_*conflicttype*_RESOLUTION

Purpose

To drop an update, delete, or uniqueness conflict resolution routine. You must call these procedures from the master definition site. The procedure that you must call is determined by the type of conflict that the routine resolves.

Conflict Type	Procedure Name
update	DROP_UPDATE_RESOLUTION
uniqueness	DROP_UNIQUE_RESOLUTION
delete	DROP_DELETE_RESOLUTION

For additional information, refer to page 6 – 27.

Syntax

The parameters for the DROP_UPDATE_RESOLUTION procedure are described in Table 12 – 139, and the exceptions are listed in Table 12 – 140. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_UPDATE_RESOLUTION(
                    sname        IN    VARCHAR2,
                    oname        IN    VARCHAR2,
                    column_group IN    VARCHAR2,
                    sequence_no  IN    NUMBER)
```

The parameters for the DROP_DELETE_RESOLUTION procedure are described in Table 12 – 139, and the exceptions are listed in Table 12 – 140. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_DELETE_RESOLUTION(
                    sname        IN    VARCHAR2,
                    oname        IN    VARCHAR2,
                    sequence_no  IN    NUMBER)
```

The parameters for the DROP_UNIQUE_RESOLUTION procedure are described in Table 12 – 139, and the exceptions are listed in Table 12 – 140. The syntax for this procedure is shown below:

```
DBMS_REPCAT.DROP_UPDATE_RESOLUTION(
                    sname               IN    VARCHAR2,
                    oname               IN    VARCHAR2,
                    constraint_name     IN    VARCHAR2,
                    sequence_no         IN    NUMBER)
```

Parameter	Description
`sname`	The schema in which the table is located.
`oname`	The name of the table for which you want to drop a conflict resolution routine.
`column_group`	The name of the column group for which you want to drop an update conflict resolution routine.
`constraint_name`	The name of the Unique constraint for which you want to drop a unique conflict resolution routine.
`sequence_no`	The sequence number assigned to the conflict resolution method that you want to drop. This number uniquely identifies the routine.

Table 12 – 139 Parameters for DROP_UPDATE/DELETE/UNIQUE_ RESOLUTION

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
missingobject	The given object does not exist as a table in the given schema, or if a conflict resolution routine with the given sequence number is not registered.
missingschema	The given schema does not exist.

Table 12 – 140 Exceptions for DROP_UPDATE/DELETE/UNIQUE_ RESOLUTION

DBMS_REPCAT.EXECUTE_DDL

Purpose

To supply DDL that you want to have executed at each master site. You can call this procedure only from the master definition site. For additional information, refer to page 7 – 14.

Syntax

The parameters for the EXECUTE_DDL procedure are described in Table 12 – 141, and the exceptions are listed in Table 12 – 142. The syntax for this procedure is shown below:

```
DBMS_REPCAT.EXECUTE_DDL(
          gname         IN     VARCHAR2,
          master_list   IN     VARCHAR2 := NULL, |
          master_table  IN     dbms_utility.dblink_array,
          ddl_text      IN     VARCHAR2)
```

Note: If the DDL is supplied without specifying a schema, the default schema is the replication administrator's schema. Be sure to specify the schema if it is other than the replication administrator's schema.

Parameter	Description
`gname`	The name of the replicated object group.
`master_list`	A comma–separated list of master sites at which you want to execute the supplied DDL. There must be no extra whitespace between site names. The default value, NULL, indicates that the DDL should be executed at all sites, including the master definition site.
`master_table`	A table of master sites at which you want to execute the supplied DDL. The first master should be at offset 1, the second at offset 2, and so on.
`ddl_text`	The DDL that you want to have executed at each of the given master sites.

Table 12 – 141 Parameters for EXECUTE_DDL

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
nonmaster	At least one site is not a master site.
ddlfailure	DDL at the master definition site did not succeed.
commfailure	At least one master site is not accessible.

Table 12 – 142 Exceptions for EXECUTE_DDL

DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE

Purpose

To generate the packages needed to support replication for a given table at all master sites. You must call this procedure from the master definition site. For additional information, refer to page 6 – 26.

Syntax

The parameters for the GENERATE_REPLICATION_PACKAGE procedure are described in Table 12 – 143, and the exceptions are listed in Table 12 – 144. The syntax for this procedure is shown below:

```
DBMS_REPCAT.GENERATE_REPLICATION_PACKAGE(
          sname               IN     VARCHAR2,
          oname               IN     VARCHAR2)
```

Parameter	Description
sname	The schema in which the table is located.
oname	The name of the table for which you are generating replication support.

Table 12 – 143 Parameters for GENERATE_REPLICATION_PACKAGE

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist as a table in the given schema awaiting row–level replication information or as a procedure or package (body) awaiting wrapper generation.
commfailure	At least one master site is not accessible.
notcompat	This procedure requires release 7.3 or greater.
notquiesced	The replicated object group was not quiesced.

Table 12 – 144 Exceptions for GENERATE_REPLICATION_PACKAGE

DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT

Purpose

To generate the triggers, packages, and procedures needed to support replication. You must call this procedure from the master definition site. For additional information, refer to page 4 – 21.

Syntax

The parameters for the GENERATE_REPLICATION_SUPPORT procedure are described in Table 12 – 145, and the exceptions are listed in Table 12 – 146. The syntax for this procedure is shown below:

```
DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT(
           sname                IN     VARCHAR2,
           oname                IN     VARCHAR2,
           type                 IN     VARCHAR2,
           package_prefix       IN     VARCHAR2 := NULL,
           procedure_prefix     IN     VARCHAR2 := NULL,
           distributed          IN     BOOLEAN := TRUE,
           gen_rep2_trigger     IN     BOOLEAN := FALSE
           gen_obj_owner        IN     VARCHAR2 := ’’)
```

Parameter	Description
sname	The schema in which the object is located.
oname	The name of the object for which you are generating replication support.
type	The type of the object. The types supported are: TABLE, PACKAGE, and PACKAGE BODY.
package_prefix	For objects of type PACKAGE or PACKAGE BODY this value is prepended to the generated wrapper package name. The default is DEFER_.
procedure_prefix	For objects of type PROCEDURE, PACKAGE or PACKAGE BODY, this value is prepended to the generated wrapper procedure names. By default, no prefix is assigned. The default is DEFER_.
distributed	This parameter must be set to TRUE if your COMPATIBLE parameter is set to 7.3.0 or greater.
gen_rep2_trigger	This parameter is provided for compatibility with previous releases. If you have any pre–release 7.3 snapshot sites, you must set this parameter to TRUE.
gen_obj_owner	The name of the user you want to as as owner of the transaction.

Table 12 – 145 Parameters for GENERATE_REPLICATION_SUPPORT

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist as a table in the given schema awaiting row–level replication information or as a procedure or package (body) awaiting wrapper generation.
typefailure	The given type parameter is not supported.
notquiesced	The replicated object group has not been suspended.
commfailure	At least one master site is not accessible.

Table 12 – 146 Exceptions for GENERATE_REPLICATION_SUPPORT

DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER

Purpose

To generate the triggers and their associated packages needed to support replication for a given object at all master sites, or to generate the triggers and their associated packages needed to support replication for all of the objects in a given object group at a list of master sites. You must call this procedure from the master definition site. The associated object group must be quiesced. For additional information, refer to page 4 – 35.

Syntax

The parameters for the GENERATE_REPLICATION_TRIGGER procedure are described in Table 12 – 147, and the exceptions are listed in Table 12 – 148.

To generate support for an object at all master sites, use the syntax shown below:

```
DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER(
                       sname             IN  VARCHAR2,
                       oname             IN  VARCHAR2,
                       gen_rep2_trigger  IN  BOOLEAN := FALSE)
```

To generate support for an object group at selected master sites, use the syntax shown below:

```
DBMS_REPCAT.GENERATE_REPLICATION_TRIGGER(
                       gname         IN  VARCHAR2,
                       master_list   IN  VARCHAR2 := NULL |
                       master_table  IN  dbms_utility.dblink_array)
```

☞ **Attention:** If you want to generate support for a list of master sites (that is, if you will not be using the default, NULL), you must either use an array or named notation.

Parameter	Description
`sname`	The schema in which the object is located.
`oname`	The name of the object for which you are generating replication support.
`gen_rep2_trigger`	This parameter is provided for compatibility with previous releases. If you have any pre–release 7.3 snapshot sites, you must set this parameter to TRUE.
`gname`	The name of the object group for which you want to generate support.

Table 12 – 147 Parameters for GENERATE_REPLICATION_TRIGGER

Parameter	Description
`master_list`	A comma–separated list of master sites at which you want to generate replication support. By default, support is generated at all master sites.
`master_table`	A PL/SQL table of master sites at which you want to generate replication support.

Table 12 – 147 Parameters for GENERATE_REPLICATION_TRIGGER

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist as a table in the given schema awaiting row–level replication information or as a procedure or package (body) awaiting wrapper generation.
notquiesced	The replicated object group has not been suspended.
commfailure	At least one master site is not accessible.
notcompat	For pre–release 7.3 snapshot sites, you must set GEN_REP2_TRIGGER TRUE.

Table 12 – 148 Exceptions for GENERATE_REPLICATION_TRIGGER

DBMS_REPCAT.MAKE_COLUMN_GROUP

Purpose

To create a new column group with one or more members. You must call this procedure from the master definition site. For additional information, refer to page 6 – 22.

Syntax

The parameters for the MAKE_COLUMN_GROUP procedure are described in Table 12 – 149, and the exceptions are listed in Table 12 – 150. The syntax for this procedure is shown below:

```
DBMS_REPCAT.MAKE_COLUMN_GROUP(
          sname                      IN     VARCHAR2,
          oname                      IN     VARCHAR2,
          column_group               IN     VARCHAR2,
          list_of_column_names       IN     VARCHAR2)
```

Parameter	Description
`sname`	The schema in which the replicated table is located.
`oname`	The name of the replicated table for which you are creating a new column group.
`column_group`	The name that you want assigned to the column group that you are creating.
`list_of_column_names`	The names of the columns that you are grouping. This can either be a comma–separated list or a PL/SQL table of column names. The PL/SQL table must be of type dbms_repcat.varchar2s. Use the single value '*' to create a column group that contains all of the columns in your table.

Table 12 – 149 Parameters for MAKE_COLUMN_GROUP

Exception	Description
nonmasterdef	The invocation site is not the masterdef site.
duplicategroup	The given column group already exists for the table.
missingobject	The given table does not exist.
missingcolumn	The given column does not exist in the designated table.
duplicatecolumn	The given column is already a member of another column group.

Table 12 – 150 Exceptions for MAKE_COLUMN_GROUP

DBMS_REPCAT.PURGE_MASTER_LOG

Purpose

To remove local messages in the RepCatLog associated with a given identification number, source, or replicated object group. For additional information, refer to page 7 – 11.

Syntax

The parameters for the PURGE_MASTER_LOG procedure are described in Table 12 – 151, and the exception is listed in Table 12 – 152. If any parameter is NULL, Oracle treats it as a wildcard. The syntax for this procedure is shown below:

```
DBMS_REPCAT.PURGE_MASTER_LOG( id     IN     NATURAL,
                              source IN     VARCHAR2,
                              gname  IN     VARCHAR2)
```

Parameter	Description
`id`	The identification number of the request, as it appears in the RepCatLog view.
`source`	The master site from which the request originated.
`gname`	The name of the replicated object group for which the request was made.

Table 12 – 151 Parameters for PURGE_MASTER_LOG

Exception	Description
nonmaster	GNAME is not NULL and the invocation site is not a master site.

Table 12 – 152 Exception for PURGE_MASTER_LOG

DBMS_REPCAT.PURGE_STATISTICS

Purpose

To remove information from the RepResolution_Statistics view. For additional information, refer to page 7 – 9.

Syntax

The parameters for the PURGE_STATISTICS procedure are described in Table 12 – 153, and the exceptions are listed in Table 12 – 154. The syntax for this procedure is shown below:

```
DBMS_REPCAT.PURGE_STATISTICS( sname       IN   VARCHAR2,
                              oname       IN   VARCHAR2,
                              start_date  IN   DATE,
                              end_date    IN   DATE)
```

Parameter	Description
`sname`	The name of the schema in which the replicated table is located.
`oname`	The name of the table whose conflict resolution statistics you want to purge.
`start_date/ end_date`	The range of dates for which you want to purge statistics. If START_DATE is NULL, purge all statistics up to the END_DATE. If END_DATE is NULL, purge all statistics after the START_DATE.

Table 12 – 153 Parameters for PURGE_STATISTICS

Exception	Description
missingschema	The given schema does not exist.
missingobject	The given table does not exist.

Table 12 – 154 Exceptions for PURGE_STATISTICS

DBMS_REPCAT.REFRESH_SNAPSHOT_REPGROUP

Purpose

To refresh a snapshot site object group with the most recent data from its associated master site. For additional information, refer to page 5 – 20.

Syntax

The parameters for the REFRESH_SNAPSHOT_REPGROUP procedure are described in Table 12 – 155, and the exceptions are listed in Table 12 – 156. The syntax for this procedure is shown below:

```
DBMS_REPCAT.REFRESH_SNAPSHOT_REPGROUP(
       gname                  IN     VARCHAR2,
       drop_missing_contents  IN     BOOLEAN := FALSE,
       refresh_snapshots      IN     BOOLEAN := FALSE,
       refresh_other_objects  IN     BOOLEAN := FALSE
       execute_as_user        IN     BOOLEAN := FALSE)
```

Parameter	Description
`gname`	The name of the replicated object group.
`drop_missing_contents`	If an object was dropped from the replicated object group, it is not automatically dropped from the schema at the snapshot site. It is simply no longer replicated; that is, changes to this object are no longer sent to its associated master site. Snapshots can continue to be refreshed from their associated master tables; however, any changes to an updatable snapshot will be lost. When an object is dropped from the object group, you can choose to have it dropped from the schema entirely by setting this argument to TRUE.
`refresh_snapshots`	Set this parameter to TRUE to refresh the contents of the snapshots in the replicated object group.
`refresh_other_objects`	Set this parameter to TRUE to refresh the contents of the non–snapshot objects in the replicated object group.
`execute_as_user`	The default, FALSE, indicates that a deferred call is authenticated at the remote system using the authentication context of the user who originally queued the deferred call (as indicated in the ORIGIN_USER column of the DefTran view). Set this to TRUE if you want the execution of a deferred call to be authenticated at the remote system using the authentication context of the session user.

Table 12 – 155 Parameters for REFRESH_SNAPSHOT_REPGROUP

Exception	Description
nonsnapshot	The invocation site is not a snapshot site.
nonmaster	The master is no longer a master site.
commfailure	The master is not accessible.

Table 12 – 156 Exceptions for REFRESH_SNAPSHOT_REPGROUP

DBMS_REPCAT.REGISTER_STATISTICS

Purpose

To collect information about the successful resolution of update, delete and uniqueness conflicts for a table. For additional information, refer to page 7 – 8.

Syntax

The parameters for the REGISTER_STATISTICS procedure are described in Table 12 – 157, and the exceptions are listed in Table 12 – 158. The syntax for this procedure is shown below:

```
DBMS_REPCAT.REGISTER_STATISTICS(     sname  IN     VARCHAR2,
                                     oname  IN     VARCHAR2)
```

Parameter	Description
sname	The name of the schema in which the table is located.
oname	The name of the table for which you want to gather conflict resolution statistics.

Table 12 – 157 Parameters for REGISTER_STATISTICS

Exception	Description
missingschema	The given schema does not exist.
missingobject	The given table does not exist.

Table 12 – 158 Exceptions for REGISTER_STATISTICS

DBMS_REPCAT.RELOCATE_MASTERDEF

Purpose

To change your master definition site to another master site in your replicated environment. For additional information, refer to page 7 – 7.

Syntax

The parameters for the RELOCATE_MASTERDEF procedure are described in Table 12 – 159, and the exceptions are listed in Table 12 – 160. The syntax for this procedure is shown below:

```
DBMS_REPCAT.RELOCATE_MASTERDEF(
          gname                       IN     VARCHAR2,
          old_masterdef               IN     VARCHAR2,
          new_masterdef               IN     VARCHAR2,
          notify_masters              IN     BOOLEAN := TRUE,
          include_old_masterdef       IN     BOOLEAN := TRUE)
```

Parameter	Description
`gname`	The name of the object group whose master definition you want to relocate.
`old_masterdef`	The fully qualified database name of the current master definition site.
`new_masterdef`	The fully qualified database name of the existing master site that you want to make the new master definition site.
`notify_masters`	If NOTIFY_MASTERS is TRUE, the procedure synchronously multicasts the change to all masters (including OLD_MASTERDEF only if INCLUDE_OLD_MASTERDEF is TRUE). If any master does not make the change, rollback the changes at all masters.
`include_old_ masterdef`	If NOTIFY_MASTERS is TRUE and INCLUDE_OLD_MASTERDEF is also TRUE, the old master definition site is also notified of the change.

Table 12 – 159 Parameters for RELOCATE_MASTERDEF

Exception	Description
nonmaster	NEW_MASTERDEF is not a master site or the invocation site is not a master site.
nonmasterdef	OLD_MASTERDEF is not the master definition site.
commfailure	At least one master site is not accessible and NOTIFY_MASTERS is TRUE.

Table 12 – 160 Exceptions for RELOCATE_MASTERDEF

Usage Notes

It is not necessary for either the old or new master definition site to be available when you call RELOCATE_MASTERDEF. In a planned reconfiguration, you should invoke RELOCATE_MASTERDEF with NOTIFY_MASTERS TRUE and INCLUDE_OLD_MASTERDEF TRUE. If just the master definition site fails, you should invoke RELOCATE_MASTERDEF with NOTIFY_MASTERS TRUE and INCLUDE_OLD_MASTERDEF FALSE. If several master sites and the master definition site fail, the administrator should invoke RELOCATE_MASTERDEF at each operational master with NOTIFY_MASTERS FALSE.

DBMS_REPCAT.REMOVE_MASTER_DATABASES

Purpose

To remove one or more master databases from a replicated environment. This procedure regenerates the triggers and their associated packages at the remaining master sites. You must call this procedure from the master definition site. For additional information, refer to page 4 – 47.

Syntax

The parameters for the REMOVE_MASTER_DATABASES procedure are described in Table 12 – 161, and the exceptions are listed in Table 12 – 162. The syntax for this procedure is shown below:

```
DBMS_REPCAT.REMOVE_MASTER_DATABASES(
                gname        IN  VARCHAR2,
                master_list  IN  VARCHAR2 |
                master_table IN  DBMS_UTILITY.DBLINK_ARRAY)
```

Parameter	Description
`gname`	The name of the object group associated with the replicated environment. This prevents confusion if a master database is involved in more than one replicated environment.
`master_list`	A comma–separated list of fully qualified master database names that you want to remove from the replicated environment. There must be no extra whitespace between names in the list.
`master_table`	In place of a list, you may also specify the database names in a PL/SQL table of type DBMS_UTILITY.DBLINK_ARRAY.

Table 12 – 161 Parameters for REMOVE_MASTER_DATABASES

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
nonmaster	At least one of the given databases is not a master site.
reconfigerror	One of the given databases is the master definition site.
commfailure	At least one remaining master site is not accessible.

Table 12 – 162 Exceptions for REMOVE_MASTER_DATABASES

DBMS_REPCAT.REPCAT_IMPORT_CHECK

Purpose

To ensure that the objects in the replicated object group have the appropriate object identifiers and status values after you perform an export/import of a replicated object or an object used by the symmetric replication facility. For additional information, refer to page 7 – 14.

Syntax

The parameters for the REPCAT_IMPORT_CHECK procedure are described in Table 12 – 163, and the exceptions are listed in Table 12 – 164. The syntax for this procedure is shown below:

```
DBMS_REPCAT.REPCAT_IMPORT_CHECK(
          gname  IN     VARCHAR2,
                        master IN   BOOLEAN := TRUE)
```

Parameter	Description
gname	The name of the replicated object group. If you omit both parameters, the procedure checks all replicated object groups at your current site.
master	Set this flag to TRUE if you are checking a master site or FALSE if you are checking a snapshot site.

Table 12 – 163 Parameters for REPCAT_IMPORT_CHECK

Exception	Description
nonmaster	MASTER is TRUE and either the database is not a master site for the schema or the database is not the expected database.
nonsnapshot	MASTER is FALSE and the database is not a snapshot site for the schema.
missingobject	A valid replicated object in the schema does not exist.

Table 12 – 164 Exceptions for REPCAT_IMPORT_CHECK

DBMS_REPCAT.RESUME_MASTER_ACTIVITY

Purpose

To resume normal replication activity after quiescing a replicated environment. For additional information, refer to page 4 – 44.

Syntax

The parameters for the RESUME_MASTER_ACTIVITY procedure are described in Table 12 – 165, and the exceptions are listed in Table 12 – 166. The syntax for this procedure is shown below:

```
DBMS_REPCAT.RESUME_MASTER_ACTIVITY(
       gname           IN     VARCHAR2,
       override        IN     BOOLEAN := FALSE)
```

Parameter	Description
gname	The name of the replicated object group.
override	If override is TRUE, it ignores any pending RepCat administration requests and restores normal replication activity at each master as quickly as possible. This should be considered only in emergency situations. If override is FALSE, it restores normal replication activity at each master only when there is no pending RepCat administration request for GNAME at that master.

Table 12 – 165 Parameters for RESUME_MASTER_ACTIVITY

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
notquiesced	The replicated object group is not quiescing or quiesced.
commfailure	At least one master site is not accessible.

Table 12 – 166 Exceptions for RESUME_MASTER_ACTIVITY

DBMS_REPCAT.SET_COLUMNS

Purpose

To use an alternate column or group of columns, instead of the primary key, to determine which columns to compare when using row–level replication. You must call this procedure from the master definition site. For additional information, refer to page 4 – 16.

Syntax

The parameters for the SET_COLUMNS procedure are described in Table 12 – 167, and the exceptions are listed in Table 12 – 168. The syntax for this procedure is shown below:

```
DBMS_REPCAT.SET_COLUMNS(
        sname            IN     VARCHAR2,
        oname            IN     VARCHAR2,
        column_list      IN     VARCHAR2 |
        column_table     IN     DBMS_UTILITY.NAME_ARRAY)
```

Parameter	Description
`sname`	The schema in which the table is located.
`oname`	The name of the table for which you will generate replication support.
`column_list`	A comma–separated list of the columns in the table that you want to use as a “primary key”. There must be no whitespace between entries.
`column_table`	Instead of a list, you can use a PL/SQL table of type DBMS_UTILITY.NAME_ARRAY to contain the column names. The first column name should be at offset 1, the second at offset 2, and so on.

Table 12 – 167 Parameters for SET_COLUMNS

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
missingobject	The given object does not exist as a table in the given schema awaiting row–level replication information.
missingcolumn	At least one column is not in the table

Table 12 – 168 Exceptions for SET_COLUMNS

DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY

Purpose

To suspend replication activity for an object group. You must call this procedure from the master definition site. For additional information, refer to page 4 – 43.

> **Note:** The current implementation of SUSPEND_MASTER_ACTIVITY quiesces all replicated object groups at each master site.

Syntax

The parameter for the SUSPEND_MASTER_ACTIVITY procedure is described in Table 12 – 169, and the exceptions are listed in Table 12 – 170. The syntax for this procedure is shown below:

```
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY(
      gname           IN     VARCHAR2
      execute_as_user IN     BOOLEAN => FALSE)
```

Parameter	Description
`gname`	The name of the object group for which you want to suspend activity.
`execute_as_user`	The default, FALSE, indicates that a deferred call is authenticated at the remote system using the authentication context of the user who originally queued the deferred call (as indicated in the ORIGIN_USER column of the DefTran view). Set this to TRUE if you want the execution of a deferred call to be authenticated at the remote system using the authentication context of the session user.

Table 12 – 169 Parameter for SUSPEND_MASTER_ACTIVITY

Exception	Description
nonmasterdef	The invocation site is not the master definition site.
notnormal	The replicated object group is not in normal operation.
commfailure	At least one master site is not accessible.
missingobjectgroup	The given object group does not exist.

Table 12 – 170 Exceptions for SUSPEND_MASTER_ACTIVITY

DBMS_REPCAT.SWITCH_SNAPSHOT_MASTER

Purpose

To change the master database of a snapshot replicated object group to another master site. This procedure does a full refresh of the affected snapshots and regenerates the triggers and their associated packages as needed. This procedure does not push the queue to the old master site before changing masters. For additional information, refer to page 7 – 7.

Syntax

The parameters for the SWITCH_SNAPSHOT_MASTER procedure are described in Table 12 – 171, and the exceptions are listed in Table 12 – 172. The syntax for this procedure is shown below:

```
DBMS_REPCAT.SWITCH_SNAPSHOT_MASTER(
                     gname              IN    VARCHAR2,
                     master             IN    VARCHAR2
                    execute_as_user    IN    BOOLEAN := FALSE)
```

Parameter	Description
`gname`	The name of the snapshot object group for which you want to change master sites.
`master`	The fully qualified database name of the new master database to use for the snapshot site.
`execute_as_user`	The default, FALSE, indicates that a deferred call is authenticated at the remote system using the authentication context of the user who originally queued the deferred call (as indicated in the ORIGIN_USER column of the DefTran view). Set this to TRUE if you want the execution of a deferred call to be authenticated at the remote system using the authentication context of the session user.

Table 12 – 171 Parameters for SWITCH_SNAPSHOT_MASTER

Exception	Description
nonsnapshot	The invocation site is not a snapshot site.
nonmaster	The given database is not a master site.
commfailure	The given database is not accessible.

Table 12 – 172 Exceptions for SWITCH_SNAPSHOT_MASTER

DBMS_REPCAT.WAIT_MASTER_LOG

Purpose

To determine if changes that were asynchronously propagated to a master site have been applied. For additional information, refer to page 7 – 9.

Syntax

The parameters for the WAIT_MASTER_LOG procedure are described in Table 12 – 173, and the exception is listed in Table 12 – 174. The syntax for this procedure is shown below:

```
DBMS_REPCAT.WAIT_MASTER_LOG(
                 gname         IN     VARCHAR2,
                 record_count  IN     NATURAL,
                 timeout       IN     NATURAL,
                 true_count    OUT    NATURAL)
```

Parameter	Description
`gname`	The name of the replicated object group.
`record_count`	The procedure returns whenever the number of incomplete activities is at or below this threshold.
`timeout`	The maximum number of seconds to wait before the procedure returns.
`true_count` (out parameter)	Returns the number of incomplete activities.

Table 12 – 173 Parameters for WAIT_MASTER_LOG

Exception	Description
nonmaster	The invocation site is not a master site.

Table 12 – 174 Exception for WAIT_MASTER_LOG

DBMS_REPCAT_ADMIN.GRANT_ADMIN_ANY_REPGROUP

Purpose

To grant the necessary privileges to the replication administrator to administer any replicated object group at the current site. For additional information, refer to page 4 – 4.

Syntax

The parameter for the GRANT_ADMIN_ANY_REPGROUP procedure is described in Table 12 – 175, and the exception is listed in Table 12 – 176. The syntax for this procedure is shown below:

```
DBMS_REPCAT_ADMIN.GRANT_ADMIN_ANY_REPGROUP(
                                         userid IN VARCHAR2)
```

Parameter	Description
userid	The name of the replication administrator to whom you want to grant the necessary privileges and roles to administer any replicated object groups at the current site.

Table 12 – 175 Parameter for GRANT_ADMIN_ANY_REPGROUP

Exception	Description
ORA–01917	The user does not exist.

Table 12 – 176 Exception for GRANT_ADMIN_ANY_REPGROUP

DBMS_REPCAT_ADMIN.GRANT_ADMIN_REPGROUP

Purpose

To grant the necessary privileges to the replication administrator to administer a schema at the current site. This procedure is most useful if your object group does not span schemas. For additional information, refer to page 4 – 4.

Syntax

The parameter for the GRANT_ADMIN_REPGROUP procedure is described in Table 12 – 177, and the exception is listed in Table 12 – 178. The syntax for this procedure is shown below:

```
DBMS_REPCAT_ADMIN.GRANT_ADMIN_REPGROUP(userid IN VARCHAR2)
```

Parameter	Description
`userid`	The name of the replication administrator. This user is then granted the necessary privileges and roles to administer the schema of the same name within a replicated object group at the current site.

Table 12 – 177 Parameter for GRANT_ADMIN_GROUP

Exception	Description
ORA–01917	The user does not exist.

Table 12 – 178 Exception for GRANT_ADMIN_GROUP

DBMS_REPCAT_ADMIN.REVOKE_ADMIN_ANY_REPGROUP

Purpose

To revoke the privileges and roles from the replication administrator that would be granted by GRANT_ADMIN_ANY_REPGROUP. For additional information, refer to page 4 – 4.

☞ **Attention:** Identical privileges and roles that were granted independently of GRANT_ADMIN_ANY_REPGROUP are also revoked.

Syntax

The parameter for the REVOKE_ADMIN_ANY_REPGROUP procedure is described in Table 12 – 179, and the exception is listed in Table 12 – 180. The syntax for this procedure is shown below:

```
DBMS_REPCAT_ADMIN.REVOKE_ADMIN_ANY_REPGROUP(
                                       userid IN VARCHAR2)
```

Parameter	Description
`userid`	The name of the replication administrator whose privileges you want to revoke.

Table 12 – 179 Parameter for REVOKE_ADMIN_ANY_REPGROUP

Exception	Description
ORA–01917	The user does not exist.

Table 12 – 180 Exception for REVOKE_ADMIN_ANY_REPGROUP

DBMS_REPCAT_ADMIN.REVOKE_ADMIN_REPGROUP

Purpose

To revoke the privileges and roles from the replication administrator that would be granted by GRANT_ADMIN_REPGROUP. For additional information, refer to page 4 – 4.

☞ **Attention:** Identical privileges and roles that were granted independently of GRANT_ADMIN_REPGROUP are also revoked.

Syntax

The parameter for the REVOKE_ADMIN_REPGROUP procedure is described in Table 12 – 181, and the exception is listed in Table 12 – 182. The syntax for this procedure is shown below:

```
DBMS_REPCAT_ADMIN.REVOKE_ADMIN_REPGROUP(userid IN VARCHAR2)
```

Parameter	Description
`userid`	The name of the replication administrator whose privileges you want to revoke.

Table 12 – 181 Parameter for REVOKE_ADMIN_REPGROUP

Exception	Description
ORA–01917	The user does not exist.

Table 12 – 182 Exception for REVOKE_ADMIN_ANY_REPGROUP

DBMS_REPCAT_AUTH.GRANT_SURROGATE_REPCAT

Purpose

To grant the privileges needed by the symmetric replication facility to a user. For additional information, refer to page 4 – 7.

Syntax

The parameter for the GRANT_SURROGATE_REPCAT procedure is described in Table 12 – 183, and the exception is listed in Table 12 – 184. The syntax for this procedure is shown below:

```
DBMS_REPCAT_AUTH.GRANT_SURROGATE_REPCAT(userid IN VARCHAR2)
```

Parameter	Description
`userid`	The name of the user to whom you wish to grant the necessary privileges.

Table 12 – 183 Parameter for GRANT_SURROGATE_REPCAT

Exception	Description
ORA–01917	The user does not exist

Table 12 – 184 Exception for GRANT_SURROGATE_REPCAT

DBMS_REPCAT_AUTH.REVOKE_SURROGATE_REPCAT

Purpose

To revoke the privileges granted to the surrogate repcat user. For additional information, refer to page 4 – 7.

Syntax

The parameters for the REVOKE_SURROGATE_REPCAT procedure are described in Table 12 – 185, and the exceptions are listed in Table 12 – 186. The syntax for this procedure is shown below:

```
DBMS_REPCAT_AUTH.REVOKE_SURROGATE_REPCAT(userid IN VARCHAR2)
```

Parameter	Description
`userid`	The name of the user from whom you wish to revoke the necessary privileges.

Table 12 – 185 Parameters for REVOKE_SURROGATE_REPCAT

Exception	Description
ORA–01917	The user does not exist

Table 12 – 186 Exception for REVOKE_SURROGATE_REPCAT

DBMS_REPUTIL.REPLICATION_OFF

Purpose

To modify tables without replicating the modifications to any other sites in the replicated environment, or to disable row–level replication when using procedural replication. You should generally quiesce your replicated environment before setting this flag. For additional information, refer to page 8 – 18.

Syntax

The syntax for the REPLICATION_OFF procedure is shown below. This procedure takes no arguments.

```
DBMS_REPUTIL.REPLICATION_OFF
```

DBMS_REPUTIL.REPLICATION_ON

Purpose

To re–enable the replication of changes after it has been temporarily suspended by calling REPLICATION_OFF. For additional information, refer to page 8 – 19.

Syntax

The syntax for the REPLICATION_ON procedure is shown below. This procedure takes no arguments.

```
DBMS_REPUTIL.REPLICATION_ON
```

DBMS_SNAPSHOT.I_AM_A_REFRESH

Purpose

To return the value of the I_AM_REFRESH package state. For additional information, refer to page 8 – 20.

Syntax

The I_AM_A_REFRESH function takes no arguments. A return value of TRUE indicates that all local replication triggers for snapshots will be effectively disabled in this session because each replication trigger first checks this state. A return value of FALSE indicates that these triggers are enabled. The syntax for this procedure is shown below.

```
DBMS_SNAPSHOT.I_AM_A_REFRESH RETURN BOOLEAN
```

DBMS_SNAPSHOT.PURGE_LOG

Purpose

To purge rows from the snapshot log. For additional information, refer to page 3 – 12.

Syntax

The parameters for the PURGE_LOG procedure are described in Table 12 – 187. The syntax for this procedure is shown below:

```
DBMS_SNAPSHOT.PURGE_LOG(
                master IN     VARCHAR2,
                num    IN     BINARY_INTEGER DEFAULT 1
                flag   IN     VARCHAR2 DEFAULT 'NOP')
```

parameter	description
`master`	Name of the master table.
`num`	Number of least recently refreshed snapshots whose rows you want to remove from snapshot log. For example, the following statement deletes rows needed to refresh the two least recently refreshed snapshots: `EXECUTE dbms_snapshot.purge_log('master_table', 2);` To delete all rows in the snapshot log, indicate a high number of snapshots to disregard, as in this example: `EXECUTE dbms_snapshot.purge_log('master_table', 9999);` This statement completely purges the snapshot log that corresponds to MASTER_TABLE if fewer than 9999 snapshots are based on MASTER_TABLE. A simple snapshot whose rows have been purged from the snapshot log must be completely refreshed the next time it is refreshed.
`flag`	Specify DELETE to guarantee that rows are deleted from the snapshot log for at least one snapshot. This argument can override the setting for the argument NUM. For example, the following statement deletes rows from the least recently refreshed snapshot that actually has dependent rows in the snapshot log: `EXECUTE dbms_snapshot.purge_log('master_table', 0, 'DELETE');`

Table 12 – 187 Parameters for DBMS_SNAPSHOT.PURGE_LOG

DBMS_SNAPSHOT.REFRESH

Purpose

To consistently refresh one or more snapshots that are not members of the same refresh group. For additional information, refer to page 3 – 20.

Syntax

The parameters for the REFRESH procedure are described in Table 12 – 188. The syntax for this procedure is shown below:

```
DBMS_SNAPSHOT.REFRESH(
          list                IN     VARCHAR2,
          method              IN     VARCHAR2 DEFAULT NULL,
          rollback_seg        IN     VARCHAR2 DEFAULT NULL,
          push_deferred_rpc   IN     BOOLEAN DEFAULT TRUE,
          refresh_after_errorsIN     BOOLEAN DEFAULT FALSE)
```

parameter	description
`list`	Comma–separated list of snapshots that you want to refresh. (Synonyms are not supported.) These snapshots can be located in different schemas and have different master tables; however, all of the listed snapshots must be in your current database. Alternatively, you may pass in a PL/SQL table of type DBMS_UTILITY.UNCL_ARRAY, where each element is the name of a snapshot.
`method`	Type of refresh to perform for each snapshot listed; ‘F’ or ‘f’ indicates a fast refresh, ‘C’ or ‘c’ indicates a complete refresh, and ‘?’ indicates a default refresh. If you specified a refresh mode when you created the snapshot, that mode is used when you specify a default refresh. If no mode was specified, Oracle performs a fast refresh if possible; otherwise, it performs a complete refresh. If the METHOD list contains fewer elements than the snapshot LIST, the trailing elements in the snapshot list are refreshed using a default refresh. For example, the following EXECUTE statement within SQL*Plus: `EXECUTE dbms_snapshot.refresh(’emp, dept, scott.salary’, ’CF’);` performs a complete refresh of the EMP snapshot, a fast refresh of the DEPT snapshot, and a default refresh of the SCOTT.SALARY snapshot.
`rollback_seg`	Name of the snapshot site rollback segment to use while refreshing snapshots. When you call REFRESH, all of the listed snapshots are updated to a single point in time. If the refresh fails for any of the snapshots, none of the snapshots are updated.

Table 12 – 188 Parameters for DBMS_SNAPSHOT.REFRESH, continued on next page

parameter	description
push_deferred_rpc	Used by updatable snapshots only. Use the default value, TRUE, if you want to push changes from the snapshot to its associated master before refreshing the snapshot. Otherwise, these changes may appear to be temporarily lost.
refresh_after_errors	Used by updatable snapshots only. Set this parameter to TRUE if you want the refresh to proceed even if there are outstanding conflicts logged in the DefError view for the snapshot's master.

Table 12 – 188 Parameters for DBMS_SNAPSHOT.REFRESH

DBMS_SNAPSHOT.SET_I_AM_A_REFRESH

Purpose

To set the I_AM_REFRESH package state to the appropriate value. For additional information, refer to page 8 – 20.

Syntax

The parameter for the SET_I_AM_A_REFRESH procedure is described in Table 12 – 189. The syntax for this procedure is shown below.

```
DBMS_SNAPSHOT.SET_I_AM_A_REFRESH(value IN BOOLEAN)
```

parameter	description
value	Value that you want to set the I_AM_A_REFRESH package state to. If this state is set to TRUE, all local replication triggers for snapshots will be effectively disabled in this session because each replication trigger first checks this state. If this state is set to FALSE, these triggers will be enabled.

Table 12 – 189 Parameters for DBMS_SNAPSHOT.SET_I_AM_A_REFRESH

Package Variables

Table 12 – 190 describes the package variables that are used by the symmetric replication facility. You may need to check the value of one or more of these variables in your own packages or triggers.

variable	type	description
schema.rep$what_am_i.i_am_a_snapshot	BOOLEAN	TRUE indicates that the local site is a snapshot site for the replicated object group. If the local site is a master site for the replicated object group, this variable is FALSE.
dbms_reputil.replication_is_on	BOOLEAN	TRUE indicates that the generated replication triggers are enabled. FALSE indicates that replication is disabled at the current site for the replicated object group. This variable is set by calling the REPLICATION_ON or REPLICATION_OFF procedures in the DBMS_REPUTIL package.
dbms_reputil.from_remote	BOOLEAN	This variable is set to TRUE at the beginning of procedures in the $RP replication packages, and is set to FALSE at the end of these procedures. You may need to check this variable if you have any triggers that could be fired as the result of an update by a $RP package.
dbms_reputil.global_name	VARCHAR2(128)	This variable contains the global database name of the local database.

Table 12 – 190 Replication Package Variables

CHAPTER

13

Data Dictionary Views

This chapter describes the data dictionary views that might be useful to users of the symmetric replication facility. The views are alphabetized within the following general groupings:

- replication catalog views
- deferred transaction views
- job queue views
- snapshot and snapshot refresh group views

Replication Catalog Views

Whenever you install symmetric replication capabilities at a site, Oracle installs the replication catalog, which consists of tables and views, at that site. As shown in Table 13 – 1, the views are used by master and snapshot sites to determine such information as what objects are being replicated, where they are being replicated, and if any errors have occurred during replication. *You should not modify the replication catalog tables directly; use the procedures provided in the DBMS_REPCAT package.*

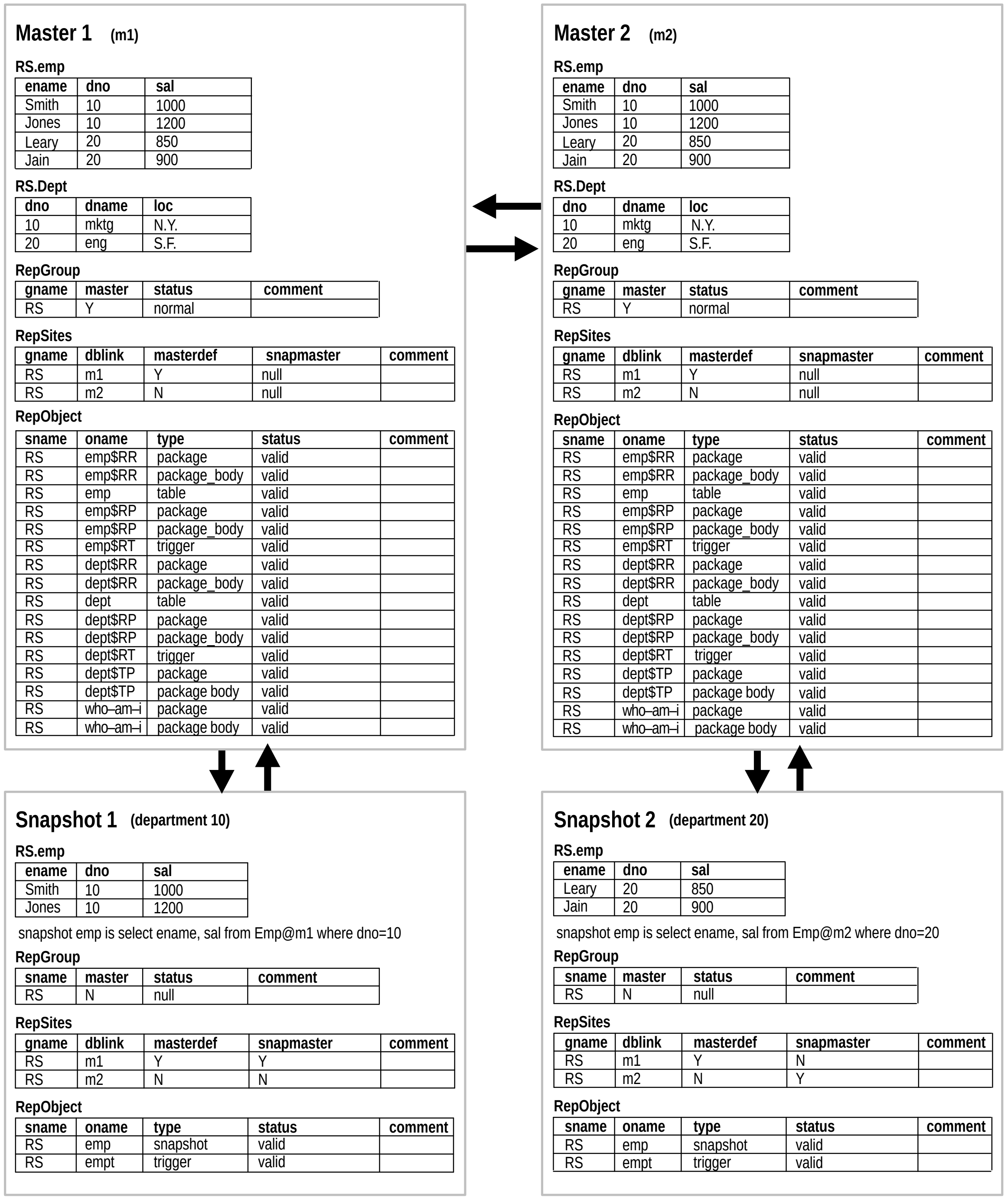
Each view has three versions, which have different prefixes: USER_*, ALL_*, and SYS.DBA_*. This section ignores any differences between these views.

RepGroup View

The RepGroup view lists all of the object groups that are being replicated. The members of each object group are listed in a different view, RepObject.

Column	Description
sname	The name of the replicated schema. Obsolete with release 7.3 or later.
gname	The name of the replicated object group.
master	'Y' indicates that this is a master site. 'N' indicates the current site is a snapshot site.
status	Used at master sites only. Status can be: normal, quiescing, or quiesced.
schema_comment	Any user–supplied comments.

Table 13 – 1 RepGroup View



Master 1 (m1)

RS.emp

ename	dno	sal
Smith	10	1000
Jones	10	1200
Leary	20	850
Jain	20	900

RS.Dept

dno	dname	loc
10	mktg	N.Y.
20	eng	S.F.

RepGroup

gname	master	status	comment
RS	Y	normal	

RepSites

gname	dblink	masterdef	snapmaster	comment
RS	m1	Y	null	
RS	m2	N	null	

RepObject

sname	oname	type	status	comment
RS	emp$RR	package	valid	
RS	emp$RR	package_body	valid	
RS	emp	table	valid	
RS	emp$RP	package	valid	
RS	emp$RP	package_body	valid	
RS	emp$RT	trigger	valid	
RS	dept$RR	package	valid	
RS	dept$RR	package_body	valid	
RS	dept	table	valid	
RS	dept$RP	package	valid	
RS	dept$RP	package_body	valid	
RS	dept$RT	trigger	valid	
RS	dept$TP	package	valid	
RS	dept$TP	package body	valid	
RS	who–am–i	package	valid	
RS	who–am–i	package body	valid	

Master 2 (m2)

RS.emp

ename	dno	sal
Smith	10	1000
Jones	10	1200
Leary	20	850
Jain	20	900

RS.Dept

dno	dname	loc
10	mktg	N.Y.
20	eng	S.F.

RepGroup

gname	master	status	comment
RS	Y	normal	

RepSites

gname	dblink	masterdef	snapmaster	comment
RS	m1	Y	null	
RS	m2	N	null	

RepObject

sname	oname	type	status	comment
RS	emp$RR	package	valid	
RS	emp$RR	package_body	valid	
RS	emp	table	valid	
RS	emp$RP	package	valid	
RS	emp$RP	package_body	valid	
RS	emp$RT	trigger	valid	
RS	dept$RR	package	valid	
RS	dept$RR	package_body	valid	
RS	dept	table	valid	
RS	dept$RP	package	valid	
RS	dept$RP	package_body	valid	
RS	dept$RT	trigger	valid	
RS	dept$TP	package	valid	
RS	dept$TP	package body	valid	
RS	who–am–i	package	valid	
RS	who–am–i	package body	valid	

Snapshot 1 (department 10)

RS.emp

ename	dno	sal
Smith	10	1000
Jones	10	1200

snapshot emp is select ename, sal from Emp@m1 where dno=10

RepGroup

sname	master	status	comment
RS	N	null	

RepSites

gname	dblink	masterdef	snapmaster	comment
RS	m1	Y	Y	
RS	m2	N	N	

RepObject

sname	oname	type	status	comment
RS	emp	snapshot	valid	
RS	empt	trigger	valid	

Snapshot 2 (department 20)

RS.emp

ename	dno	sal
Leary	20	850
Jain	20	900

snapshot emp is select ename, sal from Emp@m2 where dno=20

RepGroup

sname	master	status	comment
RS	N	null	

RepSites

gname	dblink	masterdef	snapmaster	comment
RS	m1	Y	N	
RS	m2	N	Y	

RepObject

sname	oname	type	status	comment
RS	emp	snapshot	valid	
RS	empt	trigger	valid	

Figure 13 – 1 Replication Catalog Views

RepCatLog View

The RepCatLog at each master site contains the interim status of any asynchronous administrative requests and any error messages generated. All messages encountered while executing a request are eventually transferred to the RepCatLog at the master that originated the request. If an administrative request completes without error, ultimately all traces of this request are removed from the RepCatLog view.

Column	Description
id	A sequence number. Together, the ID and SOURCE columns identify all log records at all master sites that pertain to a single administrative request.
source	Location that the request originated.
userid	Userid of person making the request.
timestamp	When the request was made.
role	Indicates if site is the ’masterdef’ or a ’master’ site.
master	If the role is ’masterdef’ and the task is remote, indicates which master is performing the task.
sname	The name of the schema for the replicated object, if applicable.
request	The name of the DBMS_REPCAT administrative procedure that was run.
oname	The name of the replicated object, if applicable.
type	The type of replicated object.
status	The status of the administrative request: ready, do_callback, await_callback, or error.
message	Any error message that has been returned.
errnum	The Oracle error number for the message.
gname	The name of the replicated object group.

Table 13 – 2 RepCatLog View

RepColumn_Group View

The RepColumn_Group view lists all of the column groups that you have defined for each replicated table.

Column	Description
sname	The name of the schema containing the replicated table.
oname	The name of the replicated table.
group_name	The column group name.
group_comment	Any user–supplied comments.

Table 13 – 3 RepColumn_Group View

RepConflict View

The RepConflict view displays the name of the table for which you have defined a conflict resolution method and the type of conflict that the method is used to resolve.

Column	Description
sname	The name of the schema containing the replicated table.
oname	The name of the table for which you have defined a conflict resolution method.
conflict_type	The type of conflict that the conflict resolution method is used to resolve: delete, uniqueness, or update.
reference_name	The object to which the routine applies. For delete conflicts, this is the table name. For uniqueness conflicts, this is the constraint name. For update conflicts, this is the column group name.

Table 13 – 4 RepConflict view

RepDDL

The RepDDL holds DDL for replication objects.

Column	Description
log_id	Identifying number of the RepCat log record.
source	Name of the database at which the request originated.
role	’Y’ if this database is the masterdef for the request; ’N’ if this database is a master.
master	Name of the database that processes this request.
line	Ordering of records within a single request.
text	Portion of an argument or DDL text.

Table 13 – 5 RepDDL View

RepGenerated

The RepGenerated view lists information about system–generated objects.

Column	Description
sname	Owner of the object.
oname	Object name.
type	Object type.
base_sname	Owner of the base object.
base_oname	Object name of the base object.
base_type	Object type of the base object.
package_prefix	Prefix for package wrapper name.
procedure_prefix	prefix for procedure wrapper for procedures within a package wrapper.
distributed	’Y’ if generation is distributed; ’N’ if generated objects are cloned. Should always be ’Y’ for Rep3 environments.
reason	The reason why this object was generated.

Table 13 – 6 RepDDL View

RepGrouped_Column View

The RepGrouped_Column view lists all of the columns that make up the column groups for each table.

Column	Description
sname	The name of the schema containing the replicated table.
oname	The name of the replicated table.
group_name	The name of the column group.
column_name	The name of the column in the column group.

Table 13 – 7 RepGrouped_Column View

RepKey Columns View

The RepKey_Columns view lists information relating to the primary key column.

Column	Description
sname	Owner of the replicated table.
oname	Name of the replicated table.
col	”Primary Key” column name in the table.

Table 13 – 8 RepKey Columns View

RepSite View

The RepSite view lists the members of each replicated object group.

Column	Description
gname	The name of the replicated object group.
dblink	The database link to the master site for this object group.
masterdef	Indicates which of the dblinks is the master definition site.
snapmaster	Used by snapshot sites to indicate which of the dblinks to use when refreshing.
master_comment	User–supplied comments.

Table 13 – 9 RepSite View

RepObject View

The RepObject view provides information about the objects in each replicated object group. An object can belong to only one object group. A replicated object group can span multiple schemas.

Column	Description
sname	The name of the schema containing the replicated object.
oname	The name of the replicated object.
type	The type of replicated object: table, view, package, package body, procedure, function, index, synonym, trigger, or snapshot.
status	CREATE indicates that Oracle is applying user supplied or Oracle–generated DDL to the local database in an attempt to create the object locally. When a local replica exists, Oracle COMPAREs the replica with the master definition to ensure that they are consistent. When creation or comparison complete successfully, Oracle updates the status to VALID; otherwise, it updates the status to ERROR. If you drop an object, Oracle updates its status to DROPPED before deleting the row from the RepObject view.
id	The identifier of the local database object, if one exists.
object_comment	Any user supplied comments.
gname	The name of the replicated object group to which the object belongs.

Table 13 – 10 RepObject View

RepParameter_Column View

In addition to the information contained in the RepResolution view, the RepParameter_Column view also contains information about the columns that you indicated should be used to resolve the conflict. These are the column values that are passed as the LIST_OF_COLUMN_NAMES argument to the ADD_*_RESOLUTION procedures in the DBMS_REPCAT package.

Column	Description
sname	The name of the schema containing the replicated table.
oname	The name of the replicated table.
conflict_type	The type of conflict that the routine is used to resolve: delete, uniqueness, or update.
reference_name	The object to which the routine applies. For delete conflicts, this is the table name. For uniqueness conflicts, this is the constraint name. For update conflicts, this is the column group name.
sequence_no	The order that resolution methods are applied, with 1 being applied first.
method_name	The name of an Oracle–supplied conflict resolution method. For user–supplied methods, this value is ’user function’.
function_name	For methods of type ’user function’, the name of the user–supplied conflict resolution routine.
priority_group	For methods of type ’priority group’, the name of the priority group.
parameter_table_ name	Defaults to object name of PL/SQL table containing columns passed to conflict resolution function.
parameter_ column_name	The name of the column used as the IN parameter for the conflict resolution routine.
parameter_ sequence_no	Ordering of column used as IN parameter.

Table 13 – 11 RepParameter_Column View

RepPriority View

The RepPriority view displays the value and priority level of each priority group member. Priority group names must be unique within a replicated object group. Priority levels and values must each be unique within a given priority group.

Column	Description
sname	The name of the replicated schema. Obsolete with release 7.3 or later.
gname	The name of the replicated object group.
priority_group	The name of the priority group or site priority group.
priority	The priority level of the member. The highest number has the highest priority.
data_type	The datatype of the values in the priority group.
fixed_data_length	The maximum length of values of datatype CHAR.
char_value	The value of the priority group member, if data_type = char.
varchar2_value	The value of the priority group member, if data_type = varchar2.
number_value	The value of the priority group member, if data_type = number.
date_value	The value of the priority group member, if data_type = date.
raw_value	The value of the priority group member, if data_type = raw.

Table 13 – 12 RepPriority View

RepPriority_Group View

The RepPriority_Group view lists the priority and site priority groups that you have defined for a replicated object group.

Column	Description
sname	The name of the replicated schema. Obsolete with release 7.3 or later. Not shown in USER views.
gname	The name of the replicated object group. Not shown in USER views.
priority_group	The name of the priority group or site priority group.
data_type	The datatype of the values in the priority group.
fixed_data_length	The maximum length for values of datatype CHAR.
priority_comment	Any user–supplied comments.

Table 13 – 13 RepPriority_Group View

RepProp View

The RepProp view indicates the technique used to propagate operations on an object to the same object at another master site. These operations may have resulted from a call to a stored procedure or procedure wrapper, or may have been issued against a table directly.

Column	Description
sname	The name of the schema containing the replicated object.
oname	The name of the replicated object.
type	The type of object being replicated.
dblink	The fully qualified database name of the master site to which changes are being propagated.
how	How propagation is performed. Values recognized are ’none’ for the local master site, and ’synchronous’ or ’asynchronous’ for all others.
propagate_ comment	Any user–supplied comments.

Table 13 – 14 RepProp View

RepResolution View

The RepResolution view indicates the routines used to resolve update, unique or delete conflicts for each table replicated using row–level replication for a given schema.

Column	Description
sname	The name of the replicated schema.
oname	The name of the replicated table.
conflict_type	The type of conflict that the routine is used to resolve: delete, uniqueness, or update.
reference_name	The object to which the routine applies. For delete conflicts, this is the table name. For uniqueness conflicts, this is the constraint name. For update conflicts, this is the column group name.
sequence_no	The order that resolution methods are applied, with 1 being applied first.
method_name	The name of an Oracle–supplied conflict resolution method. For user–supplied methods, this value is ’user function’.
function_name	For methods of type ’user function’, the name of the user–supplied conflict resolution routine.

Table 13 – 15 RepResolution View, continued on next page

Column	Description
priority_group	For methods of type ’priority group’, the name of the priority group.
resolution_ comment	Any user–supplied comments.

Table 13 – 15 RepResolution View

RepResolution Statistics Control View

The RepResol_Stats_Control view lists information about statistics collection for conflict resolutions for all replicated tables in the database.

Column	Description
sname	Owner of the table.
oname	Table name.
created	Timestamp for which statistics collection was first started.
status	Status of statistics collection: ACTIVE, CANCELLED
status_update _date	Timestamp for which the status was last updated.
purged_date	Timestamp for the last purge of statistics data.
last_purged_start _date	The last start date of the statistics purging date range.
statistics_purged_ end_date	The last end date of the statistics purging date range.

Table 13 – 16 RepResol_Stats_Control View

RepResolution_ Method View

The RepResolution_Method view lists all of the conflict resolution routines available in your current database. Initially, this view lists the standard routines provided with the symmetric replication facility. As you create new user functions and add them as conflict resolution methods for an object in the database, these functions are added to this view.

Column	Description
conflict_type	The type of conflict that the resolution routine is designed to resolve: update, uniqueness, or delete.
method_name	The name of the Oracle–supplied method, or the name of the user–supplied routine.

Table 13 – 17 RepResolution_Method View

RepResolution_ Statistics View

The RepResolution_Statistics view lists information about successfully resolved update, uniqueness, and delete conflicts for all replicated tables. These statistics are only gathered for a table if you have called DBMS_REPCAT.REGISTER_STATISTICS.

Column	Description
sname	The name of the replicated schema.
oname	The name of the replicated table.
conflict_type	The type of conflict that was successfully resolved: delete, uniqueness, or update.
reference_name	The object to which the conflict resolution routine applies. For delete conflicts, this is the table name. For uniqueness conflicts, this is the constraint name. For update conflicts, this is the column group name.
method_name	The name of an Oracle–supplied conflict resolution method. For user–supplied methods, this value is ’user function’.
function_name	For methods of type ’user function’, the name of the user supplied conflict resolution routine.
priority_group	For methods of type ’priority group’, the name of the priority group.
primary_key_value	A concatenated representation of the row’s primary key.
resolved_date	Date on which the conflict for this row was resolved.

Table 13 – 18 RepResolution_Statistics View

RepSchema View

The RepSchema view is provided for backwards compatibility with earlier versions of the symmetric replication facility (prior to release 7.3). Use the RepSite views instead.

Column	Description
sname	The name of the replicated schema. Obsolete for release 7.3 or later.
gname	The name of the replicated object group.
dblink	The database link to which transactions must be queued to replicate changes to each master site.
masterdef	Indicates which of the dblinks is the master definition site.
snapmaster	Used by snapshot sites to indicate which of the dblinks to use when refreshing.
master_comment	Any user–supplied comments.

Table 13 – 19 RepSchema View

Deferred Transaction Views

Oracle provides several views for you to use in administering deferred transactions. These views provide information about each deferred transaction, such as the transaction destinations, the deferred calls that make up the transactions, and any errors encountered during attempted execution of the transaction. *You should not modify the tables directly; use the procedures provided in the DBMS_DEFER and DBMS_DEFER_SYS packages.*

DefCall View

The DefCall view records all deferred remote procedure calls.

Column	Description
callno	Unique ID of call at deferred_tran_db.
deferred_tran_db	The originating database of the deferred call.
deferred_tran_id	The unique ID of the associated transaction.
schemaname	The schema name.
packagename	The package name.
procname	The procedure name of the deferred call.
argcount	The number of arguments to the procedure.

Table 13 – 20 DefCall View

DefCallDest View

The DefCallDest view lists the destinations for each deferred remote procedure call.

Column	Description
callno	Unique ID of call at deferred_tran_db.
deferred_tran_id	Corresponds to the deferred_tran_id in the DefTran view. Each deferred transaction is made up of one or more deferred calls.
deferred_tran_db	The originating database for the deferred transaction. The callno and deferred_tran_db uniquely identify a call.
dblink	The fully qualified database name of the destination database.

Table 13 – 21 DefCallDest View

DefDefaultDest View

If you are not using Oracle's replication facility and do not supply a destination for a deferred transaction or the calls within that transaction, Oracle uses the DefDefaultDest view to determine the destination databases to which you want to defer a remote procedure call.

Column	Description
dblink	The fully qualified database name to which to replicate a transaction.

Table 13 – 22 DefDefaultDest View

DefError View

The DefError view provides the ID of each transaction that could not be applied. You can use this ID to locate the queued calls associated with this transaction. These calls are stored in the DefCall view. You can use the procedures in the DBMS_DEFER_QUERY package to determine the arguments to the procedures listed in the DefCall view.

Column	Description
deferred_tran_db	The fully qualified database name of the database originating or copying the deferred remote procedure calls.
deferred_tran_id	The transaction ID originating or copying the deferred remote procedure calls causing the error.
callno	Unique ID of call at deferred_tran_db.
destination	Database link used to address destination.
error_time	Time error occurred.
error_number	Oracle error number.
error_msg	Error message text.

Table 13 – 23 DefError View

DefSchedule View

The DefSchedule view displays information about when a job is next scheduled to be executed.

Column	Description
dblink	Fully qualified pathname to master database site for which you have scheduled periodic execution of deferred remote procedure calls.
job	Number assigned to job when you created it by calling DBMS_DEFER_SYS.SCHEDULE_EXECUTION. Query the WHAT column of USER_JOBS view to determine what is executed when the job is run.

Table 13 – 24 DefSchedule View, continued on next page

Column	Description
interval	Function used to calculate the next time to apply any changes.
next_date	Next date that job is scheduled to be executed.
last_date	Last time the DBMS_DEFER_SYS.EXECUTE pushed (or attempted to push) remote procedure calls to this destination.
disabled	Is propagation to destination disabled?
last_txn_count	Number of transactions pushed during last attempt.
last_error	Oracle error number from last push
last_msg	Error message from last push.

Table 13 – 24 DefSchedule View

DefTran View

The DefTran view records all deferred transactions.

Column	Description
deferred_tran_id	The transaction ID originating or copying the deferred remote procedure calls.
deferred_tran_db	The fully qualified database name of the database originating or copying the deferred remote procedure calls.
origin_tran_id	The transaction ID originating the deferred remote procedure calls.
origin_tran_db	The fully qualified database name of the database originating the deferred remote procedure calls.
origin_user	The userid of the user originating the deferred remote procedure calls.
delivery_order	An identifier that determines the order of deferred transactions in the queue. The identifier is derived from the system commit number of the originating or copying transaction.
destination_list	’R’ or ’D’. ’R’ indicates that the destinations are determined by the RepSchema view. ’D’ indicates that the destinations were determined by the DefDefaultDest view or the NODE_LIST argument to the TRANSACTION, CALL, or COPY procedures.
start_time	The start time of the originating transaction
commit_comment	Any user–supplied comments.

Table 13 – 25 DefTran View

DefTranDest View

The DefTranDest view lists the destinations for a deferred transaction.

Column	Description
deferred_tran_id	The transaction to replicate to the given database link.
deferred_tran_db	The originating database for the deferred transaction. The deferred_tran_id and deferred_tran_db uniquely identify a transaction.
dblink	The fully qualified database name of the destination database.

Table 13 – 26 DefTranDest View

Viewing Job Queue Information

The following data dictionary views display information about jobs in the job queue:

- DBA_JOBS
- USER_JOBS
- DBA_JOBS_RUNNING

DBA_JOBS and USER_JOBS Views

The DBA_JOBS view provides information about all queued jobs in the database. The USER_JOBS view provides information about the jobs owned by you; that is, jobs for which you are the PRIV_USER.

Column	Description
JOB	Identifier of the job.
LOG_USER	User logged in when the job was submitted. For example, if user SCOTT calls a package that runs in the security domain of SYS, and this package starts a job, the LOG_USER would be SCOTT.
PRIV_USER	User whose default privileges apply to this job. For example, if user SCOTT calls a package that runs in the security domain of SYS, and this package starts a job, the PRIV_USER would be SYS.
SCHEMA_USER	Default schema that parses the job. For example, if the SCHEMA_USER is SCOTT and you submit the procedure HIRE_EMP as a job, Oracle looks for SCOTT.HIRE_EMP.
LAST_DATE	Last date that this job executed successfully, measured to the nearest day.

Table 13 – 27 DBA_JOBS and USER_JOBS Views, continued on next page

Column	Description
LAST_SEC	Last date that this job executed successfully, measured to the nearest second.
THIS_DATE	Date that the current execution of the job started, measured to the nearest day. Usually null if the job is not currently executing.
THIS_SEC	Date that the current execution of the job started, measured to the nearest second. Usually null if the job is not currently executing.
NEXT_DATE	Date that this job is next scheduled to be executed, measured to the nearest day.
NEXT_SEC	Date that this job is next scheduled to be executed, measured to the nearest second.
TOTAL_TIME	Total elapsed time spent by the system on this job, in seconds.
BROKEN	“N” indicates that the job is not broken. “Y” indicates that the job is considered broken and will not be executed.
INTERVAL	Date function that calculates the next time to execute the job.
FAILURES	Number of times that the job was started and failed since it last successfully completed. After 16 failures, a job is marked as broken.
WHAT	Job definition.
CURRENT_ SESSION_LABEL	Trusted Oracle7 label of the current session, as seen by this job.
CLEARANCE_HI	Highest level of clearance available to this job. Applies to Trusted Oracle7 only.
CLEARANCE_LO	Lowest level of clearance available to this job. Applies to Trusted Oracle7 only.
NLS_ENV	ALTER SESSION parameters that describe the NLS environment of this job.
MISC_ENV	Other session parameters that apply to this job.

Table 13 – 27 DBA_JOBS and USER_JOBS Views

DBA_JOBS_RUNNING View

The DBA_JOBS_RUNNING view provides information about any jobs that are currently executing.

Column	Description
SID	Identifier of the session that is executing the job.
JOB	Identifier of the job that is currently executing.
FAILURES	Number of times the job was started and failed since it last successfully completed.
LAST_DATE	Last date this job was successfully executed, measured to the nearest day.
LAST_SEC	Last date this job was successfully executed, measured to the nearest second.
THIS_DATE	Date the current execution of the job started, measured to the nearest day.
THIS_SEC	Date the current execution of the job started, measured to the nearest second.

Table 13 – 28 DBA_JOBS_RUNNING View

Snapshot and Snapshot Refresh Group Views

Oracle provides the following views that have information about snapshots and snapshot refresh groups.

- DBA_SNAPSHOTS
- USER_REFRESH
- USER_REFRESH_CHILDREN

DBA_SNAPSHOTS View

The DBA_SNAPSHOTS catalog view lists information about all of the snapshots in a database.

Column	Description
OWNER	Owner of the snapshot.
NAME	Name of the view used by users and applications for querying and updating the snapshot.
TABLE_NAME	Table in which the snapshot is stored (it has an extra column for the master rowid).
MASTER_VIEW	View of the master table, owned by the snapshot owner, used for refreshes.

Table 13 – 29 DBA_SNAPSHOTS View

Column	Description
MASTER_OWNER	Owner of the master table.
MASTER	Master table that this snapshot copies.
MASTER_LINK	Database link name to the master site.
CAN_USE_LOG	YES if this snapshot can use a snapshot log, NO if this snapshot is too complex to use a log.
UPDATABLE	‘YES’ indicates snapshot is updatable; ‘NO’ indicates read–only.
LAST_REFRESH	Date and time at the master site of the last refresh.
ERROR	Error returned last time an automatic refresh was attempted or number of failed attempts since last successful attempt.
TYPE	Type of refresh for all automatic refreshes: COMPLETE, FAST, FORCE.
NEXT	Date function used to compute next refresh dates.
START_WITH	Date function used to compute next refresh dates.
REFRESH_GROUP	Group identifier for consistent refresh.
UPDATE_TRIG	Name of the trigger that fills in the UPDATE_LOG for an updatable snapshot.
UPDATE_LOG	Name of the table that logs changes to an updatable snapshot.
QUERY	Query used to create the snapshot.

Table 13 – 29 DBA_SNAPSHOTS View

USER_REFRESH View

The USER_REFRESH view lists each refresh group found in the database, and includes information about the refresh interval for each group.

Column	Description
ROWNER	Owner of the refresh group.
RNAME	Name of the refresh group.
REFGROUP	ID number of the refresh group.
IMPLICIT_DESTROY	Implicit delete flag. If this value is Y, Oracle deletes the refresh group after you have subtracted the last member from the group.
JOB	ID number of the job used to execute the automatic refresh of the snapshot group. You can use this information to query the USER_JOBS view for more information about the job.
NEXT_DATE	Date when the members of the group will next be refreshed.
INTERVAL	The function used to calculate the interval between refreshes.
BROKEN	Flag used to indicate a problem with the refresh group. If the value of the broken flag is Y, Oracle will not refresh the group, even if it is scheduled to be refreshed.

Table 13 – 30 USER_REFRESH View

USER_REFRESH_ CHILDREN View

The USER_REFRESH_CHILDREN view lists the members of each refresh group owned by the user, and includes information about the refresh interval for each member.

Column	Description
OWNER	Owner of the refresh group member.
NAME	Name of the refresh group member.
TYPE	Type of the refresh group member; for example, SNAPSHOT.
ROWNER	Owner of the refresh group.
RNAME	Name of the refresh group.
REFGROUP	ID number of the refresh group.

Table 13 – 31 USER_REFRESH_CHILDREN View

Column	Description
IMPLICIT_DE-STROY	Implicit delete flag. If this value is Y, Oracle deletes the refresh group after you have subtracted the last member from the group.
JOB	ID number of the job used to execute the automatic refresh of the snapshot refresh group. You can use this information to query the USER_JOBS view for more information about the job.
NEXT_DATE	Date when the members of the group will next be refreshed.
INTERVAL	The function used to calculate the interval between refreshes.
BROKEN	Flag used to indicate a problem with the refresh group. If the value of the broken flag is Y, Oracle will not refresh the group, even if it is scheduled to be refreshed.

Table 13 – 31 USER_REFRESH_CHILDREN View

APPENDIX

A Release Information

This chapter describes the changes to the symmetric replication facility in the Oracle7 Server Release 7.3. This information includes the following:

- a brief description of each new feature
- a list of obsolete procedures
- a list of new and changed procedures
- a discussion on compatibility issues when running replication in a mixed release environment

New Features

The Oracle7 Server Release 7.3 contains three major enhancements to the advanced replication option:

- table comparison

  The DIFFERENCES and RECTIFY procedures in the DBMS_RECTIFIER_DIFF package allow you to identify and resolve the differences between two replicas of a table. For more information about comparing tables, see page 7 – 15.

- object groups

  The concept of replicating a schema has been eliminated; the unit of replication may now span multiple schemas. Objects must be part of a replicated object group in order to be replicated. Objects cannot belong to more than one object group. For information on creating a replicated object group, see page 4 – 13.

- synchronous data propagation

  Replicated transactions can now be synchronously propagated from one master site to one or more other master sites, or from a snapshot site to its associated master site. Changes from a master site continue to be propagated to its associated snapshot sites asynchronously, as the result of a refresh. For more information on choosing a propagation mode, see page 4 – 33.

- Conflict Notification

  User–defined routines similar to conflict resolution routines can now be used to initiate logging of conflict information in data dictionary views or to initiate events such as email to the DBA, see page 6 – 18.

- Offline Instantiation

  Offline instantiation of a master site allows you to create a new master site while limiting the time required for other sites in your replicated system to be quiesced. It is primarily useful for those sites with very large databases where the time required to transfer the data through network links to the new site would be prohibitive, see page 4 – 19.

  Offline instantiation of a snapshot site is primarily useful for those sites with very large databases (and, thus, a very large amount of snapshot data) where the time required to transfer the data through network links to the new site would be prohibitive.

- Connection Qualifiers.

  Connection qualifiers provide a way to have several database links of the same type (for example, public) that point to the same remote database, yet establish those connections using different communications pathways (for example, an ethernet link or a modem link), .see page 8 – 22.

Obsolete Procedures

This section lists procedures in the DBMS_REPCAT and DBMS_REPCAT_ADMIN packages that are obsolete with this release and lists their replacements. Although the advanced replication option is upwardly compatible with previous releases, you should revise any replication scripts that you have created as soon as possible.

The procedures listed in the "Old Procedure Name" column of Table A – 1 have been replaced by the procedures listed in the "New Procedure Name" column. These changes are the result of the addition of object groups in the current release. The procedures in the "Old Procedure Name" column are simply wrappers for the procedures in the "New Procedure Name" column. For example, if you call CREATE_MASTER_REPSCHEMA, this procedure simply calls CREATE_MASTER_REPGROUP. The new replicated object group is assigned the same name as the schema name that you passed to the CREATE_MASTER_REPSCHEMA procedure.

Old Procedure Name	New Procedure Name
`comment_on_repschema`	`comment_on_repsites`
`comment_on_repcat`	`comment_on_repgroup`
`drop_master_repschema`	`drop_master_repgroup`
`create_snapshot_repschema`	`create_snapshot_repgroup`
`drop_snapshot_repschema`	`drop_snapshot_repgroup`
`refresh_snapshot_repschema`	`refresh_snapshot_repgroup`
`create_master_repschema`	`create_master_repgroup`

Table A – 1 DBMS_REPCAT Obsolete Procedures

Table A – 2 lists the obsolete procedures in the DBMS_REPCAT_ADMIN package.

Old Procedure Name	New Procedure Name
`grant_admin_any_repschema`	`grant_admin_any_repgroup`
`revoke_admin_any_repschema`	`revoke_admin_any_repgroup`
`grant_admin_repschema`	`grant_admin_repgroup`
`revoke_admin_repschema`	`revoke_admin_repgroup`

Table A – 2 DBMS_REPCAT_ADMIN Obsolete Procedures

New and Changed Procedures

Changes to Support Table Comparison

This section describes the procedures that have been added to support the table comparison feature.

Procedure Name	For More Information
`dbms_rectifier_diff.differences`	page 12 – 37
`dbms_rectifier_diff.rectify`	page 12 – 40

Table A – 3 DBMS_RECTIFIER_DIFF Procedures

Changes to Support Object Groups

This section lists the procedures in the DBMS_REPCAT package whose syntax has changed to support the addition of object groups. Each of these procedures used to take a schema name as an argument; they now take an object group name.

Procedure Name
`add_master_database`
`remove_master_databases`
`create_master_repobject`
`alter_master_propagation`
`execute_ddl`
`comment_on_repgroup`
`comment_on_repsite`
`create_master_repgroup`
`create_snapshot_repgroup`
`generate_snapshot_support`
`suspend_master_activity`
`resume_master_activity`

Procedure Name
`relocate_masterdef`
`purge_master_log`
`wait_master_log`
`do_deferred_repcat_admin`
`repcat_import_check`
`switch_snapshot_master`
`create_snapshot_repobject`
`define_priority_group`
`comment_on_priority_group`
`drop_priority_group`
`add_priority_`*`type`*
`alter_priority_`*`type`*
`alter_priority`
`drop_priority`
`drop_priority_`*`type`*
`define_site_priority`
`comment_on_site_priority`
`drop_site_priority`
`add_site_priority_site`
`alter_site_priority_site`
`alter_site_priority`
`drop_site_priority_site`

If you were using positional notation, the sname parameter has been replaced by the gname parameter. For example, if you made the following call to suspend replication activity:

```
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY('acct');
```

You would now be suspending replication activity for the ACCT replicated object group, as opposed to the ACCT replicated schema.

If you are using named notation, your procedures will continue to work. The procedures accept the schema name as an argument, but use the schema name as an object group name. For example, if you made the following call to suspend replication activity:

```
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY(sname => 'acct');
```

Oracle would continue to recognize the SNAME parameter to this procedure; however, it would treat it as an object group name. You would now be suspending replication activity for the ACCT replicated object group, as opposed to the ACCT replicated schema. The complete interface to this procedure is as follows:

```
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY(sname IN VARCHAR2 := '',
                                    gname IN VARCHAR2 := '');
```

You must supply either an SNAME or a GNAME for this procedure to work. Either argument will be interpreted by Oracle as a group name. Although Oracle continues to accept the SNAME parameter, if you are using named notation, you should update your scripts to refer to GNAME where appropriate.

Certain procedures require both a schema name and an object group name. If you do not supply both, the object group name defaults to be the same as the schema name.

Changes to Support Synchronous Replication

This section lists the new and changed procedures in the DBMS_REPCAT package that are used to support synchronous replication.

Procedure Name
`alter_master_propagation`
`add_master_database`
`alter_snapshot_propagation`
`create_snapshot_repgroup`

Compatibility Issues

This section describes compatibility issues between release 7.3 of the advanced replication option and previous releases.

If you will be upgrading all sites in your replicated environment to release 7.3, you should not find it necessary to make any changes to your existing sites. You should review the documentation on the new features and begin to use them at your convenience.

If you will be upgrading only selected sites in your replicated environment to release 7.3, you need to be aware of the following issues:

- It is not necessary to upgrade all sites in a replicated environment to the same release.

- If your replicated environment includes sites prior to release 7.3, your object groups cannot span schemas. Your object group name should correspond to a schema name.
- Pre–release 7.3 sites can only communicate asynchronously with any other sites in the replicated environment.
- Release 7.3 master sites must generate special triggers to support pre–release 7.3 snapshot sites. These triggers are generated automatically by setting the GEN_REP2_TRIGGER flag to TRUE when you call GENERATE_REPLICATION_SUPPORT at the master site.
- If you have a release 7.3 master definition site operating in 7.3 compatibility mode, you must also set the DISTRIBUTED flag to TRUE when you call GENERATE_REPLICATION_SUPPORT to ensure that the appropriate generated objects are created at any pre–release 7.3 sites.

Upgrading and the Advanced Replication Option

This section contains the following topics:

- After a Successful Upgrade
- Release 7.3 Replication Triggers and Packages
- Setting the COMPATIBLE Parameter

After a Successful Upgrade

Enable Release 7.3 new features by setting the initialization parameter COMPATIBLE to 7.3.0.0 following the completion of an upgrade to Release 7.3.

Setting the INIT.ORA initialization parameter COMPATIBLE to 7.3.0.0 puts the database in Release 7.3 compatibility mode. However, both Release 7.3 master sites and Release 7.3 snapshot sites will still operate normally with pre–Release 7.3 replication triggers and wrappers.

Replication support must be regenerated for all replicated objects if the INIT.ORA parameter, COMPATIBLE, is reset to 7.3.0.0 or above to take advantage of new Release 7.3 features, or if you just want to upgrade to new replication triggers and wrappers.

Note: No adjustments to the Release 7.3 database are necessary if the COMPATIBLE parameter remains at a value less than 7.3.0.0 and only pre–Release 7.3 functionality will be used.

Release 7.3 Replication Triggers and Packages

Table 13 – 32 shows the new Release 7.3 replication triggers and packages.

Trigger or Package	Comments
$RT Trigger	This is the replication trigger that propagates changes to replicated tables to other sites. In 7.3.x, there are two flavors of this trigger, a pre–Release 7.3 and a Release 7.3 version. The triggers can be distinguished by the REASON column of the RepGenerated view. A pre–Release 7.3 trigger, denoted by 'REPLICATION TRIGGER', supports only asynchronous propagation. A Release 7.3 trigger, denoted by MIXED REPLICATION TRIGGER, can support both synchronous and asynchronous propagation. For each Release 7.3 trigger generated, an associated $TP package and package body is also generated (see below).
$ST Trigger	This is a pre–Release 7.3 trigger that is generated at master sites if GEN_REP2_TRIGGER=TRUE for DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT(). The purpose of this trigger is for Release 7.3 Masters with pre–Release 7.3 Snapshots. Pre–release 7.3 snapshots do not generate their own replication support, but copy the master site's replication triggers and wrappers. This trigger is created at the master in a disabled state and exists only to be copied by pre–Release 7.3 snapshot sites. Do not enable this trigger at master sites.
$TP Package	This package (and package body) is generated for each Release 7.3 $RT trigger. This package contains the procedures that queue deferred transactions and/or issue synchronous remote procedure calls.

Table 13 – 32 New Release 7.3 Replication Triggers and Packages

Note: A database is said to be in pre–Release 7.3 compatibility mode if the value assigned to its COMPATIBLE parameter (set in INIT.ORA) is less than 7.3.0.0. A database is in Release 7.3 compatibility mode if the value assigned to its COMPATIBLE parameter is 7.3.0.0 or greater.

Setting the COMPATIBLE Parameter

There are three possible upgrade scenarios that you might wish to follow:

- You wish to use new Release 7.3 features at a master definition site.
- You wish to use new, non–replication Release 7.3 features at one or more master sites that are not master definition sites.
- You wish to use new Release 7.3 features at a snapshot site.

Setting COMPATIBILITY at a Master Definition Site

If the COMPATIBLE parameter of any master definition site is reset to 7.3.0.0 to use new features, perform the following steps:

1. Use DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY() to quiesce all object groups that are registered at the affected master definition site.
2. When all object groups are quiesced and the repcatlog is empty, use DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT() to regenerate replication triggers and packages for all replicated objects such as tables, packages, and package bodies. If any pre–Release 7.3 snapshot sites exist, or if there is a possibility that pre–Release 7.3 snapshot sites may be added in the future, the GEN_REP2_TRIGGER parameter must be set to TRUE for DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT.
3. If jobqueues is used, wait until repcatlog becomes empty at the master definition site. Otherwise, use DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN() first at all non–master definition sites which replicate the object group and then at the master definition site to apply administration requests at all sites.

   Warning: This step must be performed twice if jobqueues is not used, because Release 7.3 generates replication support in two phases.
4. Use DBMS_REPCAT.RESUME_MASTER_ACTIVITY() to unquiesce all objects groups and begin normal replication activity. Even though the master definition site is now a 7.3.0.0 compatible Release 7.3 site, new Release 7.3 features will be available only to remote masters that have also reset their compatibility to 7.3.0.0.

Setting COMPATIBILITY at Master Sites

New Release 7.3 replication features can be used only if the master definition site is in Release 7.3 compatibility mode.

Setting COMPATIBILITY at a Snapshot Site

If the COMPATIBLE parameter of a snapshot site is reset to 7.3.0.0, to use new replication features, the following steps should be taken:

1. Make sure that the master site does not have valid, outstanding administration requests. If requests exist, wait until the requests are applied and removed from the administration queue at the master site. It is generally best to wait until the master site's administration queue is empty.
2. Use DBMS_REPCAT.GENERATE_SNAPSHOT_SUPPORT() to. regenerate triggers and packages for all replicated objects, such as tables, packages, and package bodies, at the snapshot site. This assumes that the master site is available and that the master object has already generated replication support. Release 7.3 snapshot sites can use new Release 7.3 features even if their master sites are pre–Release 7.3.

   Note: These steps are recommended for both snapshot sites with a pre–Release 7.3 master site and snapshot sites with a Release 7.3 master site.

Downgrading and the Advanced Replication Option

This section describes what to do after downgrading Oracle.

Additional Information: For instructions on how to upgrade and downgrade the version of your Oracle Server, see the README file provided with your Oracle Server.

Two distinct steps are performed by the downgrade scripts:

1. The object group and all objects replicated in the object group are unregistered if the group name is not an existing schema name.
2. The remaining replicated objects are unregistered as replicated objects if the name of the replicated object's owner does not correspond to the object group name.

   Note: There may be situations in which the object group's name is the same as the object owner's name, but the object group includes objects from multiple schemas.

   These objects are removed only from the replication catalog. The objects themselves are not deleted from the database.

Master Definition Sites or Master Sites

After the completion of the downgrade process, replication triggers are no longer valid because all replication trigger packages were dropped. All packages and package body wrappers were also dropped. This effectively blocks transactions against replicated tables until you regenerate replication support. From the master definition site, take the following steps (this should be done if any master site is downgraded):

1. Use DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY() to quiesce all object groups that are being replicated at the downgraded site(s).
2. When all object groups are quiesced and the administration queue is empty, use DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT() to regenerate replication triggers and packages for all replicated objects such as tables, packages, and package bodies.
3. If jobqueues are used, wait until the administration queue becomes empty at the master definition site. Otherwise use DBMS_REPCAT.DO_DEFERRED_REPCAT_ADMIN(), first at all remote sites and then at the master definition site, to execute administration requests at all sites. This step needs to be performed only once because pre–Release 7.3 databases generate replication support in a single step.
4. Use DBMS_REPCAT.RESUME_MASTER_ACTIVITY() to unquiesce all objects groups and begin normal replication activity.

Downgrading Snapshot Sites

Updatable snapshots that use the $ST trigger will continue to operate normally. All Release 7.3 updatable snapshots that relied on a $TP package are no longer updatable. Since pre–Release 7.3 snapshot sites cannot generate replication support, these snapshots need to be unregistered and re–registered with the Symmetric Replication Facility. Pre–Release 7.3 snapshots do not generate replication support, but rather copy necessary triggers, packages and package bodies from the master site. The extraction is performed only once, when the snapshot replicated object is created.

Advanced Replication Compatibility Between Release 7.3 and Earlier Releases

The following are compatibility issues between Release 7.3 and earlier releases:

- Coexistence of pre–Release 7.3 master sites and Release 7.3 master sites is permitted. You need to have a Release 7.3 master definition site to use new Release 7.3 features.
- Two–phased generation of replication support is not supported in pre–Release 7.3 compatibility mode.
- Replication behaves exactly as it would in a pre–7.3 release when COMPATIBILITY is set to a pre–Release 7.3 value.
- Many existing replication administration requests have been modified to include additional fields to support new Release 7.3 functionality. Release 7.3 sites will process requests with the new fields. Pre–Release 7.3 masters will ignore the new fields and process requests normally. Listed in Table 13 – 33 are RepCatLog requests that are new to Release 7.3:

Requests	Comments
GENERATE_SUPPORT_PHASE1 and GENERATE_SUPPORT_PHASE2	A Release 7.3 master definition site will send out LOG_REQUEST_GEN_SUPPORT_PHASE1 and LOG_REQUEST_GEN_SUPPORT_PHASE2 requests only to Release 7.3 master sites. Remote pre–Release 7.3 master sites will only receive GENERATE_REPLICATION_SUPPORT requests. These requests are only sent to Release 7.3 master sites that have the COMPATIBLE parameter set to 7.3.0 or higher.
ALTER_MASTER_PROPAGATION	This request is only sent to Release 7.3 master sites that have the COMPATIBLE parameter set to 7.3.0 or higher.
ADD_MASTER_DATABASE	The new "overloaded" version of this request is only used at Release 7.3 master definition sites (similar to the already overloaded version used only at master definition site sites in pre–Release 7.3).

Table 13 – 33 New RepCatLog Requests in Release 7.3

The replication administration requests that require regeneration of all replication triggers at all master sites, for example, ADD_MASTER_DATABASE, will check for pre–Release 7.3–compatible triggers at the master definition site and regenerate these triggers as well.

Pre–Release 7.3 snapshot sites will attempt to copy triggers from their master sites. Release 7.3 triggers are incompatible with pre–Release 7.3 triggers. Therefore, DBAs can optionally generate pre–Release 7.3–compatible triggers at Release 7.3 master sites for the pre–Release 7.3 snapshots to copy (set GEN_REP2_TRIGGER to TRUE when calling DBMS_REPCAT.GENERATE_REPLICATION_SUPPORT()). The generated pre–Release 7.3–compatible triggers are disabled at the master sites and are not used at master sites.

APPENDIX

B Operating System–Specific Information

This manual occasionally refers to other Oracle manuals that contain detailed information for using Oracle on a specific operating system. These Oracle manuals are often called installation or user’s guides, although the exact name may vary on different operating systems.

This appendix lists all the references in this manual to operating system–specific Oracle manuals, and lists the O / S–dependent initialization parameters. If you are using Oracle on multiple operating systems, this appendix can help you ensure that your applications are portable across these operating systems.

Operating system–specific topics are listed alphabetically with page numbers of sections that discuss these topics.

Topic	Page
Resources required for installation	4 – 2
Snapshot trace file	3 – 19

Index

Reader’s Comment Form

Oracle7™ Server Distributed Systems, Volume II: Replicated Data
Part No. A32545–2

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

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

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (415) 506–7200

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

Thank you for helping us improve our documentation.