SYBASE®

Administration Guide: Volume 2

Replication Server®

15.1

Contents

About This Book

Sybase® Replication Server® maintains replicated data at multiple sites on a network. Organizations with geographically distant sites can use Replication Server to create distributed database applications with better performance and data availability than a centralized database system can provide.

This book, *Replication Server Administration Guide,* provides an overview of how Replication Server works, and describes Replication Server administrative tasks.

Audience   The *Replication Server Administration Guide* is for replication system administrators, who manage the routine operation of their Replication Servers. Any user who has been granted the sa permission can be a replication system administrator, although each Replication Server usually has just one.

How to use this book   This book contains the following chapters:

- Chapter 1, "Verifying and Monitoring Replication Server" describes checking error logs, verifying that the components of a replication system are running, and monitoring the status of system components and processes.

- Chapter 2, "Customizing Database Operations" describes how to use functions, function strings, and function-string classes to customize data replication with Adaptive Server® Enterprise and data servers from other vendors.

- Chapter 3, "Managing Warm Standby Applications" describes how to create and manage warm standby applications.

- Chapter 4, "Performance Tuning" describes how to manage resources effectively and optimize the performance of individual Replication Servers.

- Chapter 5, "Using Counters to Monitor Performance" describes Replication Server counters and how to use them.

- Chapter 6, "Handling Errors and Exceptions" discusses error conditions and failed transactions and how to customize data server responses to errors.

- Chapter 7, "Replication System Recovery" describes replication system failure conditions and provides procedures for recovering from them.

- Appendix A, "Asynchronous Procedures" describes a method for replicating stored procedures associated with table replication definitions.

- Appendix B, "High Availability on Sun Cluster 2.2," provides background and procedures for configuring Sybase Replication Server for high availability (HA) on Sun Cluster 2.2.

- Appendix C, "Pre-15.1 Request Function Replication" provides information about request function replications with versions earlier than 15.1.

Volume 1 of the System Administration Guide contains these chapters:

- Chapter 1, "Introduction" introduces you to Replication Server, describing the role it plays in a distributed database system and its concepts and components.

- Chapter 2, "Replication Server Technical Overview" provides a technical overview of the replication system, giving you the background necessary to maintain and troubleshoot the system.

- Chapter 3, "Managing Replication Server with Sybase Central"describes using Sybase Central's Replication Manager plug-in, which is a graphical tool for managing Replication Server.

- Chapter 4, "Managing a Replication System" describes basic operations such as starting, stopping, and configuring Replication Server.

- Chapter 5, "Setting Up and Managing RepAgent," describes how to set up, configure and manage RepAgent.

- Chapter 6, "Managing Routes" describes how to create and manage routes between source and destination Replication Servers.

- Chapter 7, "Managing Database Connections" describes how to prepare databases for replication and how to create and manage connections between databases and Replication Servers.

- Chapter 8, "Managing Replication Server Security" describes how to create and modify login names, passwords, and permissions and how to set up network-based security.

- Chapter 9, "Managing Replicated Tables" describes how to set up and manage replicated tables.

- Chapter 10, "Managing Replicated Functions" describes how to copy the execution of user stored procedures to remote sites in a replication system using replication definitions.

- Chapter 11, "Managing Subscriptions" describes how to create and manage subscriptions, which allow Replication Server to replicate data between databases.

- Chapter 12, "Managing Replicated Objects Using Multisite Availability," describes how to create and manage database replication definitions and database subscriptions.

Related documents   The Sybase Replication Server documentation set consists of the following:

- The release bulletin for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the release bulletin may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Product Manuals at http://www.sybase.com/support/manuals/.

- *Installation Guide* for your platform – describes installation and upgrade procedures for all Replication Server and related products.

- *New Features Guide* – describes the new features in Replication Server version 15.1 and the system changes added to support those features.

- *Administration Guide* (this book) – contains an introduction to replication systems. This manual includes information and guidelines for creating and managing a replication system, setting up security, recovering from system failures, and improving performance.

- *Configuration Guide* for your platform – describes configuration procedures for all Replication Server and related products, and explains how to use the rs_init configuration utility.

- *Design Guide* – contains information about designing a replication system and integrating heterogeneous data servers into a replication system.

- *Getting Started with Replication Server* – provides step-by-step instructions for installing and setting up a simple replication system.

- *Heterogeneous Replication Guide* – describes how to use Replication Server to replicate data between databases supplied by different vendors.

- *Reference Manual* – contains the syntax and detailed descriptions of Replication Server commands in the Replication Command Language (RCL); Replication Server system functions; Sybase Adaptive Server commands, system procedures, and stored procedures used with Replication Server; Replication Server executable programs; and Replication Server system tables.

- *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Available only in print version.

- *Troubleshooting Guide* – contains information to aid in diagnosing and correcting problems in the replication system.

- Replication Manager plug-in help, which contains information about using Sybase Central™ to manage Replication Server.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks™ CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.

- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1 Point your Web browser to Technical Documents at
   http://www.sybase.com/support/techdocs/.

2 Click Certification Report.

3 In the Certification Report filter select a product, platform, and timeframe
   and then click Go.

4 Click a Certification Report title to display the report.

❖ **Finding the latest information on component certifications**

1 Point your Web browser to Availability and Certification Reports at
   http://certification.sybase.com/.

2 Either select the product family and product under Search by Base
   Product; or select the platform and product under Search by Platform.

3 Select Search to display the availability and certification report for the
   selection.

❖ **Creating a personalized view of the Sybase Web site (including support
   pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create
a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at
   http://www.sybase.com/support/techdocs/.

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1 Point your Web browser to the Sybase Support Page at
   http://www.sybase.com/support.

2 Select EBFs/Maintenance. If prompted, enter your MySybase user name
   and password.

3 Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is
   displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.

5   Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions   This section describes the style and syntax conventions, RCL command formatting conventions, and icons used in this book.

Style conventions   Syntax statements that display the syntax and options for a command are printed as follows:

alter user *user*
set password *new_passwd*
[verify password *old_passwd*]

See "Syntax conventions" on page xvi for more information.

Examples that show the use of Replication Server commands are printed as follows:

```
alter user louise
 set password somNIfic
 verify password EnnuI
```

Command names, command option names, program names, program flags, keywords, functions, and stored procedures are printed as follows:

Use alter user to change the password for a login name.

Variables, parameters, and user-supplied words are in italics in syntax and in paragraph text, as follows:

The set password *new_passwd* clause specifies a new password.

Names of database objects such as databases, tables, columns, and datatypes, are in italics in paragraph text, as follows:

The base_price column in the Items table is a money datatype.

Names of replication objects, such as function-string classes, error classes, replication definitions, and subscriptions, are in italics.

Syntax conventions   Syntax formatting conventions are summarized in the following table. Examples combining these elements follow.

Table 1: Syntax formatting conventions

Key	Definition
{ }	Curly braces mean you must choose at least one of the enclosed options. Do not include braces in the command.
[ ]	Brackets mean you may choose or omit enclosed options. Do not include brackets in the command.
|	Vertical bars mean you may choose no more than one option (enclosed in braces or brackets).
,	Commas mean you may choose as many options as you need (enclosed in braces or brackets). Separate your choices with commas, to be typed as part of the command. Commas may also be required in other syntax contexts.
( )	Parentheses are to be typed as part of the command.
...	An ellipsis (three dots) means you may repeat the last unit as many times as you need. Do not include ellipses in the command.

Obligatory choices

• Curly braces and vertical bars – choose only one option.

    {red | yellow | blue}

• Curly braces and commas – choose one or more options. If you choose more than one, separate your choices with commas.

    {cash, check, credit}

Optional choices

• One item in square brackets – choose it or omit it.

    [anchovies]

• Square brackets and vertical bars – choose none or only one.

    [beans | rice | sweet_potatoes]

• Square brackets and commas – choose none, one, or more options. If you choose more than one, separate your choices with commas.

    [extra_cheese, avocados, sour_cream]

Repeating elements

An ellipsis (...) means that you may repeat the last unit as many times as necessary. For the alter replication definition command, for example, you can list one or more columns and their datatypes for the add clause or the add searchable columns clause:

    alter replication definition *replication_definition*
    {add *column datatype* [, *column datatype*]... |
     add searchable columns *column* [, *column*]... |
     replicate {minimal | all} columns}

RCL command formatting	RCL commands are similar to Transact-SQL® commands. The following sections present the formatting rules.
Command format and command batches	• You can break a line anywhere except in the middle of a keyword or an identifier. You can continue a character string on the next line by typing a backslash (\) at the end of the line.
	• Extra spaces are ignored, except after a backslash. Do not enter any spaces after a backslash.
	• You can enter more than one command in a batch unless otherwise instructed.
	• RCL commands are not transactional. Each command is executed independently and is not affected by the completion status of other commands in the batch. However, syntax errors in a command prevent Replication Server from executing subsequent commands in a batch.
Case sensitivity	• Keywords in RCL commands are not case sensitive. You can enter them in any combination of uppercase or lowercase letters.
	• Case sensitivity in identifiers and character data depends on the sort order that is in effect.
	• If you use a case-sensitive sort order such as "binary," you must enter identifiers and character data in the correct combination of uppercase and lowercase letters.
	• If you use a sort order that is not case sensitive, such as "nocase," you can enter identifiers and character data in any combination of uppercase or lowercase letters.
Identifiers	Identifiers are names you give to servers, databases, variables, parameters, database objects, and replication objects. Database object names include names for tables, columns, and views. Replication object names include names for replication definitions, subscriptions, functions, and publications.
	• Identifiers can be 1 – 255 bytes long (equivalent to 1 – 255 single-byte characters) and must begin with a letter, the @ sign, or the _ character. See "Support for longer identifiers" on page 122 of the *Replication Server Administration Guide Volume 1*, for a list of identifiers that have been extended to 255 bytes.
	• Replication Server function parameters are the only identifiers that can begin with the @ character. Function parameter names can include 255 characters *after* the @ character.

- After the first character, identifiers can include letters, digits, and the #, $, or _ characters. Spaces are not allowed.

Parameters in function strings

Parameters in function strings have the same rules as identifiers, except that:

- They are enclosed in question marks (?). This allows Replication Server to locate them in the function string. Use two consecutive question marks (??) to represent a literal question mark in a function string.

- The exclamation point (!) introduces a parameter modifier that indicates the source of the data to be substituted for a parameter at runtime. Refer to the *Replication Server Reference Manual* for a list of modifiers.
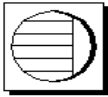
Data support   Replication Server supports all Adaptive Server datatypes.

User-defined datatypes are not supported. The double precision, nchar, and nvarchar datatypes are indirectly supported; they are mapped to other datatypes.

For more information about the supported datatypes, including how to format them, see "Datatypes," in Chapter 2, "Topics" of the *Replication Server Reference Manual*.

Icons

Illustrations in this book use icons to represent the components of a replication system.

	Description
	This icon represents Replication Server, the Sybase server program maintains replicated data on a local-area network (LAN) and processes data transactions received from other Replication Servers on wide-area network (WAN).
	This icon represents Adaptive Server, the Sybase data server. Data servers manage databases containing primary or replicated data. Replication Server also works with heterogeneous data servers, so, unless otherwise noted, this icon can represent any data server in a replication system.
	This icon represents Replication Agent™, a replication system process or module that transfers transaction log information for primary database to a Replication Server. The Replication Agent for Adaptive Server is RepAgent. Sybase provides Replication Agent products for Adaptive Server® Anywhere, DB2, Microsoft SQL Server, and Oracle data servers.
	Except for RepAgent, which is an Adaptive Server thread, all Replication Agents are separate processes. In general, this icon only appears when representing a Replication Agent that is a separate process.

	Description
	This icon represents client application. A client application is a user process or application connected to a data server. It may be a front-end application program executed by a user or a program that executes as an extension of the system.
	This icon represents the Sybase Central Replication Manager (RM) plug-in, a management utility that lets a replication system administrator develop, manage, and monitor a Sybase Replication Server environment.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Replication Server HTML documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1   Verifying and Monitoring Replication Server

This chapter describes checking error logs, verifying that the components of a replication system are running, and monitoring the status of system components and processes.

Topic	Page
Checking replication system log files for errors	2
Verifying a replication system	2
Monitoring Replication Server	4
Setting and using threshold levels	8

The replication system includes data servers and Replication Servers. It may also include Replication Agents for heterogeneous data servers. The Replication Agent for Adaptive Server is RepAgent, an Adaptive Server thread.

Note If you are using a Replication Agent for a heterogeneous data server, refer to Replication Agent documentation for your data server for information about troubleshooting your Replication Agent.

In a fully operational replication system, all data servers, Replication Servers, Replication Agents, and their internal threads and other components are running. This chapter tells you how to perform basic troubleshooting tasks on the replication system, including:

1   Checking error logs for status and error messages.

2   Logging in to system servers and checking that all threads are functioning, routes and connections are in place, and the interfaces file information is correct.

This chapter also describes how you can monitor Replication Server and its threads and check partition threshold levels.

Refer to the *Replication Server Troubleshooting Guide* for detailed information about monitoring and troubleshooting Replication Server.

Checking replication system log files for errors

The Replication Server records status and error messages, including internal errors, in the Replication Server error log file. Use the admin log_name command to display the path to the current log file. The default name for the log file is *repserver.log*. You can change the default name by executing repserver with the -E option and specifying the new log file name.

Refer to Chapter 3 "Replication Server Commands" of the *Replication Server Reference Manual*, for more information about these commands.

Internal errors are those where the only action available to Replication Server is to dump the stack and exit. For diagnostic purposes, Replication Server prints a trace of its execution stack in the log and leaves a record of its state when the error occurred.

Messages continue to accumulate in the error log files until you remove them. For this reason, you may choose to truncate the log files when the Replication Server is shut down. You can also close the Replication Server log file and begin a new log file by using the admin set_log_name command.

The Replication Server log file contains messages generated during the execution of asynchronous commands, such as create subscription and create route, which continue processing after the commands complete. While you are executing asynchronous commands, pay special attention to the log files for the Replication Servers affected by the procedure.

If a log file is unavailable, important error information is written to the standard error output file, which you can display on a terminal or redirect to a file.

Verifying a replication system

You need to verify that the entire replication system is working when you are about to create replication definitions or subscriptions or when you are performing diagnostics on your system. If you encounter errors, verifying your system allows you to rule out the possibility that threads or components are not running or that routes and connections are not properly set up.

To make sure that Replication Server threads are running, you can execute admin who_is_down, which displays only those threads that are not running. Alternatively, execute admin who to display information about all threads.

If no threads are down, you can confirm that the replication system is working by checking the following:

1   Verify that replication system servers and Replication Agents are running and available.

At the primary site, log in to these servers:

- Data server with the primary database and its Replication Agent

  If you are using Adaptive Server, execute sp_help_rep_agent at Adaptive Server to display status information for RepAgent thread.

- Replication Server managing the primary database

- RSSD (and its Replication Agent) for the primary Replication Server

  If you are using Adaptive Server, execute sp_help_rep_agent at Adaptive Server to display status information for RepAgent thread.

At replicate sites, log in to these servers:

- Data servers with replicate databases and, if request functions are executed at these databases, their Replication Agents

  If you are using Adaptive Server, execute sp_help_rep_agent at Adaptive Server to display status information for RepAgent thread.

- Replication Servers managing replicate databases

- RSSDs (and their Replication Agents) for replicate Replication Servers

  If you are using Adaptive Server, execute sp_help_rep_agent at Adaptive Server to display status information for RepAgent thread.

2   Use the admin show_connections command at Replication Server to verify that these routes and connections are in place:

- Routes from the primary Replication Server to each replicate Replication Server

- Database connection between the primary Replication Server and the primary database

- Route from a replicate Replication Server to the primary Replication Server, if the replicate Replication Server manages a replicate database in which request functions are executed

- Database connections between each replicate Replication Server and its replicate database

3   Verify the accuracy of entries in the interfaces file.

When creating subscriptions, be sure an entry for the primary data server exists in the interfaces file for the replicate Replication Server. (If you are using atomic or non-atomic materialization, the replicate Replication Server retrieves initial rows through a direct connection to the primary data server.)

4   Use the admin who command to verify that these Replication Server threads are running:

- Data Server Interface (DSI)

- Replication Server Interface (RSI)

- Distributor (DIST)

- Stable Queue Manager (SQM)

- Stable Queue Transaction interface (SQT)

- RepAgent User

For detailed information about monitoring Replication Server threads, refer to "Displaying replication system thread status" on page 6.

Monitoring Replication Server

While the replication system is in operation, you may need to monitor its components and processes. This section describes how to:

- Monitor replication system servers

- Monitor DSI, RSI, and other thread status

- Use system information commands to obtain information about various aspects of the Replication Server.

Verifying server status

You can verify the status of your servers with these methods:

- Use isql to log in to each server. If the login succeeds, you know that the server is running.

- Create a script that logs in to and displays the status of each Adaptive Server and its RepAgent thread, other Replication Agent (if any), and Replication Server. Make sure all servers in the script are included in the interfaces file.

If a login fails, it may be caused by one of the following problems:

Problem: You typed an incorrect name, or the interfaces file you are using does not have an entry for the server.

```
DB-LIBRARY error:
  Server name not found in interface file.
```

Problem: The server is running, but you specified an incorrect login name or password.

```
DB-LIBRARY error:
  Login incorrect.
```

Problem: The server is not running.

```
Operating-system error:
  Invalid argument
 DB-LIBRARY error:
  Unable to connect: Server is unavailable
  or does not exist.
```

Problem: The interfaces file cannot be found.

```
Operating-system error:
  No such file or directory
 DB-LIBRARY error:
  Could not open interface file.
```

Problem: The interfaces file exists, but you do not have permission to access it.

```
Operating-system error:
  Permission denied
 DB-LIBRARY error:
   Could not open interface file
```

If you can not log in but do not receive an error message, you can assume that the server has stopped processing. Call Sybase Technical Support if you need assistance in determining the problem.

Visual monitoring of status

Replication Manager graphically displays an environment or object status. The status of an environment is the state of its components. An object's status includes its current state and a list of reasons for the state. The state of each object is displayed on the object icon, in the parent object Details list, and on the Properties dialog box for that object. You can monitor the status of servers, connections, routes, and queues.

Use the Replication Manager GUI to monitor the status in Replication Monitoring Services (RMS). The Replication Manager connects to the servers in the environment through RMS.

Refer to Chapter 3, "Managing Replication Server with Sybase Central" in the *Replication Server Administration Guide Volume 1* for more information.

Displaying replication system thread status

You can monitor general information on current Replication Server threads. Table 1-1 describes threads that apply to database connections and routes and the admin who command you use to monitor them.

Table 1-1: Monitoring Replication Server threads

Replication Server thread	Command
Distributor (DIST) – uses SQT and SQM to read transactions from the inbound queue.	admin who, dist
Data Server Interface (DSI) – submits transactions to data server.	admin who, dsi
REP AGENT USER – verifies that transactions from the data server are valid and writes them to the inbound queue.	admin who
	Note Use sp_who or sp_help_rep_agent to display status of RepAgent thread at Adaptive Server.
Replication Server Interface (RSI) – logs in to each destination Replication Server and transfers commands from the stable queue to the destination server.	admin who, rsi
Stable Queue Manager (SQM) – manages Replication Server stable queues.	admin who, sqm
Stable Queue Transaction interface (SQT) – reads transactions in a queue and passes them to the SQT reader.	admin who, sqt

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for details on the admin who command. Refer to the *Replication Server Troubleshooting Guide* to interpret the command output for troubleshooting purposes.

Using system information commands

In addition to admin who, Replication Server offers other admin commands to assist you in monitoring Replication Server.

These commands are listed in Table 1-2. Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for details on each command.

Table 1-2: Overview of system information commands

Command	Description
admin disk_space	Displays utilization of disk partitions accessed by the Replication Server.
admin echo	Determines if the local Replication Server is running.
admin get_generation	Retrieves the generation number for a primary database, used in recovery operations.
admin health	Displays the overall status of the Replication Server.
admin log_name	Displays the path to the current log file.
admin logical_status	Displays the status of logical database connections, used in warm standby applications.
admin pid	Displays the process ID of the Replication Server.
admin quiesce_check	Determines if the queues in the Replication Server have been quiesced.
admin quiesce_force_rsi	Determines whether a Replication Server is quiescent. Also forces Replication Server to deliver outbound messages.
admin rssd_name	Displays the names of the data server and database for the RSSD.
admin security_property	Displays security features of network-based security systems supported by Replication Server.
admin security_setting	Displays network-based security settings of a particular target server.
admin set_log_name	Closes the existing Replication Server log file and opens a new log file.
admin show_connections	Displays information about all connections and routes to and from Replication Server.
admin show_function_classes	Displays the names of existing function-string classes and their parent classes and indicates the number of levels of inheritance.
admin show_route_version	Displays the version number of routes that originate at Replication Server and routes that terminate at Replication Server.
admin show_site_version	Displays the site version of Replication Server.
admin sqm_readers	Displays information about threads that are reading the inbound queue.

Command	Description
admin stats	Displays information and statistics about Replication Server counters. Replaces admin statistics.
admin statistics, md	Displays statistics about message delivery and counters.
admin statistics, mem	Displays statistics about memory utilization.
admin statistics, reset	Resets the message delivery statistics.
admin version	Displays which version of the Replication Server you are running, representing the software version.
admin who	Displays information about all threads in the Replication Server.
admin who, dsi	Displays information about DSI threads that connect to a data server.
admin who, rsi	Displays information about RSI threads that connect to other Replication Servers.
admin who, sqm	Displays information about all queues managed by the SQM.
admin who, sqt	Displays information about all queues managed by the SQT.
admin who_is_down	Displays the same information as admin who, but only about threads that are down.
admin who_is_up	Displays the same information as admin who, but only about threads that are running.

Setting and using threshold levels

Stable queue partitions fill up when a Replication Server is receiving more messages than it is sending. For example, if a network is down between a primary site and a replicate site, the Replication Server at the primary site queues up the undeliverable messages. When the network returns to service, the messages can be delivered, and then deleted from the primary Replication Server partitions.

If a partition becomes completely full, senders cannot deliver their messages to the Replication Server, and messages begin to back up in the partitions at previous sites and in the transaction logs for primary databases.

Warning! If the situation is not corrected, RepAgent is unable to update the secondary truncation point in the database log, and the transaction log fills. Clients are then unable to execute transactions at the primary database.

You can configure Replication Server to warn when partitions become too full by setting three rows in the rs_config system table: sqm_warning_thr1, sqm_warning_thr2, and sqm_warning_thr_ind. These parameters are described in Figure 4-2 on page 130.

Monitoring partition percentages

Replication Server operates on 1MB partition segments. Whenever it allocates or deallocates a partition segment, it calculates these statistics:

• Percentage of total partition segments in use

• Percentage of total partition segments in use by the affected stable queue

If the percentage of partition segments in use rises above the percentage specified by sqm_warning_thr1 or sqm_warning_thr2, a message like the following is written to the log file:

```
WARNING: Stable Storage Use is Above threshold percent
```

If you see this message often, you may need to add partitions to the Replication Server or correct a recurring failure that causes the queues to fill.

When the first percentage drops below the percentage specified by sqm_warning_thr1 or sqm_warning_thr2, a message like the following is written to the log file to note that the condition that caused the original warning no longer exists:

```
WARNING CANCEL: Stable Storage Use is Below threshold
percent
```

The percentage of total partition segments in use by the affected stable queue triggers the following warning message when the percentage of the total space used by a single stable queue exceeds the percentage specified by sqm_warning_thr_ind:

```
WARNING: Stable Storage Use by queue name is Above
threshold percent
```

This warning alerts you to problems that cause a particular stable queue to fill until it is using a disproportionate share of the total partition space. For example, if a route is suspended for a length of time, its stable queue may fill until it occupies enough partition space to trigger a warning.

When the percentage of the total partition space used by a stable queue drops below the sqm_warning_thr_ind percentage, Replication Server writes a cancel message like the following to the log file:

```
WARNING CANCEL: Stable Storage Use by queue name is
Below threshold percent.
```

Customizing Database Operations

This chapter explains how you can create and alter functions, function strings, and function-string classes to allow replication definitions to work with database servers other than Adaptive Server.

Overview

Replication Server translates commands from the primary database into Replication Server functions that represent data server operations such as insert, delete, select, begin transaction, and so on. It distributes these functions to remote Replication Servers in the system, where they execute those operations in remote databases.

The primary Replication Server distributes functions in the same format regardless of the type of data server that actually updates the replicated data. Functions are not database-specific. They include all the data needed to perform the operation, but they do not specify the syntax needed to complete the operation at the destination data server.

The remote Replication Server converts functions to commands specific to the destination data servers where they are executed. A function string contains the database-specific instructions for executing a function. The replicate Replication Server managing a database uses an appropriate function string to map the function to a set of instructions for the data server. For example, the function string for the rs_insert function provides the actual language to be applied in a replicate database.

This separation between functions and data server commands lets you maintain replicated data among heterogeneous data servers. Replication Server allows you to customize function strings, specifying how Replication Server functions map to SQL commands. You can create function strings if you require customized data server operations. You customize replicated data applications by changing the way operations are performed at the destination database.

Function strings are grouped into function-string classes, so you can group mappings of functions to commands according to data server. Replication Server provides function-string classes for Adaptive Server Enterprise, Oracle, Microsoft SQL Server, Adaptive Server Anywhere, IMS, VSAM, and DB2 databases. You can create new derived function-string classes in which you customize certain function strings and inherit all others from these or other classes. You can also create entirely new classes in which you create all new function strings.

You may also need to create function strings for replicated functions, which allow you to execute stored procedures on remote databases. You must create a function string for any replicated function for which Replication Server does not automatically generate a function string in the function-string class used by the destination database.

Working with functions, function strings, and classes

You can work with functions and function strings to customize database operations in any of these ways:

- Create a new function-string class for use with a specific type of database, and customize some or all of the function strings. See "Managing function-string classes" on page 26 for detailed information.

- For atomic materialization, use a function from a function-string class associated with the primary database connection, not a function from the function-string class associated with the replicate database connection.

- Alter function strings for the system-provided function-string class, rs_sqlserver_function_class. See "Managing function strings" on page 32 for detailed information.

- Create a function-string class that inherits, either directly or indirectly, function strings from the system-provided function-string class rs_default_function_class.

- Use the system-provided function-string classes for non-Sybase data servers: rs_db2_function_class, rs_mss_function_class, or rs_oracle_function_class. See "Translating datatypes using HDS" on page 317 in the *Replication Server Administration Guide Volume 1* for detailed information on datatype translations using the heterogeneous datatype support (HDS) feature.

This section provides an overview of functions, function strings, and function-string classes. The following sections include a summary of the system functions, procedures, and guidelines for managing function strings and function-string classes. They also summarize commands for displaying information about the function strings and classes in the replication system.

You can work with functions, function strings, and classes using Sybase Central or RCL commands. This chapter describes procedures and RCL commands that you enter at the command line using isql.

Refer to Chapter 4, "Replication Server System Functions," in the *Replication Server Reference Manual* for more information about the system functions.

Functions

Replication Server uses two major types of functions:

- System functions
- User-defined functions

You can create custom function strings for either type of function, depending on your needs.

See "Managing function strings" on page 32 for more information about when to customize function strings.

System functions

System functions represent data server operations whose function strings are supplied by Replication Server or are available when you install a new database on the replication system. Unless your application requires it, you do not need to customize function strings for system functions. The system-provided class generates them for you.

System functions include:

- Functions that represent data-manipulation operations such as insert, update, delete, select, and select with holdlock.

  These system functions have replication-definition scope. See "Function scope" on page 15 for details.

- Functions that represent transaction-control directives. These functions include operations such as begin transaction and commit transaction.

  These system functions have function-string-class scope. See "Function scope" on page 15 for details.

See "Summary of system functions" on page 16 for more information about each type of system function.

User-defined functions

User-defined functions allow you to use Replication Server to distribute replicated stored procedures between sites in the replication system. You must create function strings for user-defined functions unless you use a function-string class that directly or indirectly inherits function strings from rs_default_function_class. User-defined functions include:

- Functions that are used in replicating stored procedures associated with function replication definitions. Replication Server automatically creates a user-defined function of this type when you create a function-replication definition.

  Refer to Chapter 10, "Managing Replicated Functions" in the *Replication Server Administration Guide Volume 1* for details about function-replication definitions and replicated stored procedures.

- Functions that are used in replicating stored procedures associated with table-replication definitions. You create and maintain user-defined functions of this type yourself.

  For details about replicated stored procedures that use table-replication definitions, see Appendix A, "Asynchronous Procedures."

User-defined functions have replication-definition scope. See "Function scope" on page 15 for details.

Any function string that you create for a user-defined function should be created at the primary Replication Server, where the replication definition was created. If you are using function replication definitions, see also "Implementing an applied function" on page 337 or "Implementing a request function" on page 340 in the *Replication Server Administration Guide Volume 1*.

Function scope

The scope of a function defines the object to which the function applies: either to a replication definition or to a function-string class. Knowing a function's scope is important for determining where to customize a function string: at the primary or replicate Replication Server. Functions can have one of two scopes:

- Function-string-class scope

- Replication-definition scope

Function-string-class scope

A function with function-string-class scope is defined once for the class. Functions with function-string-class scope include system functions that represent transaction-control directives (such as rs_begin, rs_commit, or rs_marker) and do not perform data manipulation. Function strings for user-defined functions do not have class scope.

Function strings for functions with function-string-class scope must be customized at the primary Replication Server for the function-string class. See Table 2-1 on page 16 for a list of these functions. See "Primary site for a function-string class" on page 29 for information on assigning a primary site.

Replication-definition scope

A function with replication-definition scope is defined once for a specific table-replication definition or function-replication definition—although the function may have multiple function strings.

Functions with replication-definition scope include:

- System functions that perform data-manipulation operations (such as rs_insert, rs_delete, rs_update, rs_select, rs_select_with_lock, and special functions used in replicating text, unitext, and image data).

  See Table 2-2 for a list of these functions.

- User-defined functions for table- or function-replication definitions.

System functions with replication-definition scope must be customized at the Replication Server where the replication definition was created. User-defined functions with replication-definition scope must be customized at the Replication Server where the replication definition was created.

Summary of system functions

The following tables provide a summary of the available system functions. Refer to Chapter 4, "Replication Server System Functions," in the *Replication Server Reference Manual* for complete documentation of all of the system functions.

System functions with function-string-class scope

Table 2-1 lists the system functions with function-string-class scope. Replication Server provides default generated function strings for each system-provided class when you install the replication system.

Some functions are required for every Replication Server application, while other functions only apply in particular cases, such as warm standby applications, parallel DSI threads, or coordinated dumps.

If you use a function-string class other than the default (rs_sqlserver_function_class), and you are not using function-string inheritance, you must create a function-string for each system function you use that has function-string class scope.

Customize function strings for system functions with class scope at the Replication Server that is the primary site for the function-string class. See "Changing the primary site for a function-string class" on page 30 for more information about assigning or changing the primary Replication Server for a function-string class.

Table 2-1: System functions with function-string-class scope

Function name	Description
rs_batch_start	Specify the SQL statements required in addition to the rs_begin statements to mark the beginning of a batch of commands.
rs_batch_end	Specify the SQL statements required to mark the end of a batch of commands. This function string is used with rs_batch_start.
rs_begin	Begin a transaction.
rs_check_repl	Check if a table is marked for replication.
rs_commit	Commit a transaction.

Function name	Description
rs_dumpdb	Initiate a coordinated database dump.
rs_dumptran	Initiate a coordinated transaction dump.
rs_get_charset	Return the character set used by a data server.
	Sample function strings for replication into DB2 databases via Net-Gateway are installed in the Sybase release directory in *install/rs_db2_setup.sample* (UNIX systems) and *install\rs_2_db2.txt* (Windows 2000, 2003 systems).
rs_get_lastcommit	Retrieve rows from the rs_lastcommit system table.
rs_get_sortorder	Return the sort order used by a data server.
	Sample function strings for replication into DB2 databases via Net-Gateway are installed in the Sybase release directory in *install/rs_db2_setup.sample* (UNIX systems) and *install\rs_2_db2.txt* (Windows 2000 and 2003 systems).
rs_get_thread_seq	Return the current sequence number for the specified entry in the rs_threads system table. This function is executed only when you are using parallel DSI.
rs_get_thread_seq_noholdlock	Return the current sequence number for the specified entry in the rs_threads system table, using the noholdlock option. This thread is used when dsi_isolation_level is 3.
rs_initialize_threads	Set the sequence of each entry in the rs_threads system table to 0. This function is executed only when you are using parallel DSI.
rs_marker	Help coordinate subscription materialization. The function passes its first parameter to Replication Server as an independent command.
rs_raw_object_serialization	Replicate Java columns as serialized data.
rs_repl_off	Set replication off in Adaptive Server for a standby database connection.
rs_repl_on	Set replication on in Adaptive Server for a standby database connection.
rs_rollback	Roll back a transaction.
rs_set_ciphertext	Turn on set ciphertext on, which enables replication of encrypted columns for rs_default_function_class and rs_sqlserver_function_class. For all other classes, this function is set to null.
rs_set_isolation_level	Passes the isolation level for transaction to replicate data server.
rs_set_dml_on_computed	Is applied at the replicate database DSI when a connection is established. It issues the command set dml_on_computed "on" after the use database statement
rs_set_proxy	Assume the permissions, login name, and server user ID of the user.
rs_thread_check_lock	Determines whether or not the DSI executor thread is holding a lock that blocks a replicate database process.
rs_triggers_reset	Set triggers off in Adaptive Server for a standby database connection.
rs_trunc_reset	Reset the secondary truncation point in warm standby databases. This function is executed only when you create a warm standby database or when you switch to a standby database.

Function name	Description
rs_trunc_set	Set the secondary truncation point in warm standby databases. This function is executed only when you create a warm standby database or when you switch to a standby database.
rs_update_threads	Update the sequence number for the specified entry in the rs_threads table. This function is executed only when you are using parallel DSI.
rs_usedb	Change the database context.

System functions with replication-definition scope

Table 2-2 lists the system functions with replication-definition scope. Replication Server provides default function strings for each system-provided class when you create a replication definition.

Some functions are required for every Replication Server application, while other functions only apply in particular cases, such as replication of text, unitext, and image datatypes, parallel DSI threads, or performing subscription materialization or dematerialization.

Customize function strings for a system functions with replication-definition scope at the Replication Server where the replication definition was created.

Table 2-2: System functions with replication definition scope

Function name	Description
rs_datarow_for_writetext	Provide an image of the data row associated with a text, unitext, or image column updated with a Transact-SQL writetext command or with CT-Library or DB-Library™ functions.
rs_delete	Delete a row in a table.
rs_get_textptr	Retrieve the text pointer for a text, unitext, image, or rawobject column.
rs_insert	Insert a row into a table.
rs_select	Retrieve rows from a table for subscription materialization or dematerialization.
rs_select_with_lock	Retrieve subscription materialization or dematerialization rows using a holdlock.
rs_textptr_init	Allocate a text pointer for a text, unitext, image, or rawobject column.
rs_truncate	Truncate a table.
rs_update	Update a row in a table.
rs_writetext	Alter text, unitext, image, or rawobject data.

Function strings

Function strings contain instructions for executing a function in a database. These instructions may differ according to database. For example, a non-Sybase database may require different instructions and have different function strings than an Adaptive Server database.

Functions strings come in two formats: language and RPC. A language-format function string contains a command, such as a SQL statement, that the data server parses. An RPC-format function string contains a remote procedure call that executes a registered procedure in an Open Server™ gateway application or in an Adaptive Server database. Both function-string formats can contain variables that get replaced with data values. What format a function string uses is determined by the type of data server and how you want Replication Server to interact with it. See "Using output templates" on page 34 for more information.

Function strings are grouped into function-string classes. Each database connection must be assigned a function-string class according to the type of destination database. Replication Server provides function-string classes that contain default function strings. Replication Server generates default function strings for Adaptive Server, DB2, Microsoft SQL Server, and Oracle function-string classes.

When you set up a replication system or add databases to the system, you should anticipate your function-string requirements and decide how you will use function-string classes and whether you need to customize function strings. See "Function-string classes" on page 21 for more information.

See "Managing function strings" on page 32 for more information about customizing function strings.

Input and output templates

Every function string uses an output template to instruct the destination database in executing the function for a specific data server.

Function strings for the rs_select and rs_select_with_lock functions use both input templates and output templates, which together perform subscription materialization and dematerialization.

You customize function strings by altering their input and output templates. You customize function strings for functions other than rs_select and rs_select_with_lock by altering only the output template. How you alter a function string depends on the function string's format-language or RPC.

See "Function-string input and output templates" on page 33 for more information about input and output templates.

Applications for customized function strings

You can customize function strings to:

- Perform operations in any native database language (including those other than Transact-SQL) by altering function-string output templates to format the commands sent to a data server.

- Materialize and dematerialize multiple subscriptions for the same replication definition with a single function string.

- Perform the following tasks by altering output templates for existing system function strings:

  - Record auditing information.

  - Execute remote procedure calls (RPCs).

  - Replicate data into multiple replicate tables in the same database.

  - Replicate data into a replicate table with a different name, column names, or column order than the primary table.

    If the replicate Replication Server is of version 11.5 or later, you can perform the same tasks more easily by creating a customized replication definition that specifies the relevant information about the replicate table. See "Creating multiple replication definitions per table" on page 265 in the *Replication Server Administration Guide Volume 1* for more information.

System functions with multiple function strings

For the class-scope system functions, each function maps to a function string within the class. Each replication-definition-scope rs_insert, rs_delete, and rs_update system function maps to a function string within the class for each replication definition.

You can create multiple function-string instances for the same replication definition for other system functions with replication-definition scope—rs_select, rs_select_with_lock, rs_datarow_for_writetext, rs_get_textptr, rs_textptr_init, and rs_writetext. In such cases, you must give each instance of a function string a different name. System functions that can take multiple function strings include:

- rs_select and rs_select_with_lock functions – used in subscription materialization and dematerialization when multiple subscriptions exist for the same replication definition. You can give each instance of the function string any name that is unique for the replication definition. Each instance of the function string corresponds to a where clause used in creating subscriptions for the replication definition.

- rs_datarow_for_writetext, rs_get_textptr, rs_textptr_init, and rs_writetext function each instance of the function string. You must name each instance of a function string for the text, unitext, or image column specified in the replication definition.

Function-string classes

Each function string belongs to a function-string class, which groups function strings intended to be used with databases of a similar type or with similar requirements. Replication Server assigns each database connection a function-string class according to the data server of the destination database.

Replication Server applies functions to the database using the function strings from its assigned function-string class. Function-string classes contain function strings for system functions and for any user-defined functions.

You can use a function-string class on multiple databases if the function strings can execute on all of the data servers. For example, a system with several databases managed by Adaptive Server can use rs_sqlserver_function_class for all the databases.

You can even use a single function-string class with heterogeneous data servers, provided that the gateways that provide access to the various data servers share a common interface.

System-provided classes

Several function-string classes are provided with Replication Server. These are called **system-provided classes**.

- rs_sqlserver_function_class – default Adaptive Server function strings are provided for this class. The default function strings in rs_sqlserver_function_class are identical to those in rs_default_function_class. rs_sqlserver_function_class is assigned by default to Adaptive Server databases you add to the replication system using rs_init.

  You can customize function strings for this class. However, this class cannot participate in function-string class inheritance. In most cases, using derived classes that specify rs_default_function_class as a parent class is preferable to using rs_sqlserver_function_class directly.

- rs_default_function_class – default Adaptive Server function strings are provided for this class. The default function strings in rs_sqlserver_function_class are identical to those in rs_default_function_class.

  You cannot customize function strings for this class. However, this class can participate in function-string class inheritance. In most cases, using derived classes that specify rs_default_function_class as a parent class is preferable to using rs_default_function_class directly.

  Note The system-provided function-string classes rs_default_function_class and rs_sqlserver_function_class contain default function strings for all system functions except rs_dumpdb and rs_dumptran. If you need to use function strings for these functions you must create them yourself in a derived class or in rs_sqlserver_function_class.

- rs_db2_function_class – DB2-specific function strings are provided for this class. See "Creating class-level translations" on page 319 in the *Replication Server Administration Guide Volume 1* for more information about using this class.

  To allow rs_db2_function_class and other function-string classes to work, issue the following commands:

  ```
  alter connection to dataserver.database
  set dsi_sql_data_style to 'db2'
  alter connection to dataserver.database
  set dsi_cmd_separator to ';'
  ```

The rs_writetext function string of rs_db2_function_class was changed to "output none." rs_db2_function_class does not support replication of text or image data. To achieve this functionality, customize the rs_writetext function string using the RPC method through a gateway.

You cannot customize function strings for this class. If you require DB2 function strings, using derived classes that specify rs_db2_function_class as a parent class is preferable, in most cases, to using this class directly.

- rs_mss_function_class – Microsoft SQL Server function strings are provided for this class. You cannot customize function strings for this class. See "Creating class-level translations" on page 319 in the *Replication Server Administration Guide Volume 1* for more information about using this class.

- rs_oracle_function_class – Oracle function strings are provided for this class. You cannot customize function strings for this class. See "Creating class-level translations" on page 319 in the *Replication Server Administration Guide Volume 1* for more information about using this class.

Table 2-1 on page 16 illustrates function-string inheritance relationships for these and other classes.

Function-string inheritance

The ability to share function-string definitions among classes by creating relationships between classes is called **function-string inheritance**.

Using function-string inheritance in general, and inheriting from system-provided classes in particular, provides both administrative and upgrade benefits to replication system administrators. Using classes that inherit from system-provided classes, you alter only the function strings you want to customize and inherit all others.

If you use classes that do not inherit from system-provided classes, you must create all function strings yourself, and add new function strings whenever you create a new table or function replication definition.

A class that inherits function strings from a parent class is called a **derived class**. A class from which a derived class inherits function strings is called the **parent class** of the derived class. Generally, you create a derived class in order to customize certain function strings and inherit all others from the parent class.

A class that does not inherit function strings from any parent class is called a **base class**. The system-provided classes rs_default_function_class and rs_db2_function_class, and any additional classes you create that do not inherit function strings from a parent class, are base classes. The system-provided classes rs_msss_function_class, rs_oracle_function_class are derived from rs_default_function_class.

A parent class can have multiple derived classes, while a derived class can have only one parent class. A derived class can also serve as the parent class for one or more derived classes. A set of derived classes of any number of levels stemming from the same base class is called a **class tree**.

The system-provided classes rs_default_function_class and rs_db2_function_class can serve as parent classes for derived classes. However, they cannot become derived classes of other parent classes.

The system-provided class rs_sqlserver_function_class cannot serve as a parent class or become a derived class.

A base class that you have created can be modified to become a derived class, or it can be designated as the parent class for a derived class. A derived class can be modified to inherit function strings from a different parent class, or it can be detached from a parent class and become a base class.

For every base class that you create, you must provide function strings for the functions that Replication Server invokes in each database to which the class is assigned. If you assign a function-string class to a database when some of the function strings for system functions are missing, the DSI reports an error when Replication Server tries to apply the function string, and suspends the database connection.

Circular function-string inheritance relationships are disallowed. That is, a parent class cannot be modified to inherit function strings from one of its own derived classes or from a derived class of one of these derived classes.

Function-string class relationships are illustrated in Figure 2-1.

Figure 2-1: Function-string class relationships
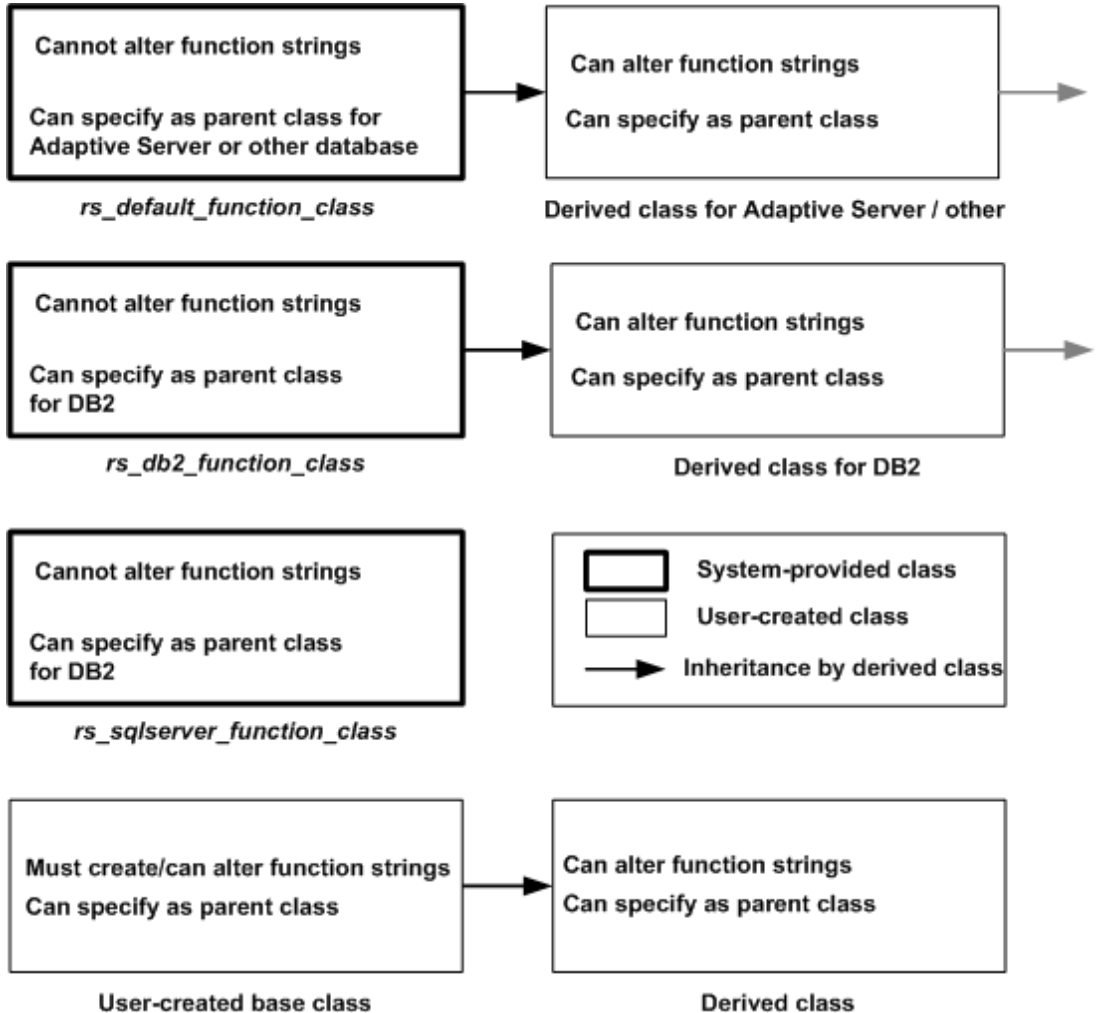
Restrictions in mixed-version systems

In a mixed-version system, only Replication Servers of version 11.5 or later can work with classes that participate in function-string inheritance.

Any class whose primary site is Replication Server version 11.0.x cannot participate in function-string inheritance. If you want to alter such a class to become a derived class or use it as a parent class, you must move that class to a primary site that is Replication Server version 11.5 or later. Then you can alter the class relationships as desired and assign the class or its derived classes to connections managed by Replication Server version 11.5 or later.

A base class that you created in Replication Server version 11.5 or later and that does not participate in function-string inheritance can be assigned to connections managed by any Replication Server in the replication system. If it is not assigned to any databases managed by Replication Server version 11.5 or later, then you can use the move primary command to assign it to a primary site managed by Replication Server version 11.0.x.

Refer to the release bulletin for more information about compatibility between Replication Servers.

Note For compatibility with Replication Servers of version 11.0.x, you may need to continue to customize function strings in rs_sqlserver_function_class. However, for databases managed by Replication Servers version 11.5 or later, using function-string inheritance and customizing function strings only in derived classes is encouraged.

Managing function-string classes

When you create or customize a function string, you specify which class it belongs to. If you want to create and use customized function strings, you can:

- Create a derived function-string class that inherits function strings from rs_default_function_class, rs_db2_function_class,or another parent class. Then, in the derived class, create only the function strings that you are interested in overriding.

  Note You cannot alter, add to, delete, or change any of the function-string classes for non-Sybase data servers.

- Create a new function-string class and create function strings for all functions.

- Customize function strings in rs_sqlserver_function_class. See "Managing function strings" on page 32 for information on this option.

Before you create customized function strings, you should decide in advance which of these approaches to take and set up your classes accordingly. Generally, it is preferable to customize function strings in derived classes rather than to customize function strings in the class rs_sqlserver_function_class. You must be using Replication Server version 11.5 or later in order to create and deploy a derived function-string class that inherits function strings from other classes.

Creating a function-string class

If function strings in an existing class do not serve your needs for particular database connections, and customizing function strings in an existing class is not feasible, you can create a new class in which to create the function strings you need. You can either:

- Create a derived class, one that inherits function strings from an existing parent class.

- Create a base class, one that does not inherit function strings from another class.

To create a derived or base function-string class and begin using it for a database connection using RCL commands, follow these steps:

1   Create the function-string class with the create function string class command, using the syntax appropriate for your task. See:

- "Creating a derived class" on page 28, or

- "Creating a base class" on page 29.

The name of the new class must conform to the rules for identifiers provided in "Identifiers" in Chapter 2, "Topics," in the *Replication Server Reference Manual*.

2   Create function strings for the new class with the create function string command, described in "Creating function strings" on page 39.

- If you are creating a derived class, you need create only the function strings that you want to override and inherit all others from the specified parent class.

- The class rs_default_function_class does not contain default function strings for the rs_dumpdb and rs_dumptran functions. If you require them in a derived class that inherits from rs_default_function_class, you must create them. See "System-provided classes" on page 22 for more information.

- If you are creating a base class, you must create all the necessary function strings for the class.

3 If you are preparing a new function-string class for an existing database connection, you must suspend the connection before you can use the new class. See "Suspending database connections" on page 171 in the *Replication Server Administration Guide Volume 1* for details.

4 Create or alter the database connection to use the new class. See "Assigning a function-string class to a database" on page 31.

5 If you altered an existing database connection to use the new class, resume the connection. See "Suspending database connections" on page 171 in the *Replication Server Administration Guide Volume 1* for details.

Creating a derived class

To create a derived function-string class that inherits function strings from a parent class, enter a command like this at the primary site of the parent:

```
create function string class
 sqlserver_derived_class
 set parent to rs_default_function_class
```

In this example, the new class sqlserver_derived_class inherits function strings from the system-provided class rs_default_function_class. You can then create function strings that override some of the inherited function strings.

You can specify as the parent class any existing class whose primary site runs Replication Server version 11.5 or later. However, you cannot specify as a parent class the system-provided class rs_sqlserver_function_class. You also cannot specify a parent class that would result in circular inheritance. See "Function-string inheritance" on page 23 for details.

If the parent class is rs_default_function_class or a function-string class for a non-Sybase data server, you can enter this command at any Replication Server with routes to the other Replication Servers where the new class will be used. This site is the primary site for the derived class and any new classes derived from it.

If the parent class is a user-created class, enter this command in the Replication Server that is the primary site for the parent class. This site is the primary site for all classes derived from the parent class.

Creating a base class

To create a base function-string class, one which does not inherit function strings from a parent class, enter a command like this:

```
create function string class base_class
```

In this example, the new class base_class does not inherit function strings from a parent class.

Enter this command at any Replication Server that has routes to the other Replication Servers where the new class will be used. This site then becomes the primary site for the class and for any derived classes for which this class serves as the parent class.

A base class can be used as a parent class for a derived class or can be modified to become a derived class.

For every base class that you create, you must provide function strings for the functions that Replication Server invokes in each database to which the class is assigned.

If you create a base class and then alter it so it becomes a derived class before actually using it with database connections, you do not have to create all the function strings.

Primary site for a function-string class

Although most function strings are executed in replicate databases, you execute the create function string class command in a Replication Server, usually a primary Replication Server, that has routes to all sites where the function-string class is to be used. This command designates that Replication Server as the primary site for the class. Function-string classes are replicated via routes, along with other replication system data.

You can only create or alter function strings that have class scope at the primary site for a class. Function strings with replication-definition scope must be created or altered at the primary site for the replication definition.

By default, the class rs_sqlserver_function_class does not have a primary site. To alter class-scope function strings for this class, you must first designate a Replication Server as a primary site for the class. To specify a site for this function-string class, execute the following command at the Replication Server that is to be the primary site:

```
create function string class
rs_sqlserver_function_class
```

After you have executed this command, you can use the move primary command to make further changes to the primary site for the function-string class.

Changing the primary site for a function-string class

Use the move primary command or Sybase Central to change the primary Replication Server for a function-string class. For example, you may need to change the primary site from one Replication Server to another so that function strings can be distributed through a new routing configuration. The new primary site must include routes to all Replication Servers where the function-string class will be used.

If you move a base class, all classes derived from that class move with it.

You cannot move the primary site for a derived class unless its parent class is a default function-string class.

Execute move primary at the Replication Server that you want to designate as the new primary site for the function-string class.

For example, the following command changes the primary site for the sqlserver2_function_class function-string class to the SYDNEY_RS Replication Server, where the command is entered:

```
move primary of function string class
  sqlserver2_function_class
  to SYDNEY_RS
```

If the class rs_sqlserver_function_class has not yet been assigned a primary site, you cannot use the move primary command to assign one. You must use the create function string class command to first designate a primary site for that class. See "Changing the primary site for a function-string class" on page 30 for details.

Assigning a function-string class to a database

You can assign a function-string class to a database connection in Sybase Central or with the create connection or alter connection commands, executed in the Replication Server that manages the database. When you add a database connection using the rs_init program, the class rs_sqlserver_function_class is assigned to the database by default.

You must suspend the connection to the database before you alter the function-string class that is assigned to the database. The set function string class clause of create connection and alter connection specifies the name of the function-string class to use with the database.

Before you can assign a function-string class to a database connection:

- The function-string class you specify must already exist and be available to the Replication Server. See "Creating a function-string class" on page 27 for more information.

- All necessary function strings must be created in the class. See "Creating function strings" on page 39 for details.

See "Creating database connections" on page 167 and "Altering database connections" on page 170 in the *Replication Server Administration Guide Volume 1* for more information about using the create connection and alter connection commands. Also refer to reference pages for these commands in the *Replication Server Reference Manual*.

Refer to the Replication Server installation and configuration guides for your platform for more information about rs_init.

Example for creating new connection

The following command creates a connection to the pubs2 database managed by the TOKYO_DS data server:

```
create connection to TOKYO_DS.pubs2
 set error class tokyo_error_class
 set function string class tokyo_func_class
 set username pubs2_maint
 set password pubs2_maint_pw
```

This command assigns the tokyo_func_class function-string class to the database connection.

Example for altering an existing connection

The following command alters an existing database connection to specify a different function-string class:

```
alter connection to TOKYO_DS.pubs2
 set function string class tokyo_func_class2
```

Dropping a function-string class

If you are sure that you will not need it again, you may want to drop a function-string class that you created from the replication system. You can drop any function-string class except the three system-provided classes and any user-created class that currently serves as a parent class. Before you can drop a function-string class, you must drop all database connections that use the function-string class, or you can alter the connections to use a different class.

Dropping a function-string class deletes all function strings defined for the class and removes all references to the class from the RSSD.

To drop a function-string class from the isql command line, use the drop function string class command. For example, the following command drops the tokyo_func_class function-string class and all of its function strings:

```
drop function string class tokyo_func_class
```

Enter this command in the Replication Server that is the primary site for the class.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about drop function string class command.

Managing function strings

Each destination Replication Server uses function strings to convert the functions to commands that are appropriate for the destination data server (such as Adaptive Server) before it submits these commands. Refer to Chapter 2, "Replication Server Technical Overview" in the *Replication Server Administration Guide Volume 1* for more information about DSI threads, the components that perform this conversion at the replicate Replication Server.

The following sections describe elements of function strings and the commands for managing them. Refer to the *Replication Server Reference Manual* for complete command syntax and permissions.

Function strings and function-string classes

If you do not require customized function strings, you can use one of the system-provided function-string classes to provide default function strings. If you require customized strings, you must use the system-provided class—rs_sqlserver_function_class—in which you can customize function strings or create a derived or base function-string class. See "Function-string classes" on page 21 for details.

- If the connection for the database in which the function will be executed uses a system-provided function-string class or a derived class that inherits directly or indirectly from rs_default_function_class or a function-string class for a non-Sybase data server, default function strings are provided for every system function and user-defined function.

- If the connection uses a user-created base function-string class (which does not inherit function strings) or a derived class that inherits from such a class, you must create function strings for every system function and user-defined function. Create them in the base class if you want them to be available in all its derived classes.

Function-string input and output templates

To customize function strings, you alter their input and/or output templates. Depending on the function, function strings may include both an input template and an output template, an output template, or neither template:

- For the rs_select and rs_select_with_lock functions, used in subscription materialization, Replication Server uses input templates to locate the function string that corresponds to a subscription's where clause.

- For all functions Replication Server uses output templates to map functions to the language commands or to apply RPC invocations at the destination data server.

Requirements for using input and output templates

When you alter templates to customize function strings, you should keep in mind the following requirements:

- Function-string input and output templates are limited to 64K bytes. The result of substituting runtime values for embedded variables in function-string input or output templates must not exceed 64K.

- Function-string input and output templates are delimited with single quotation marks (').

- Function-string variables are enclosed within a pair of question marks (?).

- A variable name and its modifier are separated with an exclamation point (!).

Language output templates involve additional related requirements. See "Using output templates" on page 34 for details.

Using output templates

You alter output templates to customize function strings. Replication Server uses output templates to determine the format of the command sent to a data server. Most output templates use one of two formats: language or RPC, corresponding to the format of the function string itself. (See "Function strings" on page 19 for information on function-string formats.) An output template for an rs_writetext function string can use the RPC format or one of the additional formats writetext or none, but not a language output template. See "Using function strings with text, unitext, image, and rawobject datatypes" on page 51 for details.

Language output templates

Language output templates contain text that the data server interprets as commands. Replication Server substitutes values for variables embedded in the output template and passes the resulting language command(s) to the data server to process.

See "Creating function strings" on page 39 for example output templates. See "Using function-string variables" on page 37 for details on embedded variables.

Within a language output template, Replication Server interprets certain characters in special ways:

- Two single quote characters (") are interpreted as one single quote

- Two question marks (??) are interpreted as one single question mark

- Two semicolons (;;) are interpreted as one single semicolon

Other than the embedded variable substitutions and these special interpretations, Replication Server does not attempt to interpret the contents of language output templates.

See "Function-string variable formatting" on page 39 for information about how Replication Server formats function-string variables when it maps function strings to data server commands.

RPC output templates

Unlike language output templates, Replication Server interprets the contents of RPC output templates. They are written in the format of the Transact-SQL execute command. Replication Server parses the output template to construct a remote procedure call to send to the Adaptive Server, Open Server gateway, or Open Server application.

RPC output templates work well with gateways or Open Servers with no language parser. RPCs are usually more compact than language requests and, since they do not require parsing by the data server, may also be more efficient. Therefore, you might choose to use an RPC even when a data server supports language requests.

Output templates for *rs_writetext* function strings

Replication Server supports three output formats for creating an rs_writetext function string: RPC, writetext, and none. The writetext and none output templates can only be used in rs_writetext function strings.

See "Using function strings with text, unitext, image, and rawobject datatypes" on page 51 for more information about writetext and none.

Using input templates

Input templates are used only for non-bulk materialization and for dematerialization with purge—those situations where Replication Server must select data to add or delete from selected tables. rs_select and rs_select_with_lock are the only function strings that can contain input templates. Replication Server determines which function string to use with a subscription during materialization or dematerialization by:

• Matching the subscription's replication definition

• Matching the input template with the where clause used in the subscription

rs_select and rs_select_with_lock also contain output templates to specify the actual select statements or other operations that perform the desired materialization or dematerialization.

For the system-provided classes, Replication Server generates default function strings for the rs_select and rs_select_with_lock functions when you create a replication definition. Generally, you only need to customize these function strings if multiple subscriptions exist for your replication definition.

Function strings for the rs_select and rs_select_with_lock functions are most often used for materialization. If you plan multiple subscriptions to the same replication definition, create the function strings before you create the subscriptions. See "Subscription materialization methods" on page 353 in the *Replication Server Administration Guide Volume 1* for more information about subscription materialization.

Function strings for rs_select and rs_select_with_lock may also be used for subscription dematerialization, which uses the where clause of the command used to create the subscription. The function strings for these functions must exist before you drop the subscriptions. See "Using the drop subscription command" on page 380 in the *Replication Server Administration Guide Volume 1* for more information about dematerialization.

An input template can contain user-defined variables whose values come from constants in the where clause of a subscription. No other types of function-string variables are allowed in input templates. An output template in the same function string can reference these user-defined variables.

If you need to customize an output template to select materialization data, you can omit the input template from an rs_select or rs_select_with_lock function string. Doing so creates a default function string that can match any select statement when no other function string's input template matches the select command.

As with other functions with replication-definition scope, you create function strings for the rs_select and rs_select_with_lock functions in the primary Replication Server where the replication definition was created.

Class in which to create function strings

When you create rs_select and rs_select_with_lock function strings for materialization, you create them in the function-string class that is assigned to the connection to the primary database from which you are selecting materialization data. If you are using bulk materialization, you do not need to create rs_select and rs_select_with_lock function strings for materialization.

When you create rs_select and rs_select_with_lock function strings for dematerialization, you create them in the function-string class that is assigned to the connection to the replicate database for which you are selecting data to be dematerialized. If you drop a subscription using drop subscription with the without purge option, you do not need rs_select and rs_select_with_lock function strings for dematerialization.

Example for *rs_select* function string

In the following example, a site subscribes to a specified publisher's book titles through the replication definition titles_rep. There must be an rs_select function string with an input template that compares the publisher column in the pubs2 database's titles table to a user-defined value that identifies the publisher.

The create function string command creates a function string with an input template that compares the publisher column pub_id to the user-defined variable ?*pub_id*!*user*?. For details on function-string variables, see "Using function-string variables" on page 37.

The input template matches any subscription with a where clause of the form where pub_id = *constant*. As a result, the output template, when it is used, includes the *constant* value. The output template selects materialization data from two different tables.

```
create function string titles_rep.rs_select;pub_id
    for sqlserver2_function_class
scan 'select * from titles where pub_id =
    ?pub_id!user?'
output language
    'select * from titles where pub_id =
    ?pub_id!user?
    union
    select * from titles.pending where pub_id =
    ?pub_id!user?'
```

See "Creating function strings" on page 39 for details. Refer to the *Replication Server Reference Manual* for complete syntax.

Using function-string variables

Variables embedded in function-string input or output templates are symbolic markers for various runtime values.

A variable can represent a column name, the name of a system-defined variable, the name of a parameter in a user-defined function, or a user-defined variable defined in an input template. The variable must refer to a value with the same datatype as anything to which it is assigned.

Function-string variables are enclosed inside of a pair of question marks (?), as shown:

```
?variable!modifier?
```

The *modifier* portion of a variable identifies the type of data the variable represents. The modifier is separated from the variable name with an exclamation (!).

The rs_truncate function string accepts position-based function string variable in the format:

```
?n!param?
```

Where *n* is a number from 1 to 255, representing the position of function parameter in the LTL. The first parameter for rs_truncate in the LTL is represented in function string as `?1!param?`. For position based function string variable, the only acceptable modifier is param.

A sample function string for rs_truncate with the position-based variable is as follows:

```
truncate table publishers partition ?1!param?
```

Replication Server recognizes the modifiers listed in Table 2-3:

Table 2-3: Function-string variable modifiers

Modifier	Description
new, new_raw	A reference to the new value of a column in a row that Replication Server is inserting or updating.
old, old_raw	A reference to the *old* values of a column in a row that Replication Server is inserting or updating.
user, user_raw	A reference to a variable that is defined in the input template of an rs_select or rs_select_with_lock function string.
sys, sys_raw	A reference to a system-defined variable.
param, param_raw	A reference to a stored-procedure parameter.
text_status	A reference to the text_status value for text, unitext, or image data. Possible values are: • 0x000 – Text field contains NULL value, and the text pointer has not been initialized. • 0x0002 – Text pointer is initialized. • 0x0004 – Real text data will follow. • 0x0008 – No text data will follow because the text data is not replicated. • 0x0010 – The text data is not replicated but it contains NULL values.

Note Function strings for user-defined functions may not use the *new* or *old* modifiers.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for a list of system-defined variables that you can use in function-string input or output templates.

See "Using the default system variable" on page 49 for information on applications for that system variable.

Function-string variable formatting

When Replication Server maps function-string output templates to data server commands, it formats the variables using the Adaptive Server format.

For most variables (except those special cases with modifiers ending in *_raw)*, Replication Server formats data as follows:

• Adds an extra single-quote character to single-quote characters appearing in character and date/time values.

• Adds single-quote characters around character and date/time values, if they are missing.

• Adds the appropriate monetary symbol (for example, the dollar sign) to values of money datatypes.

• Adds the "0x" prefix to values of binary datatypes.

• Adds a combination of a backslash (\) and newline character between existing instances of a backslash and newline character in character values. Adaptive Server treats a backslash followed by a newline as a continuation character and, therefore, deletes the added pair of characters, leaving the original characters intact.

Replication Server does not alter datatypes in these ways for modifiers that end in *_raw*.

Creating function strings

To add a function string to a function-string class, use the create function string command. Enter function-string commands at the primary site of the function string:

- For function strings with replication-definition scope, the primary site is the Replication Server where the replication definition was created.

- For function strings with class scope, the primary site is the Replication Server that is the primary site for the class. The primary site for a derived class is the same as for its parent class, unless the parent class is one of the system-provided classes. See "Primary site for a function-string class" on page 29 for more information.

If you are using a derived function-string class whose parent class is not provided by the system, you may choose to customize function strings in the parent class rather than in the derived class that is actually assigned to a particular database connection. Doing so would make the customized function strings available for any additional derived classes of that parent class.

Guidelines for creating function strings

The following guidelines for creating function strings pertain to function-string classes:

- If you need to customize function strings, you can do so in any class other than the system-provided classes rs_default_function_class and rs_db2_function_class.

- You must assign a function-string class a primary site before you can create function strings for the class. The system-provided class rs_sqlserver_function_class has no primary site until you assign one using the create function string class command.

- If the function-string class is a new base class, you must create function strings for all the necessary system functions before you can use the class.

The following guidelines pertain to function strings themselves:

- You can specify an optional name for the function string. For the rs_select, rs_select_with_lock, rs_datarow_for_writetext, rs_get_textptr, rs_textptr_init, and rs_writetext functions, Replication Server uses the function-string name to uniquely identify the function strings. Function string names are unique when you qualify them fully.

- If the input template is omitted for an rs_select or rs_select_with_lock function string, Replication Server matches any subscriptions that do not have matching function strings.

- If you are customizing function strings for functions with replication-definition scope, you must create the function strings before you create the subscriptions.

- You can put several commands in a language output template, separating them with semicolons. See "Defining multiple commands in a function string" on page 46 for details.

  Make sure that the database connection batch parameter has been set to allow command batching. See "Configuration parameters affecting individual connections" on page 173 in the *Replication Server Administration Guide Volume 1*.

- You can use Adaptive Server syntax to specify a null value for a *constant* in a function string.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for the complete syntax for the create function string command.

Example for *rs_begin* function string

The following example creates a function string for the rs_begin function that begins a transaction in the database by executing a stored procedure named begin_xact.

```
create function string rs_begin
 for gateway_func_class
 output rpc 'execute begin_xact'
```

Example for *rs_insert* function string

The following example creates a function string for a rs_insert function that references the publishers_rep replication definition, which executes an RPC at the replicate database as a result of an insert in the primary table. The stored procedure insert_publisher is defined only at the replicate database.

```
create function string publishers_rep.rs_insert
    for rs_sqlserver_function_class
    output rpc
    'execute insert_publisher
      @pub_id = ?pub_id!new?,
      @pub_name = ?pub_name!new?,
      @city = ?city!new?,
      @state = ?state!new?'
```

Altering function strings

The alter function string command replaces an existing function string. alter function string acts essentially the same as create function string except that it executes the drop function string command first. The function string is dropped and re-created in a single transaction to prevent any errors from occurring as a result of missing function strings.

You can alter a function string using either the alter function string command or the create function string command. To alter a function string using the create function string command, you must include the optional clause with overwrite after the name of the function-string class. This command drops and re-creates an existing function string, the same as the alter function string command.

To alter a function string using the alter function string command, you must first create a function string.

In a derived class, first use the create function string command to override the function string that is inherited from the parent class. You cannot alter a function string in a derived class unless the function string has been explicitly created for the derived class.

You alter function strings at the Replication Server that is the primary site for the existing function string:

- For functions of replication-definition scope, alter the function string at the primary Replication Server where the replication definition was defined.

- For functions of class scope, alter the function string at the primary site for the function-string class. The primary site for a derived class is the same as for its parent class, unless the parent class is one of the system-provided classes. See "Primary site for a function-string class" on page 29 for more information.

For system functions that allow multiple function-string mappings, such as rs_select and rs_select_with_lock, provide the complete function string name in the alter function string syntax. Replication Server uses the name to determine which function string to alter.

See "Creating function strings" on page 39 for example function strings.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for the complete syntax for the alter function string command.

Dropping function strings

To discard a customized function string in a derived class and restore the function string from the parent class, drop the function string. Use the drop function string command to remove one or more function strings in a function-string class.

Warning! If you want to drop and re-create a function string, use alter function string to replace an existing function string with a new one. Dropping and then re-creating a function string by other methods can lead to a state where the function string is temporarily missing.

If a transaction that uses this function string occurs between the time the function string is dropped and the time it is re-created, Replication Server detects the function string as missing and fails the transaction.

When you drop the function string from a derived class, you restore the function string from the parent class.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information on drop function string command.

You can also drop customized function strings from the system-provided class rs_sqlserver_function_class.

To restore a default function string for a function string with replication-definition scope that you have dropped, use the alter function string command to omit the output clause. See "Restoring default function strings" on page 44 for details.

Examples

The following command drops the rs_insert function string for the publishers_rep replication definition in the class sqlserver2_func_class:

```
drop function string
publishers_rep.rs_insert
for sqlserver2_func_class
```

The following command drops the pub_id instance of a function string for the rs_select function for the publishers_rep replication definition in the class derived_class. Drop function strings for the rs_select_with_lock function in a similar way.

```
drop function string
publishers_rep.rs_select;pub_id
for derived_class
```

The following command drops the rs_begin function string from the gateway_func_class function-string class:

```
                           drop function string rs_begin
                           for gateway_func_class
```

Dropping all function
strings for a function

In cases where there are multiple function strings for a specified function, you can drop all function strings for that function simultaneously.

The following command drops all function strings for the rs_select_with_lock function that references the publishers_rep replication definition in the class sqlserver2_func_class:

```
drop function string
publishers_rep.rs_select_with_lock;all
for sqlserver2_func_class
```

System functions that can have multiple function string mappings include the rs_select, rs_select_with_lock, rs_get_textptr, rs_textptr_init, or rs_writetext functions.

Examples of using the
all keyword as
shorthand

When dropping function strings for any system function for which you provided a lengthy name, you can use the all keyword as shorthand for the name of the function string instance. For example, the following command gives a long name for a function string:

```
create function string
publishers_rep.rs_insert;my_insert_function_string
for sqlserver2_func_class
 ...
```

In this case, the following command drops the function string without you having to enter the fully qualified name:

```
drop function string
publishers_rep.rs_insert;all
for sqlserver2_func_class
```

Restoring default function strings

To restore the Adaptive Server default function string for a system function with replication definition scope, omit the output clause in the create function string or alter function string command. You cannot omit an output template from a system function with function-string-class scope, although you can specify an empty template.

Refer to Chapter 3, "Replication Server Commands" of the *Replication Server Reference Manual*, for more information on these commands.

In all classes, even derived classes, executing the create function string or alter function string command without the output clause restores the same function string that is provided by default for the system-provided classes rs_sqlserver_function_class and rs_default_function_class.

The default function-string definition this method yields may or may not be appropriate for the databases to which you have assigned the class. This method may be most helpful when you are using a customized rs_sqlserver_function_class or when you are using other user-created base classes for Adaptive Server databases.

In a derived class, if you want to discard a customized function string and restore the function string from the parent class, drop the function string. See "Dropping function strings" on page 43 for details.

Example for *alter function string*

The following command replaces a customized rs_insert function string for the publishers_rep replication definition with the default function string:

```
alter function string publishers_rep.rs_insert
for rs_sqlserver_function_class
```

See "Altering function strings" on page 41 for details on using the alter function string command.

Example for *create function string* in a derived class

You can use this method in a derived function-string class to override an inherited function string with the Adaptive Server default function string. The following command replaces an inherited rs_insert function string for the publishers_rep replication definition with the default function string:

```
create function string publishers_rep.rs_insert
for derived_class
```

See "Creating function strings" on page 39 for details on using the create function string command.

Creating empty function strings with the output template

You can create an empty function string—one that performs no action—by including the output language clause with an empty function string specified with two single quotes.

For example, the following command defines no action for the rs_insert function string for the publishers_rep replication definition:

```
alter function string publishers_rep.rs_insert
for derived_class
output language ''
```

See "Altering function strings" on page 41 for details on using the alter function string command.

Remapping table and column names with function strings

You can use function strings to translate the table name and column names for a replicated table to names other than those specified in the replication definition. The function strings that Replication Server generates for the rs_sqlserver_function_class function-string class use the names specified by the replication definition for the table, but you can define your own function strings with any names you like.

This procedure is useful if a site has existing client applications that use different table and column names than those defined by the replication definition for the primary data. Customizing function strings allows Replication Server to maintain the data in the table and does not require that you alter the site's applications.

To do this, you can use either language function strings or RPC function strings with Adaptive Server stored procedures at the remote site.

Defining multiple commands in a function string

Language output templates can contain many commands. Adaptive Server permits multiple commands in a batch. Although most other data servers do not offer this feature, Replication Server allows you to batch commands in function strings for any data server by separating commands with a semicolon (;).

Use two consecutive semicolons (;;) to represent a semicolon that is *not* to be interpreted as a command separator.

If the data server supports command batches, Replication Server replaces the semicolons with the DSI command separator character (dsi_cmd_separator configuration parameter), as necessary, and submits the commands in a single batch.

If the data server does not support command batches, Replication Server submits each command in the function string separately.

For example, the output template in the following function string contains two commands:

```
create function string rs_commit
```

```
for sqlserver2_function_class
output language
'execute rs_update_lastcommit
    @origin = ?rs_origin!sys?,
    @origin_qid = ?rs_origin_qid!sys?,
    @secondary_qid = ?rs_secondary_qid!sys?;
    commit transaction'
```

Support for batches is enabled or disabled in Replication Server with the alter connection command.

Set batch to "on" to allow command batching for a database, or set it to "off" to send individual commands to the data server. The default is "on."

To set batching "on" for this example, enter:

```
alter connection to SYDNEY_DS.pubs2
    set batch to 'on'
```

To set batching "off," enter:

```
alter connection to SYDNEY_DS.pubs2
    set batch to 'off'
```

Using declare statements in language output templates

To include declare statements, used to define local variables, in the language output templates, make sure that the batch configuration parameter is set to "off" for the Replication Server connected to the database. When batch is set to "on" (the default), Replication Server can send multiple invocations of a function string to the data server as a single command batch, thereby putting multiple declarations of the same variable in that batch, which is unacceptable to Adaptive Server.

Performance is slower when batch mode is off because Replication Server must wait for a response to each command before the next one is sent. If your performance requirements are low, you can use declare statements in your function strings if you set batch to "off." Alternatively, if you want to use batch mode for improved performance, create function-string language output templates that execute stored procedures, which can include declare statements and other commands.

Refer to "Setting and changing parameters affecting physical connections" on page 172 in the *Replication Server Administration Guide Volume 1* for more information about batch.

Displaying function-related information

You can obtain information about existing function strings and classes in your replication system in two ways:

- Using Replication Server admin command

- Using Adaptive Server stored procedures

Refer to Chapter 3, "Replication Server Commands" of the *Replication Server Reference Manual*, for more information on admin command.

Obtaining information using the *admin* command

You can display the names of the function-string classes used in your Replication Server system using one of Replication Server's admin commands.

Use admin show_function_classes to display the names of existing function-string classes and their parent classes. It also indicates the inheritance level of the class. Level 0 is a base class such as rs_default_function_class or rs_db2_function_class, level 1 is a derived class that inherits from a base class, and so on.

For example:

```
                admin show_function_classes
Class                   ParentClass              Level
--------                ------------             -----
sql_derived_class       rs_default_function_class   1
rs_db2_derived_class    rs_db2_function_class       1
rs_db2_function_class                               0
...
```

For more information about this command, refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual*.

Obtaining information using stored procedures

You can obtain information about existing functions, function strings, and function-string classes in your system using stored procedures in a Replication Server RSSD.

Refer to Chapter 6, "Adaptive Server Stored Procedures," in the *Replication Server Reference Manual* for more information about these stored procedures.

rs_helpfunc	rs_helpfunc displays information about system functions and user-defined functions for a Replication Server or for a particular table or function replication definition. The syntax is:
	rs_helpfunc [*replication_definition* [, *function_name*]]
rs_helpfstring	rs_helpfstring displays the parameters and function-string text for functions associated with a replication definition. The syntax is:
	rs_helpfstring *replication_definition* [, *function_name*]
rs_helpclass	rs_helpclass lists all function-string classes and error classes and their primary Replication Servers. The syntax is:
	rs_helpclass [*class_name*]
rs_helpclassfstring	rs_helpclassfstring displays the function-string information for class-scope functions. The syntax is:
	rs_helpclassfstring *class_name* [, *function_name*]

Using the default system variable

The *rs_default_fs* system variable allows you to perform the following tasks:

• Extend function strings with replication-definition scope to include additional commands (such as those for auditing or tracking)

• Customize rs_update and rs_delete function strings and still be able to use the replicate minimal columns option in your replication definitions

Note Function strings containing the *rs_default_fs* system variable may only be applied on Adaptive Servers or data servers that accept Adaptive Server syntax. Otherwise, errors will occur.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for a complete list of function string system variables.

Extending default function strings

You can use the *rs_default_fs* system variable with all function strings that have replication-definition scope (table or function) as a way to extend the default function-string behavior.

Using the *rs_default_fs* system variable reduces the amount of typing required when you want to keep the functionality of the default function string intact and include additional commands. For example, you can add commands to extend the capabilities of the default function string for auditing or tracking purposes.

Commands that you add to the output language template may either precede or follow the *rs_default_fs* system variable. They may or may not affect how the row is replicated into the replicate table.

The following example shows how you might use the *rs_default_fs* system variable in the create function string command (or the alter function string command) to verify that an update has occurred:

```
create function string replication_definition.rs_update
  for function_string_class
  output language '?rs_default_fs!sys?;


if (@@rowcount = 0)
  begin
  raiserror 99999 "No rows updated!"
  end'
```

In this example, the *rs_default_fs* system variable, embedded in the language output template, maintains the functionality of the default rs_update function string while the output template then checks to see if any rows have been updated. If they have not been updated, Replication Server raises an error.

In this example, the commands that follow the system variable do not affect how the row is to be replicated at the replicate site. You can use the *rs_default_fs* system variable with similar additional commands for verification or auditing purposes.

Using *replicate minimal columns*

If you have specified replicate minimal columns for a replication definition, you normally cannot create non-default function strings for the rs_update, rs_delete, rs_get_textptr, rs_textptr_init, or rs_datarow_for_writetext system functions.

You can create non-default function strings for the rs_update and rs_delete functions by embedding the *rs_default_fs* system variable in the output language template of the create function string or alter function string commands and still use the minimal columns option.

You cannot use any variables, including the *rs_default_fs* system variable, that access non-key column values in rs_update or rs_delete function strings for replication definitions that use the minimal columns option. When you create such a function string, you may not know ahead of time which columns will be modified at the primary table. You may, however, include variables that access key column values.

See "create replication definition" in Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about the replicate minimal columns option.

Using function strings with *text*, *unitext*, *image*, and *rawobject* datatypes

In an environment that supports text, unitext, image, and rawobject datatypes, you can customize function strings for the rs_writetext function using the output template formats writetext or none. The methods discussed in this section can only be used with rs_writetext function string.

Refer to Chapter 4, "Replication Server System Functions" of the *Replication Server Reference Manual*, for more information on rs_writext function string.

For Replication Server version 11.5 or later, you can use multiple replication definitions instead of function strings. Refer to Chapter 9, "Managing Replicated Tables" in the *Replication Server Administration Volume 1* for information about multiple replication definitions.

Using *output writetext* for *rs_writetext* function strings

The writetext output template option for rs_writetext function string instructs Replication Server to use the Client-Library™ function ct_send_data to update a text, unitext, image, or rawobject column value. It specifies logging behavior for text, unitext, image, and rawobject columns in the replicate database.

writetext output templates support the following options:

- use primary log – logs the data in the replicate database, if the logging option was specified in the primary database.

- with log – logs the data in the replicate database transaction log.

- no log – does not log the data in the replicate database transaction log.

Using *output none* for *rs_writetext* function strings

The none output template option for rs_writetext function strings instructs Replication Server not to replicate a text, unitext, or image column value. This option provides necessary flexibility for using text, unitext, and image columns within a heterogeneous environment.

Heterogeneous replication and *text, unitext, image,* and *rawobject* data

To replicate text, unitext, image, and rawobject data from a foreign data server into an Adaptive Server database, you must include the text, unitext, image, and rawobject data in the replication definition so that a subscription can be created for the Adaptive Server database. However, you might not want to replicate the text, unitext, image, and rawobject data into other replicate data servers, whether they are other foreign data servers or other Adaptive Servers.

With the none output template option, you can customize rs_writetext function strings to map operations to a smaller table at a replicate site and to instruct the rs_writetext function string not to perform any text, unitext, image, or rawobject operation against the replicate site.

There is one rs_writetext function string for each text, unitext, image, and rawobject column in the replication definition. If you do not want to replicate a certain text, unitext, image, or rawobject column, customize the rs_writetext function string for that column. Specify the column name in the create or alter function string command, as shown in the example below. You may also need to customize the rs_insert function string.

Example
Assume that a replication definition does not allow null values in a text, unitext, image, or rawobject column and that you do not require certain text, unitext, image, or rawobject columns at the replicate site.

If inserts occur in those columns at the primary site, you must customize the rs_writetext function strings for the text, unitext, image, or rawobject columns that are not needed at the replicate site. You must also customize the rs_insert function string for the replication definition.

For example, assume that you have primary table foo:

```
foo (int a, b text not null, c image not null)
```

In foo, you perform the following insert:

```
insert foo values (1, "111111", 0x11111111)
```

By default, Replication Server translates rs_insert into the following form for application by the DSI thread into the replicate table foo:

```
insert foo (a, b, c) values (1, "", "")
```

The DSI thread calls:

- ct_send_data to insert text data into column b

- ct_send_data to insert image data into column c

Because null values are not allowed for the text column b and the image column c, the DSI thread shuts down if the replicate table does not contain either column b or column c.

If the replicate table only contains columns a and b, you need to customize the rs_writetext function for column c to use output none, as follows:

```
alter function string foo_repdef.rs_writetext;c
    for rs_sqlserver_function_class
    output none
```

You must specify the column name (c in this example) as shown to alter the rs_writetext function string for that column.

If the replicate table only contains columns a and b, you also need to customize the rs_insert function string for the replication definition so that it will not attempt to insert into column c, as follows:

```
alter function string foo_repdef.rs_insert
    for rs_sqlserver_function_class
    output language
    'insert foo (a, b) values (?a!new?, "")'
```

You do not have to customize rs_insert if the replication definition specifies that null values are allowed for column c. By default, rs_insert does not affect any text, unitext, or image columns where null values are allowed.

Managing Warm Standby Applications

This chapter describes one way to create and manage a warm standby application using Replication Server.

Topic	Page
Overview	56
What information is replicated?	61
Setting up warm standby databases	73
Replicating the master database in a warm standby environment	85
Switching the active and standby databases	87
Monitoring a warm standby application	94
Setting up clients to work with the active data server	97
Altering warm standby database connections	99
Warm standby applications using replication	104
Using replication definitions and subscriptions	111
Loss detection and recovery	122

This chapter describes how to set up and configure a warm standby application between two Adaptive Server databases—the primary or active database and a single standby database. Changes to the primary database are copied directly to the warm standby database. To change or qualify the data sent, you must add table and function replication definitions.

You can also use multisite availability (MSA) to set up a warm standby application between Adaptive Server databases. MSA enables replication to multiple standby and replicate databases. You can choose whether to replicate the entire database or replicate (or not replicate) specified tables, transactions, functions, system stored procedures, and data definition language (DDL). See Chapter 12, "Managing Replicated Objects Using Multisite Availability," in the *Replication Server Administration Guide Volume 1* for information about setting up a warm standby application using MSA.

Overview

A **warm standby application** is a pair of Adaptive Server databases, one of which is a backup copy of the other. Client applications update the **active database**; Replication Server maintains the **standby database** as a copy of the active database.

If the active database fails, or if you need to perform maintenance on the active database or on the data server, a switch to the standby database allows client applications to resume work with little interruption.

To keep the standby database consistent with the active database, Replication Server reproduces transaction information retrieved from the active database's transaction log. Although replication definitions facilitate replication into the standby database, they are not required. Subscriptions are not needed to replicate data into the standby database.

How a warm standby works

Figure 3-1 illustrates the normal operation of an example warm standby application.

Figure 3-1: Warm standby application
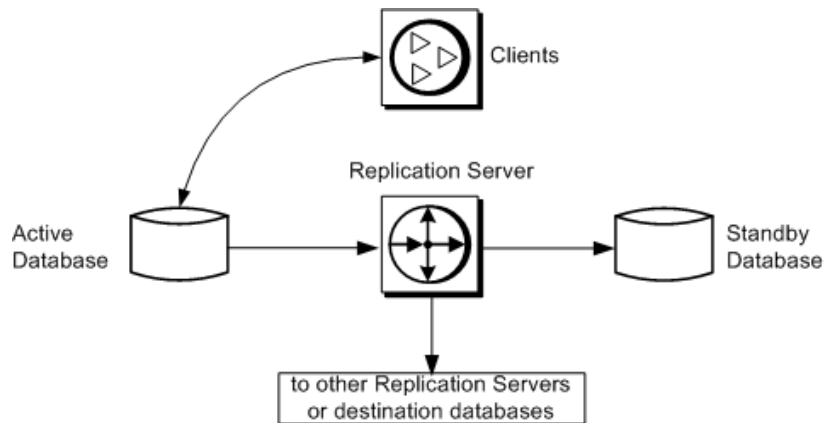
In this warm standby application:

• Client applications execute transactions in the active database.

- The RepAgent for the active database retrieves transactions from the transaction log and forwards them to Replication Server.

- Replication Server executes the transactions in the standby database.

- Replication Server may also copy transactions to destination databases and remote Replication Servers.

  See Figure 3-4 on page 88 for more details about the components and processes in a warm standby application.

Database connections in a warm standby application

In a warm standby application, the active database and the standby database appear in the replication system as a connection from the Replication Server to a single logical database. The replication system administrator creates this **logical connection** to establish one symbolic name for both the active and standby databases.

Thus, a warm standby application involves three database connections from the Replication Server:

- A physical connection for the active database

- A physical connection for the standby database

- A logical connection for the active and standby databases

Replication Server maps the logical connection to the currently active database and copies transactions from the active to the standby database.

See "Setting up warm standby databases" on page 73 for details on creating the logical and physical database connections. See Chapter 7, "Managing Database Connections" in the *Replication Server Administration Guide Volume 1* for more information about physical database connections.

Primary and replicate databases and warm standby applications

In many Replication Server applications:

- A primary database is the source of data that is copied to other databases through the use of replication definitions and subscriptions.

- A destination database receives data from the primary (source) database.

Replication Server treats a logical database like any other database. Depending on your application, the logical database in a warm standby application may function as:

- A primary database, or

- A replicate database, or

- A database that does not participate in replication

See "Switching the active and standby databases" on page 87 for more information about warm standby applications that do not participate in standard replication.

See "Warm standby applications using replication" on page 104 for more information about warm standby applications for primary or replicate databases.

Comparison of database relationships

In most of this book, databases are defined as "primary" or "replicate." In discussing warm standby applications, however, databases are also defined as "active" or "standby." Table 3-1 explains the difference.

Table 3-1: Active and standby vs. primary and destination databases

Active and standby databases	Primary and replicate databases
The active and standby databases must be managed by the same Replication Server.	Primary and destination databases may be managed by the same or different Replication Servers.
The active and standby databases must be Adaptive Server databases.	Except where they participate in warm standby applications, primary and destination databases need not be Adaptive Server databases.
The active database has one standby database. Information is always copied from the active to the standby database.	A primary database can have one or more destination databases. Some databases contain both primary and copied data.
The use of replication definitions is optional. Subscriptions are not used.	Replication definitions and subscriptions are required for replication from a primary to a destination database.
The connection to the standby database uses the function-string class rs_default_function_class. You cannot customize function strings for this class.	The connection to a replicate database can use a function-string class in which you can customize function strings. For example, it may use a derived class that inherits function strings from rs_default_function_class.
You can switch the roles of the active and standby databases.	You cannot switch the roles of primary and replicate databases.

Active and standby databases	Primary and replicate databases
Client applications generally connect to the active database. (However, you can perform read-only operations at the standby database.) No mechanism is provided for switching client applications when you switch the Replication Server to the standby database.	Client applications can connect to either primary or destination database. Only primary data can be directly modified. Generally, client applications do not need to switch between primary and destination databases.
The RepAgent for the active database submits all transactions on replicated tables, including maintenance user transactions, to the Replication Server, which reproduces them in the standby database. In a warm standby application for a destination database, transactions in the active database are normally executed by the maintenance user.	In most applications, RepAgent does not submit maintenance user transactions to the Replication Server to be reproduced in destination databases. The maintenance user does not generally execute transactions in primary databases.

Warm standby requirements and restrictions

The following restrictions apply to all Replication Server warm standby applications:

- You must use a Sybase Adaptive Server that supports warm standby applications. Refer to your release bulletin for more information.

- One Replication Server manages both the active and standby databases. Both the active and standby databases must be Adaptive Server databases.

- You cannot create a standby database for the RSSD. You can only create a standby database for the master database if the Adaptive Server supports master database replication.

- Replication Server does not switch client applications to the standby database. See "Setting up clients to work with the active data server" on page 97 for more information.

- You should run Adaptive Server for the active and standby databases on different machines. Putting the active and standby databases on the same data server or hardware resources undermines the benefits of the warm standby feature.

- Although Adaptive Server allows tables that contain duplicate rows, tables in the active and standby databases should have unique values for the primary key columns in each row.

- Failover support is not a substitute for warm standby. While warm standby keeps a copy of a database, Failover support accesses the same database from a different machine. Failover support works the same for connections from Replication Server to warm standby databases.

  For more detailed information about how Sybase Failover works in Adaptive Server, refer to *Using Sybase Failover in a High Availability System*, which is part of the Adaptive Server Enterprise version 15.0 documentation set.

  For more detailed information about how Failover support works in Replication Server, see "Configuring the replication system to support Sybase Failover" in Chapter 7, "Replication System Recovery".

- The commands and procedures for abstract plans are replicated, except for the following:

  - The and set *@plan_id* clause of create plan is not replicated. For example, this command is not replicated as shown.

    ```
    create plan "select avg(price)
    from titles" "(t_scan titles)
    into dev_plans and set @plan_id
    ```

    Rather, it is replicated as:

    ```
    create plan "select avg(price)
    from titles" "(t_scan titles)
    into dev_plans
    ```

  - The abstract plan procedures that take a plan ID as an argument (sp_drop_qplan, sp_copy_qplan, sp_set_qplan) are not replicated.

  - The set plan command is not replicated.

Function strings for maintaining standby databases

Replication Server uses the system-provided function-string class rs_default_function_class for the standby DSI, which is the connection to the standby database. Replication Server generates default function strings for this class. You cannot customize the function strings in the class rs_default_function_class.

What information is replicated?

Replication Server supports different methods for enabling replication to the standby database. The level and type of information that Replication Server copies to the standby database depends on the method you choose.

You must choose one of these two methods:

- Use the sp_reptostandby system procedure to mark the entire database for replication to the standby database. sp_reptostandby enables replication of data manipulation language (DML) commands and a set of supported data definition language (DDL) commands and system procedures.

  - DML commands, such as insert, update, delete, and truncate table, change the data in user tables.

  - DDL commands and system procedures change the schema or structure of the database.

  sp_reptostandby allows replication of DDL commands and procedures that make changes to system tables stored in the database. You can use DDL commands to create, alter, and drop database objects such as tables and views. Supported DDL system procedures affect information about database objects. They are executed at the standby database by the original user.

- If you choose not to use sp_reptostandby, you can mark individual user tables for replication with sp_setreptable. This procedure enables replication of DML operations for the marked tables.

Optionally, you can also tell Replication Server which user stored procedures to replicate to the standby database:

- If you use Adaptive Server version 11.0.x, you can copy the execution of user stored procedures to the standby database by marking them with the sp_setrepproc system procedure. Normally, only stored procedures associated with function replication definitions are replicated to standby databases.

  Refer to "Using sp_setrepproc to copy user stored procedures" on page 68 for more information.

Comparing replication methods

Table 3-2 compares sp_reptostandby and sp_setreptable, detailing how each copies information to the standby database. Many of these issues are discussed in detail later in the chapter.

Table 3-2: Comparison of table replication methods

sp_reptostandby	sp_setreptable
Copies all user tables to the standby database.	Lets you choose which user tables are copied to the standby database.
Allows replication of DML commands and supported DDL commands and system procedures. Supported DDL operations are listed in "Supported DDL commands and system procedures" on page 64.	Allows replication of DML commands executed on marked tables. **Note** Supported DDL operations can be replicated for an isql sessions. Refer to "Forcing replication of DDL commands to the standby database" on page 72 for more information.
Does not copy DML and DDL operations to replicate databases. If the warm standby application also copies data to a replicate database, you must mark tables to be copied to the replicate database with sp_setreptable.	Copies DML operations to standby and replicate databases.
Copies execution of the truncate table command to the standby database. No subscription is needed. **Note** You can enable or disable replication of truncate table to standby databases with the alter logical connection command. See "Replicating truncate table to standby databases" on page 101 for more information.	If you use Adaptive Server databases, copies execution of truncate table to standby databases. No subscription is needed.
Replication Server uses table name and table owner information to identify a table at the standby database.	If you include the owner_on keywords when you mark a table for replication to the warm standby, Replication Server uses table name and table owner information to identify a table at the standby database. If you include the owner_off keywords when you mark a table for replication to the warm standby, Replication Server uses the table name and "dbo" to identify a table at the standby database.

sp_reptostandby	sp_setreptable
By default, text, unitext, image, and rawobject columns are copied to the standby database only if changed.	By default, text, unitext, and image columns are always copied to the standby database.
If you mark the database tables with sp_reptostandby and sp_setreptable, text, unitext, image, and rawobject data may be treated in a different way. Refer to "Replicating text, unitext, image, and rawobject data" on page 70 for more information.	If you set the replication status with sp_setrepcol, text, unitext, image, and rawobject columns are treated as marked: always_replicate, replicate_if_changed, or do_not_replicate.
The easiest method to use when the active and standby databases are identical. Replication definitions are not required, but can be used to optimize performance.	Replication definitions are not required, but can be used to optimize performance.

Using *sp_reptostandby* to enable replication

Use sp_reptostandby to copy DML and supported DDL commands for all user tables to the standby database.

Restrictions and requirements when using *sp_reptostandby*

Consider the following issues when you set up your warm standby application and enable replication with sp_reptostandby.

- Both the active and standby databases must be managed by Adaptive Servers and must support RepAgent. Both databases must have the same disk allocations, segment names, and roles. Refer to the *Adaptive Server Enterprise System Administration Guide* for details.

- The active database name must exist in the standby server. Otherwise, replication of commands or procedures containing the name of that database will fail.

- Replication Server does not support replication of DDL commands containing local variables. You must explicitly define site-specific information for these commands.

- Login information is not replicated to the standby database. Refer to "Making the server user's IDs match" on page 82 for information about adding login information to the destination Replication Server.

- Some commands not copied to the standby database include:

    - select into

    - update statistics

- Database or configuration options such as sp_dboption and sp_configure

The following section lists the DDL commands, Transact-SQL commands and Adaptive Server system procedures, that Replication Server reproduces at the standby database when you enable replication with sp_reptostandby. An asterisk marks those commands and stored procedures whose replication is supported for Adaptive Server 12.5 and later.

Supported DDL commands and system procedures

alter encryption key
alter table
alter key
create default
create encryption key
create index
create key
create plan*
create procedure
create rule
create schema*
create table
create trigger
create view
drop default
drop encryption key
drop index
drop procedure
drop rule
drop table
drop trigger
drop view
grant
installjava*
remove java*
revoke
sp_addalias
sp_addgroup
sp_addmessage
sp_add_qpgroup*
sp_adduser
sp_addtype

sp_bindefault
sp_bindmsg
sp_bindrule
sp_changegroup
sp_chgattribute
sp_commonkey
sp_config_rep_agentsp_dropalias
sp_drop_all_qplans*
sp_dropgroup
sp_dropkey
sp_dropmessage
sp_drop_qpgroup*
sp_droptype
sp_dropuser
sp_encryption
sp_export_qpgroup*
sp_foreignkey
sp_import_qpgroup*
sp_primarykeysp_procxmode
sp_recompile
sp_rename
sp_rename_qpgroup*
sp_setrepcol
sp_setrepdefmode
sp_setrepmode
sp_setrepproc
sp_setreptable
sp_unbindefault
sp_unbindmsg
sp_unbindrule

If the database is the master database, the DDL commands and system procedures that are supported for replication in a user database are not supported for replication in the master database.

In the master database, the supported DDL commands and system procedures are:

alter role
create role
drop role
grant role
revoke role
sp_addlogin

> sp_displaylevel
> sp_droplogin
> sp_locklogin
> sp_modifylogin
> sp_password
> sp_passwordpolicy
> sp_role

If a DDL command or system procedure contains password information, the password information is sent through the replication environment using the ciphertext password value stored in the source Adaptive Server system tables.

To enable replication of DML and DDL commands, execute sp_reptostandby in the Adaptive Server that manages the active database. The syntax is:

> sp_reptostandby *dbname*, [[, 'L1' | 'ALL' | 'NONE'] [, use_index]]

where *dbname* is the name of the active database and the keywords L1, all, and none set the level of replication support.

L1 represents the level of replication supported by Adaptive Server version 12.5.

Use the all keyword to make sure that schema replication support is always at the highest level available. For example, to set the schema replication support level to that of the latest Adaptive Server version, log in to Adaptive Server and execute this command at the isql prompt:

```
sp_reptostandby dbname, 'all'
```

Then, if the database is upgraded to a later Adaptive Server version with a higher level of replication support, all new features of that version are enabled automatically. Refer to Chapter 5, "Adaptive Server Commands and System Procedures," in the *Replication Server Reference Manual* for more information about sp_reptostandby command.

Replicating *alter table*: limitations

When Adaptive Server performs an alter table ... add *column_name* default ... statement, the server creates a constraint for the default value using the objid. After Replication Server replicates this statement, the standby Adaptive Server creates the same constraint but with a different objid.

If the constraint is later dropped at the primary using alter table ... drop constraint ... , the statement cannot be performed at the warm standby because the objid is not the same.

To drop the constraint at both the primary and standby databases, use either of these two methods:

- Execute this statement at the primary:

```
alter table table_name
    ...
    replace column_name default null
```

- Execute this statement at the primary:

```
alter table table_name
    ...
    drop constraint constraint_name
```

This statement causes the DSI to shut down. Execute the same command at the standby database with its corresponding objid, and then resume the connection to the DSI, skipping a transaction.

Replicating the master database: limitations

The user tables and user stored procedures are not replicated if the database used is the master database.

If the master database is replicated, the following system procedures must be executed in the master database:

sp_addlogin
sp_displaylevel
sp_droplogin
sp_locklogin
sp_modifylogin

Both the source and target Adaptive Servers must support the master database replication feature if the database used is the master database.

If the database is the master database, both the source ASE server and the target ASE server must be the same hardware architecture type (32-bit versions and 64-bit versions are compatible) and the same operating system (different versions are also compatible).

Disabling replication

To turn off data and schema replication, log in to Adaptive Server and enter this command at the isql prompt:

```
sp_reptostandby dbname,'none'
```

When replication is turned off, Adaptive Server locks all user tables in exclusive mode and saves information about each of them. This process may take some time if there are a large number of user tables in the database.

Use this procedure *only* if you are disabling the warm standby application.

Note If you want to turn off replication for the current isql session only, use the set replication command. See "Changing replication for the current isql session" on page 72 for more information. Also, if the database is marked for replication to use indexes on text, unitext, image, and rawobject columns, the above command also drops indexes for replication on tables not explicitly marked for replication.

Using *sp_setreptable* to enable replication

Use sp_setreptable to mark individual tables for replication to replicate or replicate and standby databases. Replication Server copies DML operations on those tables to the standby and replicate databases.

Use sp_setreptable to mark tables for replication to the standby database if:

*   You use Adaptive Server databases, or
*   You choose not to use sp_reptostandby.

Using sp_setreptable maintains data, but not schema, consistency between the active and standby databases. sp_setreptable normally does not copy supported DDL commands and procedures to the standby database. You can, however, use the set replication command to force replication of DDL commands for the current isql session. Refer to "Changing replication for the current isql session" on page 72 for more information about set replication.

If the database is the master database, user tables are not replicated.

Using *sp_setrepproc* to copy user stored procedures

To copy the execution of a user stored procedure to the standby database, mark the stored procedure for replication with sp_setrepproc. Procedures marked with sp_setrepproc are also reproduced at replicate databases if subscriptions have been created for them.

There are two possible scenarios for stored procedure execution in warm standby applications:

*   If you have marked the stored procedure for replication with sp_setrepproc, Replication Server copies execution of the procedure to the standby database. It does *not* copy the effects of the stored procedure to the standby database.

*   If you have not marked the stored procedure for replication, Replication copies DML changes effected by the procedure to the standby database, if the affected tables have been marked for replication.

See Chapter 10, "Managing Replicated Functions" in the *Replication Server Administration Guide Volume 1* for more information about the sp_setrepproc system procedure.

If the database is the master database, user procedures are not replicated.

Replicating tables with the same name but different owners

Adaptive Server and Replication Server allow you to replicate tables with the same name but different owners.

When you mark a database for replication with sp_reptostandby, updates are copied automatically to the table of the same name and owner in the standby database.

When you mark a table for replication using sp_setreptable, you can choose whether the table owner name is used to select the correct table in the standby database.

*   If you set owner_on, Replication Server sends the table name and table owner name to the standby database.

*   If you set owner_off, Replication Server sends the table name and "dbo" as the owner name to the standby database.

Note If you are copying information to a replicate database and have used sp_setreptable to set owner_off, Replication Server sends the table name to the replicate database. It does not send owner information.

Refer to "Enabling replication with owner_on status" on page 271 in the *Replication Server Administration Guide Volume 1* for syntax and other information about using sp_setreptable to set owner status.

Note If you mark a table with a non-unique name for replication and then create a replication definition for it, you must include owner information in the replication definition. Otherwise, Replication Server will be unable to find the correct table in the replicate or standby database.

Replicating *text*, *unitext*, *image*, and *rawobject* data

If a database is marked with sp_reptostandby, the replication status is automatically replicate_if_changed, and Adaptive Server logs only text, unitext, image, and rawobject columns that have been changed. This ensures that the standby database stays in sync with the active database. You cannot change the replication status of such a table using sp_setrepcol.

If a table is marked for replication with sp_setreptable, the default replication status is always_replicate, and Adaptive Server logs all text, unitext, image, and rawobject column data. You can change the replication status of text, unitext, image, and rawobject columns in tables marked with sp_setreptable. Use sp_setrepcol to change the replication status to replicate_if_changed or do_not_replicate. A column or combination of columns must uniquely identify each row.

If you use replication definitions, the primary key must be a set of columns that uniquely identify each row in the table. You have to make sure that the replication status is the same at the Adaptive Server and the Replication Server. If the replication status differs, you must resolve the inconsistencies. See "Resolving inconsistencies in replication status" on page 284 in the *Replication Server Administration Guide Volume 1* for more information.

Using the *use_index* option in a replicate database

The use_index option is used to speed up the process of setting the text, unitext, image, or rawobject columns for replication. It is specially useful for large tables containing one or more text, unitext, image, or rawobject columns. You can set use_index option at a database level, table level, or column level. For example, a table can be marked without using indexes, but you can explicitly mark only one column to use an index for replication.

When you use the use_index option with sp_reptostandby, the database is marked to use indexes on text, unitext, image, or rawobject columns, and internal indexes are created on tables that are not explicitly marked for replication.

For a database marked for replication to use indexes, if a new table with off-row columns is created, the indexes for replication are created as well. Similarly, when an alter table...add column command is executed in a database marked to use indexes, an internal index is created in the new off-row column. With the alter table...drop column command, if the column being dropped is marked to use an index, the internal index for replication is dropped as well.

The replication index status at different object levels is in this order: column, table, and database. If the database is marked to use indexes for replication, but you marked a table without using indexes, the table status overrides the database status.

Note The replication performance on off-row (text, unitext, image, or rawobject) columns does not change. Only the process of marking a database, table or column for replication is affected.

You can use the use_index option if the table has a large number of rows or if the database has one or more tables with a considerable number of rows and several off-row columns.

Replicating encrypted columns

For considerations when working with encrypted columns, see "Replicating encrypted columns" in Chapter 9 "Managing Replicated Tables" in the *Replication Server Administration Guide Volume 1*.

When warm standby involves a replicate database

You can copy information from an active database to a standby database and also copy information from the active database to a replicate database. Replication Server must copy a table's text, unitext, image, and rawobject columns to the standby and replicate databases with the same replication status.

Do not change the replication status for the table if you want to copy all text, unitext, image, and rawobject columns to the standby and replicate databases. By default, all text, unitext, image, and rawobject columns are copied to standby and replicate databases.

If you want to copy only text, unitext, image, and rawobject columns that have changed, use sp_setrepcol to set the replication status to replicate_if_changed.

Changing replication for the current *isql* session

You can use set replication to control replication of DML and/or DDL commands and procedures for an isql session.

Execute set replication at the Adaptive Server that manages the active database. The syntax is:

```
set replication [on | force_ddl | default | off]
```

The default setting is "on." Default behavior depends on whether or not the database has been marked for replication with sp_reptostandby. Table 3-3 describes the default behavior of set replication.

Table 3-3: Default behavior of set replication

If the database has been marked for replication with sp_reptostandby	If the database has not been marked for replication with sp_reptostandby
Replication Server copies DML and supported DDL commands to the standby database for all user tables.	Replication Server copies DML commands to standby and replicate databases for tables marked with sp_setreptable.

Some examples of set replication follow. See Chapter 5, "Adaptive Server Commands and System Procedures" in the *Replication Server Reference Manual* for more information about set replication command.

Forcing replication of DDL commands to the standby database

To force replication of all supported DDL commands and system procedures for an isql session, enter:

```
set replication force_ddl
```

This command enables replication of DDL commands and system procedures for tables marked with sp_setreptable.

To turn off force_ddl and return set replication to default status, enter:
```
set replication default
```

Turning off all replication to the standby database

To turn off all replication to the standby database for an isql session, enter:

```
set replication off
```

Setting up warm standby databases

Setting up databases for a warm standby application involves three high-level tasks. Each of these tasks may include additional tasks.

1   Create a single logical connection that will be used by both the active and standby databases.

2   Use Sybase Central or rs_init to add the active database to the replication system. You do not need to add the active database if you have designated as the active database a database that was previously added to the replication system.

3   Use sp_reptostandby or sp_setreptable to enable replication for tables in the active database.

4   Use Sybase Central or rs_init to add the standby database to the replication system, then initialize the standby database.

Before you begin

Before setting up the databases, note that:

•   The Replication Server that manages the active and standby databases must be installed and running. A single Replication Server manages both the active and the standby database.

•   The Adaptive Servers that contain the active and standby databases must be installed and running. Ideally, these databases should be managed by data servers running on different machines.

•   Before you can add a database to the replication system as an active or standby database, it must already exist in the Adaptive Server.

See "Warm standby requirements and restrictions" on page 59 for additional information.

Client application issues

Depending on your client applications and your method of initializing the standby database, you may be suspending transaction processing in the active database until you have initialized the standby database.

If you do not suspend transaction processing, ensure that Replication Server has sufficient stable queue space to hold the transactions that execute while you are loading data into the standby database.

Before you set up the warm standby databases, you should have decided on and implemented a mechanism for switching client applications to the new active database. See "Setting up clients to work with the active data server" on page 97 for more information.

Task one: Creating the logical connection

This section explains how to create the logical connection for the warm standby application. See "Database connections in a warm standby application" on page 57 for more information about logical connections.

This section also explains how to reconfigure RepAgent for the active database, which you must do if the active database was already part of the replication system.

Naming the logical connection

When you create the logical connection, use the combination of logical data server name and logical database name, in the form *data_server.database*.

There are two methods for naming the logical connection:

- *If the active database has not yet been added to the replication system* – use a different name for the logical connection than for the active database. Using unique names for the logical and physical connections makes switching the active database more straightforward.

- *If the active database has previously been added to the replication system* – use the *data_server* and *database* names of the active database for the logical connection name. The logical connection inherits any existing replication definitions and subscriptions that reference this physical database.

When you create a replication definition or subscription for a warm standby application, specify the logical connection instead of a physical connection. Specifying the logical connection allows Replication Server to reference the currently active database.

See "Warm standby applications using replication" on page 104 for more information. Also see "Using replication definitions and subscriptions" on page 111.

Procedure for creating the logical connection

Follow these steps to create the logical connection:

1   Using a login name with sa permission, log in to the Replication Server that will manage the warm standby databases.

2   Execute the create logical connection command:

        create logical connection to *data_server.database*

The data server name can be any valid Adaptive Server name, and the database name can be any valid database name.

Reconfiguring and restarting RepAgent

If you designate as the active database a database that was previously added to the replication system, the RepAgent thread for the active database shuts down when you create the logical connection.

1   Reconfigure RepAgent with sp_config_rep_agent, setting the send_warm_standby_xacts configuration parameter.

2   Restart RepAgent.

For information about how to configure and start RepAgent, refer to "Setting up RepAgent" on page 112 in the *Replication Server Administration Guide Volume 1*. Refer to the *Replication Server Reference Manual* for more information about the sp_config_rep_agent system procedure.

Task two: Adding the active database

To add a database to the replication system as the active database for a warm standby application, rs_init, as described in the Replication Server installation and configuration guides for your platform. Perform the steps described for adding a database to the replication system.

Task three: Enabling replication for objects in the active database

You can enable replication for tables in the active database in either of two ways:

- Use sp_reptostandby to mark the database for replication, enabling replication of data and supported schema changes.

- Use sp_setreptable to mark individual tables for replication of data changes.

Refer to "What information is replicated?" on page 61 for more information about these commands.

1   Log in to the Adaptive Server as the System Administrator or as the Database Owner, and:

```
use active_database
```

2   Mark database tables for replication, using one of these methods:

- Mark all user tables by executing the sp_reptostandby system procedure:

```
sp_reptostandby dbname, [ 'L1' | 'all' ]
```

where *dbname* is the name of the active database, L1 sets the replication level to that of Adaptive Server version 11.5, and all sets the replication level to the current version of Adaptive Server. This method replicates both DML and DDL commands and procedures.

- Mark all user tables by executing sp_reptostandby with the use_index option:

```
sp_reptostandby dbname, [[, 'L1' | 'ALL'][,
use_index]]
```

where *dbname* is the name of the active database. With the use_index option, the database is marked to use indexes on text, unitext, image, or rawobject columns, and internal indexes are created on those tables not explicitly marked for replication.

- Mark individual user tables by executing the sp_setreptable system procedure for each table that you want to replicate into the standby database:

```
sp_setreptable table_name, 'true'
```

where *table_name* is the name of the table. This method replicates DML commands.

3   If you are using the replicated functions feature described in Chapter 10, "Managing Replicated Functions," in the *Replication Server Administration Guide Volume 1*, execute the following system procedure for every stored procedure whose executions you want to replicate into the standby database:

```
sp_setrepproc proc_name, 'function'
```

4   If you are using replicated stored procedures associated with table replication definitions, as described in Appendix A, "Asynchronous Procedures," execute the following system procedure for every such stored procedure whose executions you want to replicate into the standby database:

```
sp_setrepproc proc_name, 'table'
```

Enabling replication for objects added later

Later on, you may add new tables and user stored procedures that you want to replicate to the standby database.

•   If you marked the database for replication with sp_reptostandby, new tables are automatically marked for replication.

•   If you marked database tables for replication to the standby database with sp_setreplicate, you must mark each new table that you want to replicate with sp_setreplicate.

•   You must mark each new user stored procedure that you want to replicate with sp_setrepproc.

Task four: Adding the standby database

Use rs_init to add the standby database and its RepAgent to the replication system, then you initialize it with data from the active database.

This section explains how to add the standby database to the replication system and prepare it for operation.

This section also describes enabling replication for objects in the standby database and granting permissions to the maintenance user in the standby database. Whether or not you need to perform these steps depends on your method for initializing the standby database.

Before you add the standby database:

1   Create the standby database, if it does not already exist.

2   Determine how to initialize the standby database.

3   Add the standby database maintenance user—if you are using dump and load to initialize the standby database.

4   Online the new database using the online database clause before replicating.

Creating the standby database

If it does not already exist, you must create the standby database in the appropriate Adaptive Server, according to your needs.

Refer to the *Adaptive Server Enterprise System Administration Guide* for details on creating databases.

Determining how to initialize the standby database

You initialize the standby database with data from the active database. To do this, use these Adaptive Server commands and utilities:

• dump and load, or

• bcp, or

• quiesce database ... to *manifest_file* and mount.

Replication Server writes an "enable replication" marker into the active database transaction log when you add the standby database using Sybase Central or rs_init. Adaptive Server writes a dump marker into the active database transaction log when you perform a dump operation—either a dump database or a dump transaction.

If you do not suspend transaction processing during initialization:

• Choose the "dump marker" option in Sybase Central or rs_init, and use the dump and load commands.

If you suspend transaction processing during initialization:

• Do not choose the "dump marker" option in Sybase Central or rs_init, and use the dump and load commands, or

• Use bcp, or

• Use quiesce database ... to *manifest_file* and mount.

The target database cannot be materialized with dump or load if the database used is the master database. You may use other methodologies such as bcp where the data can be manipulated to resolve inconsistencies.

Table 3-4 summarizes each of the initialization methods and the role of these markers.

Table 3-4: Issues in initializing the standby database

Issue	Use dump and load with "dump marker"	Use dump and load without "dump marker"	Use bcp	Use mount
Working with client applications.	Use if you can not suspend transaction processing for client applications.	Use if you can suspend transaction processing for client applications.		Use if you can suspend transaction processing for client applications.
When does Replication Server begin replicating into the standby database?	Replication Server starts replicating into the standby database from the first dump marker after the enable replication marker.	Replication Server starts replicating into the standby database from the enable replication marker.		Replication Server starts replicating into the standby database from the enable replication marker.
Creating maintenance user login names and making sure all user IDs match.	Add the login name for the standby database maintenance user in both the active Adaptive Server and the standby Adaptive Server, and ensure that the server user's IDs match. (You create login names in the active Adaptive Server because using dump and load to initialize the standby database with data from the active database overrides any previous contents of the standby database with the contents of the active database.)		When you add the standby database, Sybase Central or rs_init adds the maintenance user login name and user in the standby Adaptive Server and the standby database.	Add the login name for the standby database maintenance user in both the active and standby Adaptive Servers. Ensure that the server user's IDs match. (You create login names in the active Adaptive Server because using mount to initialize the standby database with data from the active database overrides any previous contents of the standby database with the contents of the active database.)
Initializing standby database.	Use dump and load to transfer data from the active database to the standby database. You can use database dumps and/or transaction dumps.		Use bcp to copy each replicated table from the active database to the standby database.	Use quiesce database ... to *manifest_file* and mount database to transfer data from the active database to the standby database.

Issue	Use dump and load with "dump marker"	Use dump and load without "dump marker"	Use bcp	Use mount
Active database connection state.	The connection to the active database does not change.	Replication Server suspends the connection to the active database.		Replication Server suspends the connection to the active database.
Resuming connections.	Resume connection to the standby database.	Resume connections to the active and standby databases; resume transaction processing in the active database.		Resume connections to the active and standby database; resume transaction processing in the active database.

If you do not suspend transaction processing

If you do not suspend transaction processing for the active database while initializing the standby database, choose the "dump marker" option when you add the standby database. Then initialize the standby database by using the dump and load commands.

Replication Server starts replicating into the standby database from the first dump marker after the enable replication marker in the transaction log of the active database.

In Figure 3-2, transaction T1, executed after you added the standby database, appears after the enable replication marker in the log. T1 is included in dumps, so it is present in the standby database after you have loaded the dumps. Replication Server does not need to replicate it into the standby database.

Figure 3-2: Using dump and load with dump marker
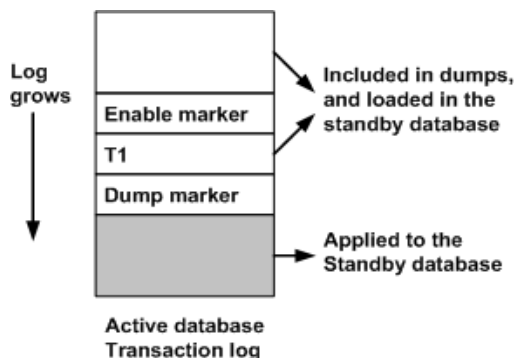
Transactions can be executed in the active database between the time the enable replication marker is written and the time the data in the active database is dumped.

You can load the last full database dump and any subsequent transaction dumps into the standby database until both markers have been received and the standby database is ready for operation. Then, optionally, you can use a final transaction dump of the active database to bring the standby database up to date. Any transactions not included in dumps will be replicated.

Replication Server does not replicate transactions from the active to the standby database until it has received both the enable replication marker and the first subsequent dump marker. After receiving both markers, Replication Server starts executing transactions in the standby database.

See Table 3-4 for more information about this method.

If you suspend
transaction processing

If you suspend transaction processing for the active database while initializing the standby database, do not choose the "dump marker" option when you add the standby database. You can initialize the standby database by using the dump and load commands, by using bcp, or by using mount.

Replication Server starts replicating into the standby database from the enable replication marker in the transaction log of the active database. No transactions occur after the enable replication marker, because client applications are suspended.

Figure 3-3: Using dump and load without dump marker, or using bcp
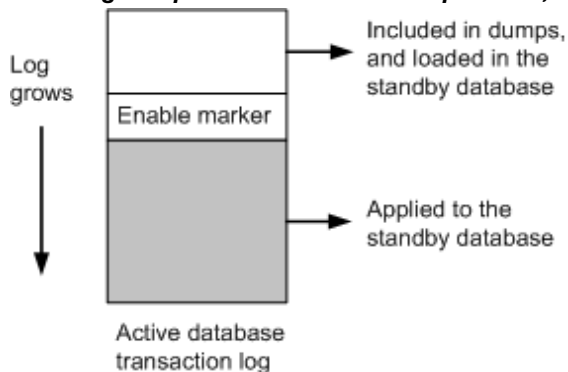
As shown in Figure 3-3, no transactions are executed in the active database between the time the enable replication marker is written and the time the data in the active database is dumped using the dump command, or copied using bcp or mount.

You can load the last full database dump or the last set of replicated tables copied with bcp into the standby database until the standby database receives the enable replication marker.

After receiving this marker, Replication Server starts executing transactions in the standby database.

Adding the standby database maintenance user

If you plan to initialize the standby database using the dump and load commands, with or without the "dump marker" option, you must create the maintenance user login name for the standby database in both the standby and the active data servers. Do this before you add the standby database.

Both Sybase Central and rs_init automatically add the active database maintenance user in the active data server when you add the active database.

Making the server user's IDs match

Within each data server, the server user's ID (suid) for each login name must be the same in the syslogins table in the master database and the sysusers table in each user database. This must be true for the active and standby databases in a warm standby application. The server user's ID and role settings must also be the same in the syslogins and sysloginroles tables in the master database.

Use one of these three methods to make the server user's IDs match:

- Add all login names, including maintenance user names, to both Adaptive Servers in the same order. Adaptive Server assigns server user's IDs sequentially, so the server user's IDs for all login names will match.

- After loading the dump into the standby, reconcile the sysusers table in the standby database with the syslogins table in the master database of the standby Adaptive Server.

- Maintain a master Adaptive Server with all of your login names and copy the syslogins table from the master database for the master Adaptive Server to all newly created Adaptive Servers.

Adding the maintenance user

To add the maintenance user login name for the standby database to both the standby and the active data servers:

1   In the standby data server, execute the sp_addlogin system procedure to create the maintenance user login name.

    Refer to the *Adaptive Server Enterprise System Administration Guide* for more information about using sp_addlogin.

2   In the active data server, execute sp_addlogin to create the same maintenance user login name that you created in the standby data server.

    When you set up the standby database using the dump and load commands, the sysusers table is loaded into the standby database along with the other data from the active database.

Adding the standby database to the replication system

To add the standby database to the replication system:

1   Suspend transaction processing in the active database, if appropriate for your client applications and your method of initializing the standby database.

    You must use dump and load with the "dump marker" method if you do not suspend transaction processing.

2   Use Sybase Central or rs_init to add the standby database to the replication system. Perform the steps described for adding a database to the replication system.

3   To monitor the status of the logical connection at any time, enter:

    ```
    admin logical_status, logical_ds, logical_db
    ```

    The Operation in Progress and State of Operation in Progress output columns indicate the standby creation status.

4   If you are initializing the standby database using dump and load, use the dump command to dump the contents of the active database, and load the standby database. For example:

    ```
    dump database active_database to dump_device
    load database standby_database from dump_device
    ```

5   If you have already loaded a previous database dump and subsequent transaction dumps, you can just dump the transaction log and load it into the standby database. For example:

    ```
    dump transaction active_database to dump_device
    load transaction standby_database from dump_device
    ```

6   After completing load operations, bring the standby database online:

    ```
    online database standby_database
    ```

    Refer to the *Adaptive Server Enterprise Reference Manual* for help with using the dump and load commands and the online database command.

7   Initialize the standby database. Use bcp or quiesce ... to *manifest_file* and mount.

    •   To initialize the standby database using bcp, copy each of the replicated tables in the active database to the standby database.

        You must copy the rs_lastcommit table, which was created when you added the active database to the replication system.

Refer to the Adaptive Server utility programs manual for help with using the bcp program.

- To initialize the standby database using quiesce ... to *manifest_file* and mount, quiesce the database and create the manifest file. Make a copy of both the database and log devices. Mount the devices on the standby database.

8 If you initialized the standby database by using dump and load without the "dump marker" method, or by using bcp, or by using quiesce database ... to *manifest_file* and mount, Replication Server suspended the connection to the active database. Resume the connection by executing the following command in the Replication Server:

```
resume connection to active_ds.active_db
```

9 Regardless of your method for initializing the standby database, you must resume the connection to the standby database by executing the following command:

```
resume connection to standby_ds.standby_db
```

10 Resume transaction processing in the active database, if it was suspended.

Using a blocking command for standby creation

In Replication Server, the wait for create standby command is a blocking command. It tells Replication Server not to accept commands until the standby database is ready for operation. You can use this command in a script that creates a standby database. The syntax is:

```
wait for create standby for logical_ds.logical_db
```

Enabling replication for objects in the standby database

To be ready to switch to the standby database, replication must be enabled for the tables and stored procedures in the standby database that you want to replicate into the new standby database after the switch.

- If you initialized the standby database using the dump and load or mount commands, the tables and stored procedures in the standby database will have the same replication settings as the active database.

- If you initialized the standby database using bcp, enable replication for these objects by using sp_setreptable or sp_reptostandby, and sp_setrepproc. To do this, adapt the procedure under "Task three: Enabling replication for objects in the active database" on page 76 to enable replication for objects in the standby database.

Enabling replication for objects added later	Later on, you may add new tables and user stored procedures that you want to replicate to the new standby database.

- If you marked the standby database for replication with sp_reptostandby, any new tables are automatically marked for replication.

- If you marked individual database tables for replication to the new standby database with sp_setreplicate, you must mark each new table that you want to replicate with sp_setreplicate.

- You must mark each new user stored procedure that you want to replicate with sp_setrepproc.

Granting permissions to the maintenance user

After adding the standby database, you must grant the necessary permissions to the maintenance user.

To grant permissions:

1   Log in to the Adaptive Server as the System Administrator or as the Database Owner, and enter:

```
use standby_database
```

2   Grant replication_roleto the maintenance user. replication_role ensures that the maintenance user can execute truncate table at the standby database.

```
sp_role "grant", replication_role, maintenance_user
```

3   Execute this command for each table:

```
grant all on table_name to maintenance_user
```

Replicating the master database in a warm standby environment

You can replicate Adaptive Server logins from one master database to another. The master database replication is limited to DDL, and the system commands used to manage logins and roles. Master database replication does not replicate data from system tables, nor replicate data or procedures from any other user tables in the master database.

Both the source Adaptive Server and the target Adaptive Server must be the same hardware architecture type (32-bit versions and 64-bit versions are compatible), and the same operating system (different versions are also compatible).

Do not initialize the active and standby databases with a load from another master database. To synchronize the syslogins, suids and roles at each master, use bcp to refresh the appropriate tables or manually synchronize the IDs and roles prior to setting up your replication.

For a list of supported DDL and system procedures that apply to the master database, see "Restrictions and requirements when using sp_reptostandby" on page 63 and "Supported DDL commands and system procedures" on page 64.

Replication Server versions 12.0 and later support master database replication in a warm standby environment, and in an MSA environment in Replication Server 12.6 and later. The primary or active Adaptive Server must be version 15.0 ESD #2 or later.

See "Replicating the master database in an MSA environment" in *Replication Server Administration Guide Volume 1* for information about master database replication in an MSA environment.

❖ **Setting up master database replication in a warm standby environment**

1   Set up the active master database and the standby master database in the Replication Server as warm standby pair. Do not "initialize the standby with dump and load" nor "use the dump marker to start replicating to standby". To synchronize the syslogins and suids at each master, use bcp or manually synchronize the IDs.

    See "Setting up warm standby databases" for more information on setting up a warm standby environment.

2   Mark the master database on both the active and the standby database to send system procedures:

    ```
    sp_reptostandby master, 'all'
    ```

3   Stop the RepAgent on the active master database:

    ```
    sp_stop_rep_agent master
    ```

4   Configure the Replication Agents on both the active and the standby databases to send warm standby transactions:

    ```
    sp_config_rep_agent master, 'send warm standby
    xacts', 'true'
    ```

5   Restart the RepAgent on the active master database:

```
sp_start_rep_agent master
```

6   Resume the DSI connections to both the active and the standby master
     databases on the Replication Server:

```
resume connection to active_ds.master
go
resume connection to standby_ds.master
go
```

7   Verify the status of warm standby:

```
admin logical_status
```

Switching the active and standby databases

This section contains information about switching to the standby database
when the active database fails or when you want to perform maintenance on the
active database.

Determining if a switch is necessary

Determining when it is necessary to switch from the active to the standby
database depends on the requirements of your applications.

In general, you should not switch when the active data server experiences a
transient failure. A transient failure is a failure from which the Adaptive Server
recovers upon restarting with no need for additional recovery steps. You
probably should switch if the active database will be unavailable for a long
period of time.

Determining when to switch depends on issues such as how much recovery the
active database requires, to what degree the active and standby databases are in
sync, and how much downtime your users or applications can tolerate.

You may also want to switch the roles of the active and standby databases to
perform planned maintenance on the active database or its data server.

Before switching active and standby databases

Figure 3-4 illustrates a warm standby application for a database that does not participate in the replication system other than through the activities of the warm standby application itself. Figure 3-4 represents the warm standby application in normal operation, before you switch the active and standby databases.

Figure 3-4: Warm standby application example—before switching
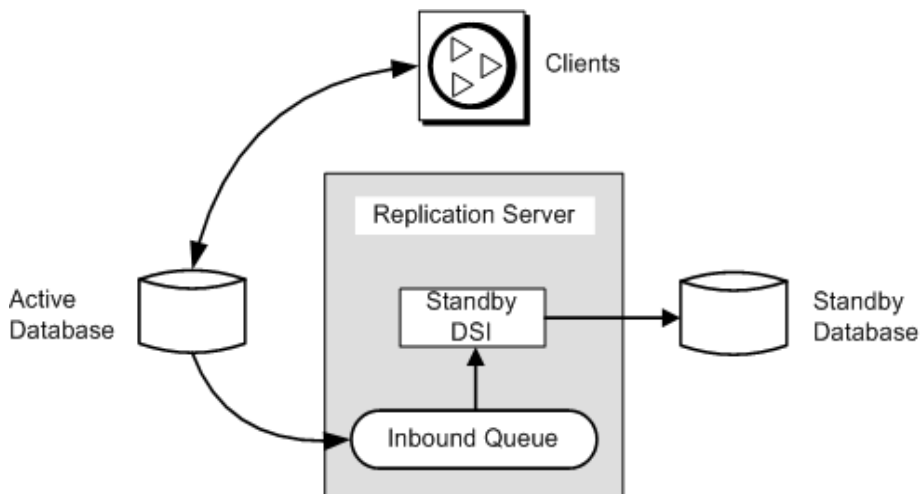
Figure 3-4 adds internal detail to Figure 3-1, to show that:

- Replication Server writes transactions received from the active database into an inbound message queue.

  See "Distributed concurrency control" on page 48 in the *Replication Server Administration Guide Volume 1* for more information about inbound and outbound queues.

- This inbound queue is read by the DSI thread for the standby database, which executes the transactions in the standby database.

  Messages received from the active database cannot be truncated from the inbound queue until the standby DSI thread has read them and applied them to the standby database.

In this example, transactions are simply replicated from the active database into the standby database. The logical database itself does not:

- Contain primary data that is replicated to replicate databases or remote Replication Servers, or

•   Receive replicated transactions from another Replication Server

See "Warm standby applications using replication" on page 104 for information about warm standby applications for a primary or replicate database.

Internal switching steps

When you switch active and standby databases, here is what Replication Server does:

1   Issues log suspend against the active and standby RepAgent connections.

2   Reads all messages left in the inbound queue and applies them to the standby database and, for subscription data or replicated stored procedures, to outbound queues.

All committed transactions in the inbound queue must be processed before the switch can complete.

3   Suspends the standby DSI.

4   Enables the secondary truncation point in the new active database.

5   Places a marker in the transaction log of the new active database. Replication Server uses this marker to determine which transactions to apply to the new standby database and to any replicate databases.

6   Updates data in the RSSD pertaining to the warm standby databases.

7   Resumes the connection for the new active database, and resumes log transfer for the new active database so that new messages can be received.

After switching active and standby databases

After you have switched the roles of the active and standby databases, the replication system will have changed, as shown in Figure 3-5:

Figure 3-5: Warm standby application example—after switching
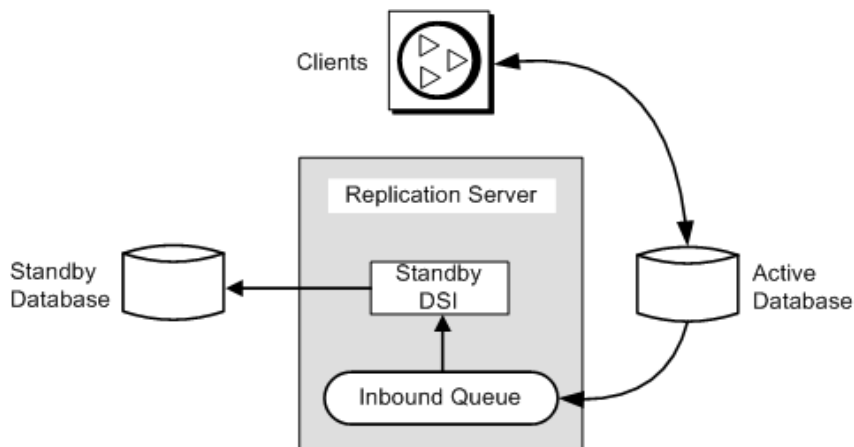
- The previous standby database is the new active database. Client applications will have switched to the new active database.

- The previous active database, in this example, becomes the new standby database. Messages for the previous active database are queued for application to the new active database.

Note RepAgent for the previous active database has shut down. RepAgent for the new active database has started.

Making the switch

❖ **Switching from the active to the standby database**

1   Disconnect client applications from the active database if they are still using it

2   In Replication Server, switch the active and standby databases

3   Restart client applications with the new active database

4   Start RepAgent for the new active database

5   Determine whether to drop the old active database or use it as the new standby database

The following sections contain the procedures for these tasks.

Disconnect client applications from the active database

Before you switch to the standby database, you must stop clients from executing transactions in the active database. If the database failed, of course, clients cannot execute transactions. However, you may need to take steps to prevent them from updating that database after it is back online.

See "Setting up clients to work with the active data server" on page 97 for more information about client application issues.

Procedure for switching the active and standby databases

Before switching, you must implement a method for switching clients, as described in "Setting up clients to work with the active data server" on page 97.

Follow these steps to switch the active and standby databases for a logical connection:

1   At the Adaptive Server of the active database, ensure that the RepAgent is shut down. Otherwise, use sp_stop_rep_agent to shut down the RepAgent.

2   At the Replication Server, enter:

```
switch active for logical_ds.logical_db
to data_server.database
```

data_server.database is the new active database.

See "Internal switching steps" on page 89 for information on what Replication Server does when you switch.

3   To monitor the progress of a switch, you can use the admin logical_status command:

```
admin logical_status, logical_ds, logical_db
```

The Operation in Progress and State of Operation in Progress output columns indicate the switch status.

4   When the active database switch is complete, you must restart RepAgent for the new active database:

```
sp_start_rep_agent dbname
```

Note If Replication Server stops in the middle of switching, the switch resumes after you restart Replication Server.

Using a blocking command for switch active

In Replication Server, the wait for switch command is a blocking command. It tells Replication Server to wait until the standby database is ready for operation. You can use this command in a script that switches the active database. The syntax is:

```
wait for switch for logical_ds.logical_db
```

Monitoring the switch

You can use admin logical_status to check for replication system problems that prevent the switch from proceeding. Such problems may include a full transaction log for the standby database or a suspended standby DSI. If you cannot resolve the problems, you can abort the switch using the abort switch command.

The Operation in Progress and State of Operation in Progress output columns indicate the switch status.

For example, suppose the admin logical_status command persistently returns one of the following messages in its State of Operation in Progress output column:

```
Standby has some transactions that have not been applied
```

or

```
Inbound Queue has not been completely read by
Distributor
```

These messages may indicate a problem that you cannot resolve, in which case you may choose to abort the switch. You can use admin who commands to obtain more information about the state of the switching operation.

See "Commands for monitoring warm standby applications" on page 96 for more information.

Aborting a switch

Unless Replication Server has proceeded too far in switching the active and standby databases, you can abort the process by using the abort switch command:

```
abort switch for logical_ds.logical_db
```

If the abort switch command cancels the switch active command successfully, you may have to restart the RepAgent for the active database.

You cannot cancel the switch active command after it reaches a certain point. If this is the case, you must wait for the switch active command to complete, then use it again to return to the original active database.

Restart client applications

When the admin logical_status command indicates that there is no operation in progress, or when the wait for switch command returns an isql prompt, you can restart client applications in the new active database.

Client applications must wait until Replication Server switch to the new active database is complete before they begin executing transactions in the new active database. You should provide an orderly method for moving clients from the old active database to the new active database. See "Setting up clients to work with the active data server" on page 97 for more information.

Resolving paper-trail transactions

If the old active database failed, determine if any transactions were not transmitted to the new active database. Such transactions are called **paper-trail transactions** if there is an external record of their execution.

When you switch from an active to a standby database, all committed transactions in the inbound queue are applied to the new active database before the switch is complete. However, it is possible that some transactions that committed at the active database before the failure were not received by Replication Server and, therefore, were not applied to the standby database.

When you switch the active and standby databases, you can re-execute the paper-trail transactions in the new active database. If there are dependencies, you may need to re-execute the paper-trail transactions before you allow new transactions to execute. Be sure to execute the paper-trail transactions using the original client's login name, not the maintenance user login name.

If you bring the old active database online as the new standby database, you must first reverse the paper-trail transactions so they will not be duplicated in the standby database.

Manage the old active database

After you have switched to the new active database, you must decide what to do with the old active database. You can:

• Bring the database online as the new standby database and resume the connection so that Replication Server can apply new transactions, or

- Drop the database connection using the drop connection command, and add it again later as the new standby database. If you drop the database, any queued messages for the database are deleted. Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about drop connection command.

Bringing the old active database online as the new standby

If the old active database is undamaged, you can bring it back online as the new standby database by entering:

```
resume connection to data_server.database
```

where *data_server.database* is the physical database name of the old active database.

You may need to resolve paper-trail transactions in the database in order to avoid duplicate transactions. Depending on your applications, you may need to do this before you bring the old active database back online as the new standby database.

Because paper-trail transactions must be re-executed in the new active database, you must prepare the new standby database so that it can receive the transactions again when they are delivered through the replication system.

To resolve the conflicts, you can:

- Undo or reverse the duplicate transactions in the new standby database, or

- Ignore the duplicate transactions and deal with them later.

Monitoring a warm standby application

This section describes methods you can use to monitor a warm standby application.

Replication Server log file

You can read the Replication Server log file for messages pertaining to warm standby operations. This section discusses log messages you will see when you add the standby database.

Standby connection created

These are examples of the messages that Replication Server writes while creating the physical connection for a standby database:

```
I. 95/11/01 17:47:50. Create starting : SYDNEY_DS.pubs2
I. 95/11/01 17:47:58. Placing marker in TOKYO_DS.pubs2 log
I. 95/11/01 17:47:59. Create completed : SYDNEY_DS.pubs2
```

In these examples, SYDNEY_DS is the standby data server and TOKYO_DS is the active data server.

When you create the physical connection for the standby database, Replication Server writes an "enable replication" marker in the active database transaction log. The standby DSI ignores all transactions until it has received this marker. If, however, you chose the "dump marker" option, the standby DSI continues to ignore messages until it encounters the next dump marker in the log.

When the appropriate marker arrives at the standby database from the active database RepAgent, the standby DSI writes a message in the Replication Server log file and then begins executing subsequent transactions in the standby database.

In the example messages above, Replication Server has created the connection for the standby database, SYDNEY_DS.pubs2, and suspended its DSI thread. At this point, the Database Administrator dumps the contents of the active database, TOKYO_DS.pubs2, and loads it into the standby database.

Standby connection resumed after initialization

After the Database Administrator has loaded the dump into the standby database and resumed the connection to the standby database, the standby DSI begins processing messages from the active database. Replication Server writes in its log messages similar to this:

```
I. 95/11/01 18:50:34. The DSI thread for database 'SYDNEY_DS.pubs2' is started.
I. 95/11/01 18:50:41. Setting LTM truncation to 'ignore' for SYDNEY_DS.pubs2 log
I. 95/11/01 18:50:43. DSI for SYDNEY_DS.pubs2 received and processed Enable
       Replication Marker. Waiting for Dump Marker
I. 95/11/01 18:50:43. DSI for SYDNEY_DS.pubs2 received and processed Dump
      Marker. DSI is now applying commands to the Standby
```

When you see the final message in the log file, the warm standby database creation process has completed.

Commands for monitoring warm standby applications

Use the admin commands to monitor the status of a warm standby application. Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about these commands.

admin logical_status

The admin logical_status command tells you:

- How the addition of a standby database or the switching between active and standby databases is progressing.

- Whether the active or standby database connection is suspended.

- Whether the standby DSI is ignoring messages. The standby DSI ignores messages while it waits for a marker to arrive in the transaction stream from the active database.

admin who, dsi

The admin who, dsi command provides another method to check the status of the standby DSI. The IgnoringStatus output column contains either:

- "Applying" – if the DSI is applying messages to the standby database, or

- "Ignoring" – if the DSI is waiting for a marker.

admin who, sqm

The admin who, sqm command provides information about the state of stable queues. In a warm standby application, the inbound queue is read by the Distributor thread, if you have not disabled it, and by the standby DSI thread. Replication Server cannot delete messages from the inbound queue until both threads have read and delivered them.

If Replication Server is not deleting messages from the inbound queue, you can use the admin who, sqm command to investigate the problem. The command tells you how many threads are reading the queue and the minimum deletion point in the queue.

admin sqm_readers

The admin sqm_readers command monitors the read and delete points of the individual threads that are reading the inbound queue. If the inbound queue is not being deleted, admin sqm_readers will help you find the thread that is not reading the queue.

The admin sqm_readers command takes two parameters: the queue number and the queue type for the logical connection.

You can find the queue number and queue type in the Info column of the admin who, sqm command output: the queue number is the 3-digit number to the left of the colon, while the queue type is the digit to the right of the colon.

Queue type 1 is an inbound queue. Queue type 0 is an outbound queue. The inbound queue for a logical connection can be read by more than one thread. For example, to find out about the threads reading inbound queue number 102, execute admin sqm_readers as follows:

```
admin sqm_readers, 102, 1
```

Setting up clients to work with the active data server

When you switch the active and standby databases in Replication Server using the switch active command, Replication Server does not switch client applications to the new active data server and database automatically. You must devise a method to switch client applications. This section describes three sample methods for setting up client applications to connect to the currently active data server. You can create:

- Two interfaces files
- An interfaces file entry with a symbolic data server name for client applications
- A mechanism that automatically maps the client application data server connections to the currently active data server

You must implement your method before you set up the warm standby databases.

Regardless of your method for switching applications, do not modify the interfaces file entries used by Replication Server.

Two interfaces files

With this method, you set up two interfaces files, one for the client applications and one for Replication Server. When you switch the clients, you can modify their interfaces file entry to use the host name and port number of the data server with the new active database.

Symbolic data server name for client applications

With this method, you create an interfaces file entry with a symbolic data server name for client applications.

The interfaces file might contain entries like these:

Table 3-5: Symbolic data server name in interfaces file

	Data server name	Host name	Port number
Client applications	CLIENT_DS	*machine_1*	2800
Active database	TOKYO_DS_X	*machine_1*	2800
Standby database	TOKYO_DS_Y	*machine_2*	2802

You could create an interfaces entry for a data server named CLIENT_DS. Client applications would always connect to CLIENT_DS. The CLIENT_DS entry would use the same host name and port number as the data server with the active database.

Replication Server connects to the same host name and port number as the client applications but uses a different data server name. In this example, Replication Server would switch between the TOKYO_DS_X and TOKYO_DS_Y data servers.

After switching the active database, you would change the CLIENT_DS interfaces entry to the host name and port number of the data server with the new active database—in this example, *machine_2* and port number 2802.

Map client data server to currently active data server

With this method, you create a mechanism, such as an intermediate Open Server application, that automatically maps the client application data server connections to the currently active data server.

Refer to Open Server documentation, such as the *Open Server Server-Library/C Reference Manual*, for more information about how to create such an Open Server application.

Altering warm standby database connections

This section describes options for reconfiguring or modifying the logical database connection and the physical database connections. Under ordinary circumstances, if you set up a warm standby application through the usual procedure, the default settings will work correctly.

Altering logical connections

Use the alter logical connection command to modify parameters that:

• Affect logical connections

• Enable or disable the Distributor thread

• Enable or disable the replication of truncate table to the standby database

Changing parameters affecting logical connections

To update parameters that affect logical connections, log in to the source Replication Server and, at the isql prompt, enter:

```
alter logical connection
 to logical_ds.logical_db
set logical_database_param to 'value'
```

where *logical_ds* is the data server name for the logical connection, *logical_db* is the database name for the logical connection, *logical_database_param* is a logical database parameter, and *value* is a character string setting for the parameter.

New settings take effect immediately.

Warning! You should reset the logical connection parameters materialization_save_interval and save_interval *only* when there is a serious lack of stable queue space. Resetting them (from strict to a given number of minutes) may lead to message loss at the standby database.

Table 3-6 displays the configuration parameters that affect logical database connections.

Table 3-6: Configuration parameters affecting logical connections

logical_database_param	value
materialization_save_interval	Materialization queue save interval. This parameter is only used for standby databases in a warm standby application.
	Default: "strict" for standby databases
replicate_minimal_columns	Specifies whether Replication Server should send all replication definition columns for all transactions or only those needed to perform update or delete operations at the standby database. Values are "on" and "off."
	Replication Server uses this value in standby situations only when a replication definition does not contain a "send standby" parameter. Otherwise, Replication Server uses the value of the "replicate minimal columns" or "replicate all columns" parameter in the replication definition.
	Default: on
save_interval	The number of minutes that the Replication Server saves messages after they have been successfully passed to the destination data server. See "Save interval for recovery" on page 226 for details.
	Default: 0 minutes
send_standby_repdef_cols	Specifies which columns Replication Server should send to the standby database for a logical connection. Overrides "send standby" options in the replication definition that tell Replication Server which table columns to send to the standby database. Values are:
	• on – send only the table columns that appear in the matching replication definition. Ignore the "send standby" option in the replication definition.
	• off – send all table columns to the standby. Ignore the "send standby" option in the replication definition.
	• check_repdef – send all table columns to the standby based on "send standby" option.
	Default: check_repdef

Disabling the Distributor thread

If you do not replicate data from the active database into databases other than the standby database, Replication Server does not need a Distributor thread for the logical connection. You can disable the Distributor thread to save Replication Server resources.

To disable the Distributor thread, you must first drop any subscriptions for the data in the logical database. Then execute alter logical connection at the Replication Server:

```
alter logical connection
 to logical_ds.logical_db
set distribution off
```

If you decide later to replicate data out of the active database, you can use this command to reenable the Distributor thread.

Warning! If you disable the Distributor thread and then drop the standby database from the replication system, no Replication Server threads will be left to read the inbound queue from the active database. The inbound queue will continue to fill until you either add another standby database, set distribution to "on" for the logical connection, or drop the active database from the replication system.

Replicating *truncate table* to standby databases

Replication Server copies execution of truncate table to warm standby databases. The active and standby databases must be Adaptive Server version 11.5 or later to support this feature.

To enable or disable replication of truncate table, log in to the source Replication Server and enter:

```
alter logical connection
 to logical_ds.logical_db
set send_truncate_table to {on | off}
```

If your warm standby application was created *before* you upgraded or installed Replication Server version 11.5 or later, Replication Server does not copy truncate table to the standby database unless you enable this feature with alter logical connection. To preserve compatibility with existing warm standby applications, the default setting is "off."

If your warm standby application was created *after* you upgraded or installed Replication Server version 11.5 or later, Replication Server automatically copies truncate table to the standby database unless you disable this feature with alter logical connection. The default setting is "on."

Altering physical connections

Use the alter connection command at the source Replication Server to modify parameters that affect physical connections for warm standby applications:

```
alter connection to data_server.database
set database_param to 'value'
```

where *data_server* is the destination data server, *database* is the database the data server manages, *database_param* is a parameter that affects the connection and *value* is a setting for *database_param.*

You must suspend the connection before altering it; then, after executing alter connection, you resume the connection for new parameter settings to take effect. See "Altering database connections" on page 170 in the *Replication Server Administration Guide Volume 1* for more information.

Configuring triggers in the standby database

By default, the standby DSI thread executes a set triggers off Adaptive Server command when it logs in to a standby database. This prevents Adaptive Server from firing triggers for the replicated transactions, thereby preventing duplicate updates in the standby database.

You can alter the default behavior by using the alter connection command to configure a connection to fire or not fire triggers. To do this, set the dsi_keep_triggers configuration parameter to "on" or "off." The default dsi_keep_triggers setting for all connections except standby databases is "on."

Configuring replication in the standby database

The dsi_replication configuration parameter specifies whether or not transactions applied by the DSI are marked in the transaction log as being replicated. It must be set to "on" for the active replicate database. By default, it is set to "off" for the standby database and set to "on" for all other databases.

When dsi_replication is set to "off," the DSI executes set replication off in the database, preventing Adaptive Server from adding replication information to log records for transactions that the DSI executes. Since these transactions are executed by the maintenance user and, therefore, are not replicated further (except if there is a standby database), setting this parameter to "off" where appropriate writes less information into the transaction log.

Use admin who, dsi to see how this parameter is set for a connection.

Changing configuration parameters in the standby database

When you create the standby database, the following configuration parameters, if they are set for the active database, are copied from the active database to the standby database:

Table 3-7: Configuration parameters copied to standby database

batch	batch_begin	dsi_cmd_batch_size
db_packet_size	dsi_charset_convert	dsi_keep_triggers
dsi_cmd_separator	dsi_fadeout_time	dsi_max_text_to_log
dsi_large_xact_size	dsi_max_cmds_to_log	dsi_replication
dsi_num_large_xact_threads	dsi_num_threads	dsi_sqt_max_cache_size
dsi_serialization_method	dsi_xact_in_group	dump_load
dsi_xact_group_size	dsi_isolation_level	use_batch_markers
parallel_dsi	command_retry	

You can change the setting of any of these configuration parameters. See Chapter 7, "Managing Database Connections" in the *Replication Server Administration Guide Volume 1* for more information.

Dropping logical database connections

If you are dismantling a warm standby application, you may need to remove a logical database from the replication system. To do this, drop the standby database, then execute the drop logical connection command. Before you execute the command, you must drop the standby database. See "Dropping database connections" on page 187 in the *Replication Server Administration Guide Volume 1* for information about dropping physical database connections.

The syntax for drop logical connection is:

    drop logical connection to *data_server.database*

data_server and *database* represent the logical data server and logical database.

For example, to drop the connection to the pubs2 logical database in the LDS logical data server, enter:

```
drop logical connection to LDS.pubs2
```

Dropping a logical database from the ID Server

When a warm standby application exists in the replication system, logical databases, along with physical databases, data servers, and Replication Servers, are listed in the rs_idnames system table in the RSSD for the ID Server. Occasionally, it may be necessary to remove the entry for a logical database from this system table.

For example, if a drop logical connection command fails, you may have to force the ID Server to delete from the rs_idnames system table the row that corresponds to the logical database. Logical database connections show an "L" in the ltype column.

The sysadmin dropldb command logs in to the ID Server and deletes the entry for the specified logical database. The syntax is:

sysadmin dropldb, *data_server*, *database*

data_server and *database* refer to the logical data server and the logical database names.

You must have sa permission to execute any sysadmin command.

Warm standby applications using replication

This section describes warm standby applications that involve replication, where the logical database serves as a primary or replicate database in the replication system.

Also see "Using replication definitions and subscriptions" on page 111.

Warm standby application for a primary database

Figure 3-6 illustrates a warm standby application for a primary database. In this example, one Replication Server manages three databases:

*   The active database for a logical primary database,

*   The standby database for a logical primary database, and

*   A replicate database that has subscriptions for the data in the logical primary database.

In this example, a single Replication Server manages both the primary and replicate databases. In other instances, different Replication Servers may manage the primary and replicate databases.

Figure 3-6: Warm standby application for a primary database
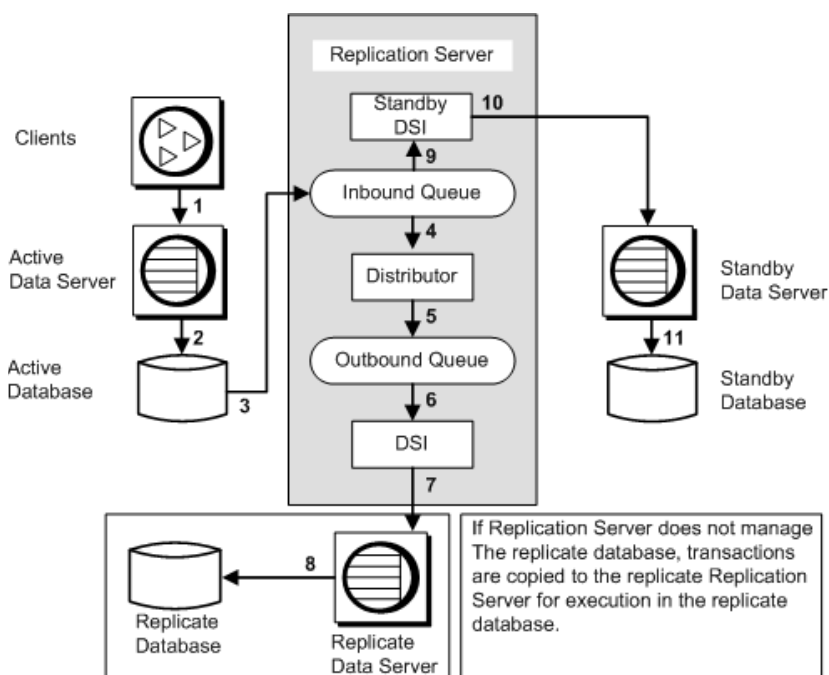
The numbers in Figure 3-6 indicate the flow of transactions from client applications through the replication system in a warm standby application for a primary database.

From client applications to inbound queue	In Figure 3-6, numbers 1 through 3 trace transactions from clients to an inbound queue in the Replication Server:

- Clients execute transactions in the active primary data server.

- The active primary data server updates the active primary database.

- The RepAgent thread for the active primary database reads transactions for replicated data in the database log. It forwards the transactions to the Replication Server, which writes them into an inbound queue.

  All transactions for replicated data, including those executed by the maintenance user, are sent to the Replication Server for application in the standby database.

From inbound queue to replicate database	In Figure 3-6, numbers 4 through 8 trace transactions from the inbound queue to the replicate database:

- The Distributor thread reads transactions from the inbound queue.

- The Distributor thread processes transactions against subscriptions and writes replicated transactions into an outbound queue.

  Transactions executed by the maintenance user, which are always replicated into the standby database (because you set the send_warm_standby_xacts parameter when you configure RepAgent with sp_config_rep_agent), are not replicated to replicate databases unless you also set the send_maint_xacts_to_replicate parameter for RepAgent.

- A DSI thread reads transactions from the outbound queue.

- The DSI thread executes the transactions in the replicate data server.

- The replicate data server updates the replicate database.

  If the transactions are to be replicated to a database managed by a different Replication Server, they are written into an RSI outbound queue managed by an RSI thread instead of a DSI thread. The RSI thread delivers the transactions to the other Replication Server.

From inbound queue to standby database	In Figure 3-6, numbers 9 through 11 trace transactions from the inbound queue to the standby database for the logical primary database:

- The standby DSI thread reads transactions from the inbound queue.

- The standby DSI thread executes transactions in the standby data server.

- The standby data server updates the standby database.

The inbound queue is read by the standby DSI and the Distributor. The two threads do their work concurrently. Messages cannot be truncated from the inbound queue until both threads have read them and delivered them to their destination. The messages remain in the queue until the DSI has applied them to the standby database and, if there are subscriptions or replicated stored procedure executions, the Distributor has written them to the outbound queue.

Depending on your replication system, the transactions may be replicated into the standby database before the replicate database. However, Replication Server guarantees that the standby primary database and replicate databases will be kept in sync with the active primary database.

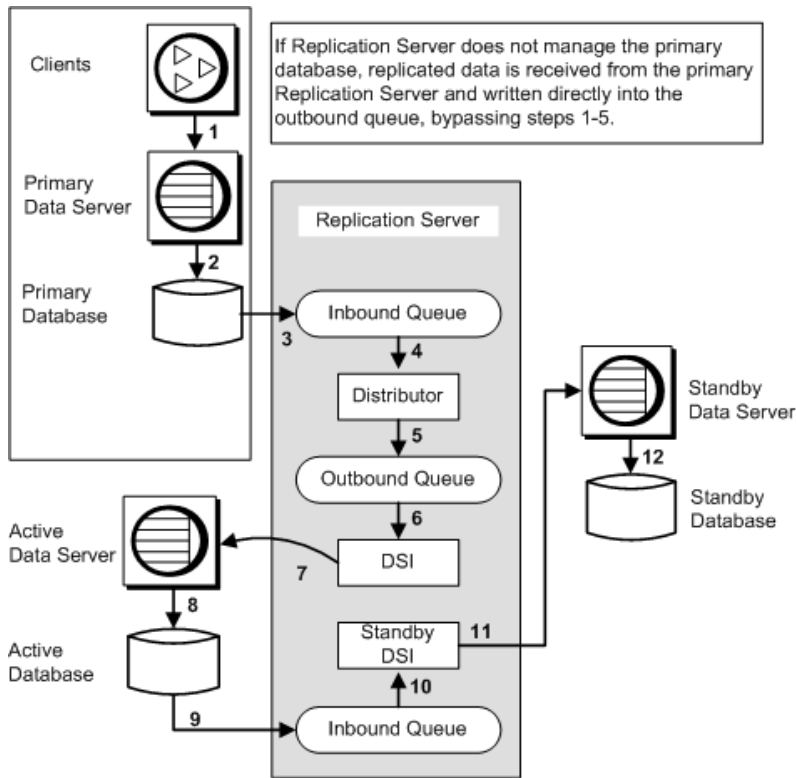## Warm standby application for a replicate database

Figure 3-7 illustrates a warm standby application for a replicate database. In this example, a single Replication Server manages three databases:

• A primary database,

• The active database for a logical replicate database, and

• The standby database for a logical replicate database.

The logical replicate database has subscriptions for the data in the primary database. Therefore, updates from the primary database are replicated to both the active and the standby databases.

In this example, a single Replication Server manages both the primary and replicate databases. In other instances, different Replication Servers may manage the primary and replicate databases.

Figure 3-7: Warm standby application for a replicate database

The numbers in Figure 3-7 indicate the flow of transactions from client applications through the replication system in a warm standby application for a replicate database.

From client applications to primary and active databases

In Figure 3-7, numbers 1 through 8 trace transactions from clients to the primary database, and, via normal replication, to the active replicate database:

•   Clients execute transactions in the primary data server.

•   The primary data server updates the primary database.

•   RepAgent for the primary database reads transactions for replicated data in the transaction log and forwards them to the Replication Server, which writes them into an inbound queue.

•   The Distributor thread reads transactions from the inbound queue.

- The Distributor processes transactions against subscriptions and writes replicated transactions into an outbound queue.

  If the Replication Server managing the warm standby application for the replicate database does not also manage the primary database, replicated data is received from the primary Replication Server and written directly to the outbound queue. Steps 1 through 5 are bypassed.

- A DSI thread reads transactions from the outbound queue.

- The DSI thread executes the transactions in the replicate data server, which is the active data server for the warm standby application.

- The active data server updates the active database.

  If the transactions originate in a primary database managed by a different Replication Server, the Distributor thread in the primary Replication Server writes them into an RSI outbound queue. Then they are replicated to a DSI outbound queue in the replicate Replication Server in order to be applied in the active database for the logical replicate database.

From active database to standby database

In Figure 3-7, numbers 9 through 12 trace transactions from the active database for the logical replicate database to its standby database:

- RepAgent for the active database reads the transactions in the active database log and forwards them to the Replication Server, which writes them into an inbound queue.

  All transactions for replicated data, including those executed by the maintenance user, are sent to the Replication Server for application in the standby database.

- The standby DSI thread reads transactions from the inbound queue.

- The standby DSI thread executes transactions in the standby data server.

- The standby data server updates the standby database.

Configuring logical connection save intervals

This section describes some options for reconfiguring the save intervals for a logical replicate database. A save interval for a connection specifies how long messages will be retained in a stable queue before they can be deleted. If you set up a warm standby application through the usual procedure, the default settings will work correctly.

You can use the configure logical connection command to configure the DSI queue save interval and the materialization queue save interval for the logical connection.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for the syntax of configure logical connection command.

Warning! The DSI queue save interval and the materialization queue save interval settings for a logical connection should be reset *only* under serious conditions stemming from a lack of stable queue space. Resetting these save intervals (from strict to a given number of minutes) may lead to message loss at the standby database. Replication Server cannot detect this type of loss; you have to verify the integrity of the standby database yourself.

The DSI queue save interval

By default, the DSI queue save interval for the logical connection is set to 'strict' when you create a standby database. This causes Replication Server to retain DSI queue messages until they are delivered to the standby database. If you must change the DSI queue save interval for the logical connection, use the configure logical connection command.

For example, to force a replicate Replication Server to save messages destined for its logical replicate data server LDS for one hour (sixty minutes), enter the following command:

```
configure logical connection to LDS.logical_pubs2
set save_interval to '60'
```

To reset this save interval back to 'strict', enter:

```
configure logical connection to LDS.logical_pubs2
set save_interval to 'strict'
```

The materialization queue save interval

The materialization queue save interval for the logical connection is set to 'strict' by default when you create a subscription. This causes Replication Server to retain materialization queue messages until they are delivered to the standby database. If you must change the materialization queue save interval for the logical connection, use the configure logical connection command.

For example, to force a replicate Replication Server to save messages in the materialization queue for its logical replicate data server LDS for one hour (sixty minutes), enter the following command:

```
configure logical connection to LDS.logical_pubs2
set materialization_save_interval to '60'
```

To reset this save interval back to 'strict', enter:

```
configure logical connection to LDS.logical_pubs2
set materialization_save_interval to 'strict'
```

Using replication definitions and subscriptions

This section contains information about using warm standby databases with replication definitions and subscriptions. See "Warm standby applications using replication" on page 104 for more information about warm standby applications for a primary or replicate database.

Creating replication definitions for warm standby databases

Replication Server does not require replication definitions to maintain a standby database, although using replication definitions can improve performance when replicating into a standby database. You can create a replication definition for each table in the logical database. You can also use function replication definitions when replicating into a standby database.

Replication definitions can change how Replication Server replicates data into a standby database, allowing you to optimize your warm standby application or enable a non-default behavior that your application requires.

You can use replication definitions in a warm standby application in the following scenarios:

• To improve the performance of the replication system, as described under "Using replication definitions to optimize performance" on page 114.

• In normal replication into or out of the logical database, as described in "Warm standby applications using replication" on page 104.

alter table support for warm standby

Adaptive Server Enterprise version 12.0 and later allows users to alter existing tables—add non-nullable columns, drop columns, and modify column datatypes.

This section describes how Replication Server supports table changes resulting from the alter table command when the table has no subscriptions.

Note To support table changes that result from alter table when subscriptions exist for that table, you need to alter the table's replication definition. See "Modifying replication definitions" on page 293 in the *Replication Server Administration Guide Volume 1* for instructions.

In previous releases, when a replication definition was defined for a table, Replication Server always used the column datatype defined in the warm standby replication definition. Beginning with Replication Server version 12.0, and depending on the situation, Replication Server may or may not use a table's replication definition.

No replication definition

When you use alter table against a table without replication definitions, Replication Server sends warm standby databases the same information it receives from the primary server. All options of alter table are supported. When you execute alter table at the primary, the command is replicated to the warm standby, and replication to the standby continues—no action is required in the Replication Server.

See the *Adaptive Server Enterprise Reference Manual, Volume 2: Commands* for alter table syntax and information.

***alter table* add column with default**

When you issue the alter table command in the active database to add a column with a default value, Adaptive Server creates a constraint with an auto-generated name. When the command is replicated to the standby database, the standby database also creates the same constraint with another, different auto-generated name. When you drop the constraint in the active database, the standby database does not recognize the constraint name and generates a data server interface (DSI) error.

To avoid this, drop the constraint in the active database first. The data server interface (DSI) shuts down automatically. Then drop the constraint created in the standby database and issue the resume dsi skip transaction command.

An alternative workaround is to execute:

```
alter table table name
replace column name
```

```
default null
```

This automatically drops the constraints created on both active and standby sites.

Warm standby with no *send standby* clause

When there is no send standby clause associated with any replication definition, Replication Server sends whatever data it receives from the primary table without referring to the replication definitions.

Replication Server uses the original column names and datatypes to send data received from the RepAgent. The replication definition is used only to find the primary key. The primary keys are the union of primary keys in all replication definitions for the table.

If schema changes do not involve dropping all primary key columns in all replication definitions of the table, the scenario is the same as discussed in "No replication definition" on page 112. All options of alter table are supported, and no action is required in the Replication Server.

You can alter the replication definition at any point to drop all primary keys in the replication definitions, and add the new primary key columns to the replication definitions before you alter the primary table.

Drop the old primary keys only after all of the old data rows are out of the replication system. Otherwise, the Data Server Interface (DSI) shuts down. If this occurs, see for recovery instructions.

Warm standby with *send standby all columns* clause

When send standby all columns is associated with a replication definition, Replication Server sends whatever data it receives from the RepAgent using the original column names and datatypes. The replication definition is used only to find the primary key.

If schema changes do not involve dropping all primary key columns in the replication definition with the send standby all columns clause, the scenario is the same as "No replication definition" on page 112. All options of alter table are supported, and no action is required in the Replication Server.

You can alter the replication definition at any time to drop all primary keys in the replication definition with the send standby all columns clause, and add the new primary key columns to the replication definition before you alter the primary table.

Drop the old primary keys after all of the old data rows have left the replication system. Otherwise, the Data Server Interface (DSI) shuts down. If this occurs, see "Recovering from inbound queue problems" on page 298 in the *Replication Server Administration Guide Volume 1* for recovery instructions.

Warm standby with *send standby replication definition columns* clause

When there is a send standby replication definition columns clause in the replication definition, the standby will continue to use the replicate table name and column names as well as the datatype defined in the table's corresponding replication definition.

If you want the replication definition datatype to be used in the standby, always create a replication definition with a send standby replication definition columns clause.

Please note that:

- To add or alter columns in the primary database, follow the "Migration procedure" on page 296 in the *Replication Server Administration Guide Volume 1*.

- To drop columns in the primary database, you do not need to alter the replication definition of the table as long as you do not drop all primary key columns.

- To drop all primary key columns, alter the replication definition to add new primary key columns before you alter the primary table. You can drop the old primary keys when the old data rows have been removed from the replication system.

Using replication definitions to optimize performance

When you specify that you want to use a replication definition for replicating into a standby database:

- Replication Server optimizes updates and deletes by using the primary key defined in the replication definition to generate the where clause.

- You can specify whether Replication Server uses the replication definition's replicate minimal columns setting for replicating into the standby database. This setting indicates whether updates replace the values for all columns or only the columns with changed values.

- You can specify whether Replication Server replicates all of a table's columns or all of a stored procedure's parameters to the standby database or only those columns or parameters listed in the table or function replication definition.

Creating a replication definition for replicating into a standby database

To create a replication definition just for replicating into the standby database, use the send standby clause in the create replication definition command. The replication definition's primary key and replicate minimal columns setting will be used in replicating into the standby database.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for detailed information about using create replication definition command.

Specifying a primary key

Replication Server generates a where clause to specify target rows for updates and deletes.

- If a replication definition for a table is marked with the send standby clause, the generated where clause contains only the columns listed in the primary key clause of the create replication definition command.

- If there are replication definitions for a table but none are marked with the send standby clause, the generated where clause contains the columns listed in the union of the primary key clauses of all of the replication definitions.

- If there is no replication definition for a table, the generated where clause includes all columns in the table except text, unitext, image, rawobject, rawobject in row, timestamp, and sensitivity columns.

Updating minimal columns

If you create a replication definition for replicating into a standby database, you can take advantage of another replication system performance optimization, the minimal columns setting.

When you use the replicate minimal columns clause, replicated update and delete transactions include only the required columns. Values for unchanged columns can be omitted from update commands. Omitting the unnecessary columns reduces the size of messages delivered through the replication system and requires Adaptive Server to do less work.

If you are not using replication definitions for replicating into the standby, you can still attain this performance benefit.

Minimal column replication occurs automatically if you have no replication definitions for a table or if you have replication definitions for a table but do not use one for replicating into the standby database.

Specifying columns to replicate into the standby database

If you create a replication definition for replicating into a standby database, you can specify which set of columns to replicate:

- Specify send standby or send standby all columns to replicate all the columns in the table into the standby database.

- Specify send standby replication definition columns to replicate only the replication definition's columns into the standby database.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about using the send standby clause with the create replication definition command.

Specifying parameters to replicate into the standby database

If you create a function replication definition, you can specify which set of parameters to replicate:

- Specify send standby all parameters (or omit the all parameters clause) to replicate all the parameters for the stored procedure into the standby database.

- Specify send standby replication definition parameters to replicate only the replication definition's parameters into the standby database.

If a replicated stored procedure has no function replication definition, when the stored procedure is executed, Replication Server replicates all of its parameters from the active database into the standby database. You can create only one function replication definition per replicated stored procedure.

See Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about using the send standby clause with the create applied/request function replication definition command.

Using replication definitions for tables with more than 1024 columns

Adaptive Server limits the number of expressions in the where clause to 1024. For warm standby applications, you must use replication definitions to replicate tables with more than 1024 columns, and make sure that the primary key is not more than 1024 columns. Adaptive Server generates an error if the Replication Server where clause has more than 1024 columns.

See "Using replication definitions to optimize performance" on page 114 for more information about the primary key and replication into the standby database.

Using replication definitions to copy redundant updates

Without a replication definition, Replication Server does not replicate redundant updates to the warm standby. That is, if an update merely changes the current value to the same value, and thus the before and after images are identical, Replication Server does not replicate the update.

However, if you want to replicate redundant updates, create a replication definition for the column that includes the send standby replication definition parameters option.

If you create a replication definition for a table, Replication Server *always* sends redundant updates, even when the replication definition is created with the replicate minimal columns option.

Using subscriptions with warm standby application

Although subscriptions are not used in replicating from the active to the standby database, you can:

- Create subscriptions for the data in a logical primary database, or
- Create subscriptions in order to replicate data from other databases into a logical replicate database.

The create subscription and define subscription commands use the logical database and data server names instead of the physical names.

See "Warm standby applications using replication" on page 104 for more information about warm standby applications for a primary or replicate database. Also see Chapter 11, "Managing Subscriptions" in the *Replication Server Administration Guide Volume 1* for more information about subscriptions and subscription materialization.

Restrictions on using subscriptions

Replication Server supports all forms of subscription materialization and dematerialization in warm standby applications. These restrictions apply to the creation of subscriptions that replicate data from or into warm standby databases:

- When there is a logical connection for a database, you cannot create a subscription for the physical active or standby database. You must create the subscription for the logical database in order to replicate subscription data into or from both the active and standby databases.

- You cannot create subscriptions while adding the standby database to the replication system. You must wait until the standby database has been properly initialized.

- You cannot add the standby database to the replication system while any subscriptions are being created.

- You cannot create new subscriptions while the switch active command is executing.

Subscription materialization for logical primary database

This section describes subscription materialization issues for a logical primary database. It also describes what happens if you execute the switch active command for a logical primary database during subscription materialization.

During subscription materialization, data is selected from the active primary database into a materialization queue.

When you execute the switch active command, the primary Replication Server replicates RSSD information to notify replicate sites that the active database has been changed. When a replicate Replication Server with a materializing subscription receives this information, the materialization queue is dropped. A new queue is built by reselecting the subscription data from the new active primary database.

Note The RepAgent thread for the RSSD of the primary Replication Server must be running for replicate Replication Servers to detect that the active database has been changed.

Subscription materialization for logical replicate database

This section describes subscription materialization issues for a logical replicate database. It also describes what happens if you execute the switch active command for a logical replicate database during subscription materialization.

The following sections discuss each subscription materialization method.

Atomic materialization   When you use atomic materialization, Replication Server sets the save interval for the materialization queue to 'strict'. Transactions are not deleted from the materialization queue until the data has been applied to the active database and replicated into the standby database.

Replication Server executes a marker in the active replicate database when the materialization queue has been applied. The marker marks the start of transactions that execute after the materialization queue is applied.

When the marker is executed at the active replicate database, Replication Server writes an informational message like this in its log:

```
I. 95/10/03 18:00:15. REPLICATE RS: Created atomic subscription
publishers_sub for replication definition publishers_rep at active replicate
for <LDS.pubs2>
```

When the marker arrives at the standby replicate database, Replication Server writes an informational message like this in its log:

```
I. 95/10/03 18:00:15. REPLICATE RS: Created atomic subscription
publishers_sub for replication definition publishers_rep at standby
replicate for <LDS.pubs2>
```

Materialization is now complete and Replication Server drops the materialization queue. The subscription is considered VALID at both the active and the standby replicate database.

If you execute the switch active command while the materialization queue is being processed, Replication Server reapplies the materialization queue to the new active database. If you used the incrementally option to create the subscription, only the batches of materialization rows that were not already replicated into the new active database are reexecuted.

Nonatomic materialization

When you use nonatomic materialization, the save interval is set to 0, allowing Replication Server to delete rows from the materialization queue after they are applied to the active database.

If a subscription is materializing when you execute the switch active command, Replication Server finishes processing the materialization queue, but marks the subscription "suspect." Use the check subscription command to find the subscription status in the active and replicate databases. You must drop and re-create suspect subscriptions.

Bulk materialization

If you use bulk materialization to create a subscription that replicates data into a warm standby application, you must ensure that the subscription data is loaded into the active and standby replicate databases.

If you load the data with a method that logs the inserted rows, such as logged bcp, Replication Server replicates the rows into the standby database. If you load the data with a non-logged method, you must also load it into the standby database because the active database log contains no insert records to replicate into the standby database.

During bulk materialization, you execute the activate subscription with suspension command before you load the subscription data into the replicate database. By default, activate subscription with suspension suspends the DSI threads for both the active database and the standby database. Suspending DSI threads allows you to load the data into both databases.

If you load the data using logged bcp or some other method that logs the rows, execute activate subscription with suspension at active replicate only so that Replication Server only suspends the DSI for the active database. This allows the inserted rows to be replicated from the active database into the standby database.

Checking subscriptions

For a warm standby application for a logical replicate database, you can use the check subscription command to check subscription status. The Replication Server managing the warm standby application returns either one or two status messages, depending on whether or not the status is different for the active and the standby database.

For example, while you are creating a subscription, the materialization status may be VALID at the active database and ACTIVATING at the standby database.

Dropping subscriptions

For a logical replicate database, you can drop a subscription using the drop subscription command with the with purge option. A drop subscription marker follows the dematerialization data from the DSI queue to the active database, and then travels to the standby database. After the marker has been received at both databases, subscription data is deleted from both databases.

While executing
switch active

You can execute the switch active command at the replicate Replication Server while you drop a subscription using the drop subscription command with the with purge option. Replication Server suspends DSI threads and temporarily suspends dematerialization. After switch active completes, the DSI threads are resumed and dematerialization restarts.

Suspect *drop subscription*

Dropping a subscription using the with purge option for a logical replicate database may lead to a suspect drop subscription if:

• The subscription is materializing in the active database, and

• You switch the active and standby databases, then

- You drop the subscription while it is materializing in the new active database.

Dematerialization restarts and proceeds normally for the new active database, but the new standby (old active) database may retain some subscription data that is not purged. To resolve the discrepancy, you can reconcile the active and the standby database using the rs_subcmp program, or you can drop and re-create the standby database.

For example, you may see a warning message like this when you try to execute drop subscription:

```
W. 95/10/02 20:59:15. WARNING #28171 DSI(111 SYDNEY_DS.pubs2) -
/sub_dsi.c(1231)
 REPLICATE RS: Dropped subscription publishers_sub for replication
 definition publishers_rep at standby replicate for <SYDNEY_DS.pubs2> before
 it completed materialization at the Active Replicate. Standby replicate may
 have some subscription data rows left in the database
```

Missing columns when you create the standby database

When you create a standby database for an existing database that has replication definitions, missing columns may result under the following combination of circumstances:

- If the existing database has a replication definition that does not include all columns in the table, and

- An insert or update transaction that has not been committed is in the inbound queue, and

- You create a standby database for the existing database (now the active database), after which

- The transaction commits.

Although, by default, a standby database is supposed to receive all columns, at the time the transaction began, the standby database did not exist. Replication Server would have discarded values for columns not in the replication definition. If a column is not in the replication definition and the standby database allows a null value for the column, the row can be inserted into or updated in the standby database without the missing value. Otherwise, you must reconcile the databases yourself.

Loss detection and recovery

Creating a warm standby application introduces additional types of loss detection messages into a replication system. See Chapter 7, "Replication System Recovery" for general information on Replication Server recovery, and for recovery procedures.

If you rebuild queues in a Replication Server that participates in a warm standby application, the Replication Server may detect losses between any of the following databases:

Table 3-8: Loss detection in warm standby applications

Loss detected from	To
Logical replicate database	Logical primary database
Logical primary database	Physical replicate database
Physical primary database	Logical replicate database
Physical active database	Physical standby database
Logical primary database	Replication Server

If you need to use the ignore loss command in database recovery operations where a warm standby application is involved, use the same logical or physical data server and database designations that appear in the loss detection messages you received.

CHAPTER 4   **Performance Tuning**

To meet the needs and demands of your Replication Server system, you must manage resources effectively and optimize the performance of individual Replication Servers. You can affect the performance of a Replication Server by changing the values of configuration parameters, by using parallel DSI threads, or by choosing disk allocations. To manage these resources successfully, you should understand something about Replication Server internal processing.

Replication Server internal processing

During replication, data operations are carried out by several Replication Server **threads**. On UNIX platforms, they are POSIX threads. On Windows platforms, they are WIN32 threads. Replication Server also stores data in queues and relies on the Replication Server System Database (RSSD) for critical system information. This section describes how these internal operations support various processes within the primary and replicate Replication Servers.

Threads, modules, and daemons

Replication Server runs multiple threads concurrently. The total number of threads depends on the number of databases that a Replication Server manages and the number of Replication Servers to which it has direct routes. Each thread performs a specific function such as managing a user session, receiving messages from a RepAgent, receiving messages from another Replication Server, or applying transactions to databases.

Some threads call specific portions (or "**modules**") of Replication Server to determine the destination of messages and transactions, and to determine what operations to replicate and how to replicate them.

Daemon threads, which run in the background and perform specified operations at predefined times or in response to certain events, run during such Replication Server activities as subscription materialization.

For details on Replication Server threads, modules, and daemons involved in processes specific to the primary Replication Server, see "Processing in the primary Replication Server" on page 124.

When you troubleshoot the replication system, verify the status of Replication Server threads, modules, and daemons. See Chapter 1, "Verifying and Monitoring Replication Server" for details.

Processing in the primary Replication Server

This section describes how a transaction that originates in a primary data server is sent to the primary Replication Server and subsequently distributed to a replicate Replication Server as illustrated in Figure 4-1.
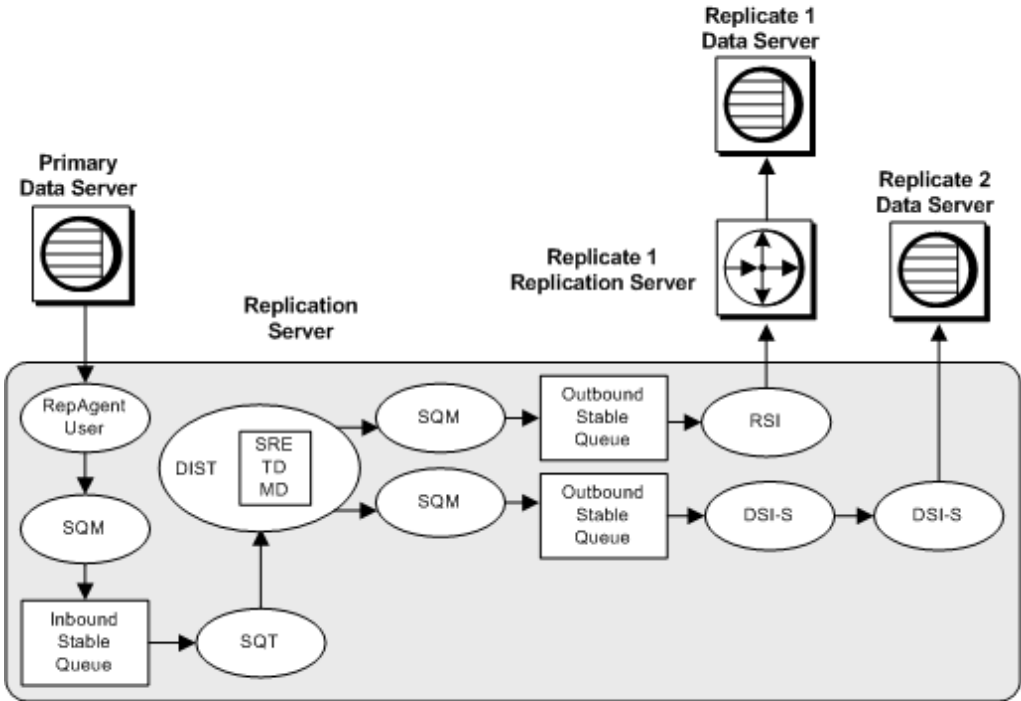
Replication agent user thread

The information in this section applies to all Replication Agents.

RepAgent logs in to Replication Server through an Open Client™ interface. It scans the transaction log, converts log records directly into LTL (Log Transfer Language) commands, and sends them to Replication Server as soon as they are logged—either in batches or one at a time. Replication Server then distributes the transaction information to subscribing replicate databases.

Replication Server has one RepAgent user thread for each primary database that it manages. Thus, Replication Server has one RepAgent user thread for each RepAgent. The RepAgent user thread verifies that RepAgent submissions are valid and writes them into the inbound stable queue for the database.

Stable Queue Manager thread

There is one Stable Queue Manager (SQM) thread for each stable queue accessed by the primary Replication Server, whether inbound or outbound. Each RepAgent user thread works with a dedicated SQM thread that reclaims stable queue space after a transaction is forwarded to a data server or to another Replication Server.

Stable Queue Transaction thread

Commands stored in transaction log records and in the inbound queue are ordered according to the sequence in which they were committed—although they are not necessarily grouped by transaction. It is the task of the Stable Queue Transaction (SQT) thread to reassemble transactions and place the transactions in commit order. Transactions must be in commit order for final application on the destination's data servers and for materialization processing.

The SQT thread reassembles transactions as it reads commands from its stable inbound queue and keeps a linked list of transactions. For the outbound queue, the DSI/S thread schedules transactions, and performs the SQT function of assembling and ordering transactions. When it reads a commit record, the SQT makes that transaction available to the distributor (DIST) thread or to the DSI thread, depending on what process required the SQT ordering of the transaction.

When it reads a rollback record, the SQT thread tells the SQM thread to delete affected records from all stable queues. Operated by the DSI/S thread, the SQT library notifies the DSI when a transaction exceeds the large transaction threshold. See "Using parallel DSI threads" on page 151 for more information on transaction thresholds.

Distributor thread and related modules

For each primary database managed by a Replication Server, there is a distributor (DIST) thread, which in turn uses SQT to read from the inbound queue and SQM threads to write transactions to the outbound queue. Thus, for example, if there are three primary databases, then there are three inbound queues, three DIST threads, and three SQT threads.

Note If the only destination for transactions is a standby database, disable the DIST thread, which also disables the SQT thread. The SQM thread is present and responsible for writing to the queue.

In determining the destination of each transaction row, the DIST thread makes calls to the following modules: Subscription Resolution Engine (SRE), Transaction Delivery, and Message Delivery. All DIST threads share these modules. These modules, and the role they play in the replication system, are described in the following sections.

Subscription Resolution Engine

The Subscription Resolution Engine (SRE) matches transaction rows with subscriptions. When it finds a match, it attaches a destination-database ID to each row. It marks only rows required for subscriptions, thereby minimizing network traffic. If no subscriptions match, the DIST thread discards the row data.

For each row, the SRE determines whether subscription migration occurs.

- A row migrates *into* a subscription when its column values change so that the row matches the subscription and must be added to the replicate table.

- A row migrates *out of* a subscription when its column values change so that it no longer matches the subscription and must be deleted from the replicate table.

When the SRE detects subscription migration, it determines which operation to replicate (insert, delete, or update) to maintain consistency between the replicate and primary tables.

Transaction Delivery module

The Transaction Delivery (TD) module is called by the DIST thread to package transaction rows for distribution to data servers and other Replication Servers.

Message Delivery module

The Message Delivery (MD) module is called by the DIST thread to optimize routing of transactions to data servers or other Replication Servers. The DIST thread passes the transaction row and the destination ID to the MD module. Using this information and routing information in the RSSD, the module determines where to send the transaction:

- To a data server via a DSI thread, or

- To a Replication Server via an RSI thread.

After determining how to send the transaction, the MD module places the transaction into the appropriate outbound queue.

Distributor status recording

A distributor (DIST) thread reads transactions from the inbound queue and writes replicated transactions into the outbound queue. A DIST thread is created when the Replication Server connects to the primary database, and can be suspended or resumed manually, or through a Replication Server configuration. Resuming and suspending a DIST thread modifies the DIST status of the thread.

You can save the DIST status of a distributor thread in the RSSD. This allows the DIST thread to retain its status even after a Replication Server is shut down.

For more information about DIST status recording, see the *Replication Server Reference Manual*.

Data Server Interface threads

Replication Server starts DSI threads to submit transactions to a replicate database to which it maintains a connection.

Each DSI thread is composed of a scheduler thread (DSI-S) and one or more executor threads (DSI-E). Each DSI executor thread opens an Open Client connection to a database.

To improve performance in sending transactions from a Replication Server to a replicate database it manages, you can configure a database connection so that transactions are applied using more than one DSI executor thread. See "Using parallel DSI threads" on page 151 for a description of this feature.

The DSI scheduler thread calls the SQT interface to:

- Collect small transactions into groups by commit order
- Dispatch transaction groups to the next available DSI executor thread

The DSI executor threads:

- Map functions using the function strings defined for the functions, according to the function-string class assigned to the database connection
- Execute the transactions in the replicate database
- Take action on any errors returned by the data server; depending on the assigned error actions, also record any failed transactions in the exceptions log

The DSI thread may apply a mixture of transactions from all primary databases supported by the Replication Server. The transactions are read from a single outbound stable queue for the replicate data server.

Replication Server Interface thread

RSI threads are asynchronous interfaces to send messages from one Replication Server to another. One RSI thread exists for each destination Replication Server to which the source database has a direct route.

The DIST thread in the primary Replication Server processes transactions, causing those destined for other Replication Servers to be written to RSI outbound queues. An RSI thread logs in to each replicate Replication Server and transfers messages from the stable queue to the replicate Replication Server.

When a direct route is created from one Replication Server to another, an RSI thread in the source Replication Server logs in to the replicate Replication Server. When an indirect route is created, Replication Server does not create a new stable queue and RSI thread. Instead, messages for indirect routes are handled by the RSI thread for the direct route. For details, see Chapter 6, "Managing Routes," in the *Replication Server Administration Guide Volume 1*.

Miscellaneous daemon threads

The Replication Server daemon threads shown in Table 4-1 perform miscellaneous tasks in the replication system.

Table 4-1: Additional Replication Server daemon threads

| Thread or daemon name	Description
Alarm daemon (dALARM)	The alarm daemon keeps track of alarms set by other threads, such as the fade-out time for connections and the interval for the subscription retry daemon.
Asynchronous I/O daemon (dAIO)	The asynchronous I/O daemon manages asynchronous I/O to Replication Server stable queues.
Connection manager daemon (dCM)	The connection manager daemon manages connections to data servers and other Replication Servers.
Recovery daemon (dREC)	The recovery daemon takes care of various operations in connection with warm standby applications, routing, and recovery procedures.
Subscription retry daemon (dSUB)	The subscription retry daemon wakes up after a configurable timeout period (sub_daemon_sleep_time configuration parameter in the rs_config system table) and attempts to resume processing for subscriptions that may have failed.
Version daemon (dVERSION)	The version daemon activates briefly when the Replication Server is started for the first time after an upgrade. It communicates the Replication Server new version number to the ID Server.

Thread or daemon name	Description
RS user thread	The RS user thread manages connections from replicate Replication Servers during the process of creating or dropping subscriptions.
	See "Subscription materialization methods" on page 353 in the *Replication Server Administration Guide Volume 1* for the data flow involved in creating and dropping subscriptions.
USER thread	A USER thread is created when a user logs in to a Replication Server to execute RCL commands.

Processing in the replicate Replication Server

This section describes the processes involved when a replicate Replication Server receives incoming messages from a primary Replication Server.

"Processing in the primary Replication Server" on page 124 describes processing for some of the threads—SQM, RSI, DSI—described in this section. Refer to Figure 4-1 on page 125.

Figure 4-2: Transaction processing in the replicate Replication Server
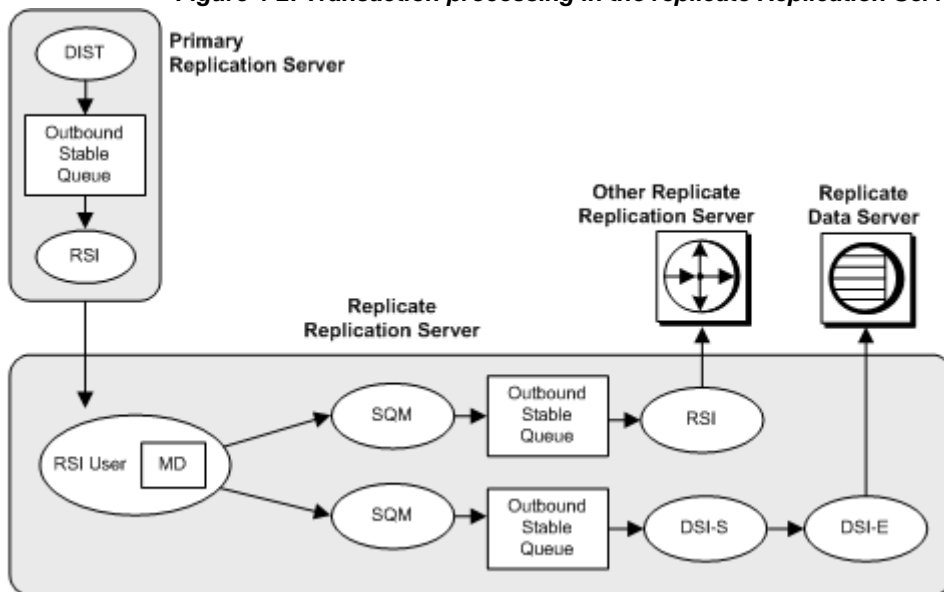
RSI user thread

The RSI user thread is a client connection thread for incoming messages from another Replication Server. It calls the Message Delivery (MD) module to determine whether to send the message to:

- A data server using the DSI thread, described in "Data Server Interface threads" on page 128. The DSI thread is composed of a scheduler thread (DSI-S) and one or more executor threads (DSI-E).

- Another Replication Server using the RSI thread, described in "Replication Server Interface thread" on page 129.

The RSI user thread writes commands destined for other Replication Servers or databases into outbound queues. See "Processing in the primary Replication Server" on page 124 for details on how messages are processed after they are stored in the outbound queues.

Configuration parameters that affect performance

Replication Server provides configuration parameters for improving performance that affect the entire server, or are targeted for individual connections or routes.

Replication Server parameters that affect performance

rs_init sets default configuration parameters after you install your Replication Server. You can change the values of the configuration parameters shown in Table 4-2 to improve Replication Server performance.

 See "Changing Replication Server parameters" on page 94 in the *Replication Server Administration Guide Volume 1* for information on how to modify these parameters using configure replication server.

Table 4-2: Replication Server parameters that affect performance

Configuration parameter	Description
deferred_queue_size	The maximum size of an Open Server deferred queue. If Open Server limits are exceeded, increase the maximum size. The value must be greater than 0.
	Note If modified, you must restart the Replication Server for the change to take effect.
	Default: 2048 on Linux and HPIA32
	1024 on other platforms
dynamic_sql	Turns dynamic SQL feature on or off. Other dynamic SQL related configuration parameters will only take effect if this parameter is set to on.
	Default: off
dynamic_sql_cache_size	Gives the Replication Server a hint on how many database objects may use the dynamic SQL statement for a connection.
	Default: 100
	Minimum: 1
	Maximum: 65536
dynamic_sql_cache_management	Manages the dynamic SQL cache for a DSI executor thread. Values: mru - keeps the most recently used statements and deallocates the rest to allocate new dynamic statements when dynamic_sql_cache_size is reached. fixed (default) - Replication Server stops allocating the new dynamic statements once dynamic_sql_cache_size is reached.
exec_cmds_per_timeslice	Specifies the number of LTL commands an LTI or RepAgent executor thread can process before yielding the CPU. By increasing this value, you allow the RepAgent executor thread to control CPU resources for longer periods of time, which may improve throughput from RepAgent to Replication Server.
	Set this parameter at the connection level using alter connection.
	See "Controlling the number of commands the RepAgent executor can process" on page 147.
	Default: 5
	Minimum: 1
	Maximum: 2,147,483,647

Configuration parameter	Description
exec_sqm_write_request_limit	Specifies the amount of memory available to the LTI or RepAgent Executor thread for messages waiting to be written to the inbound queue.
	Default: 1MB
	Minimum: 16K
	Maximum: 2GB
init_sqm_write_delay	The initial amount of time an SQM Writer should wait for more messages before writing a partially full block of messages to the queue. The SQM Writer always tries to write full blocks to the queue. If it has partially filled a block, and cannot fill it, SQM Writer waits the amount of time specified by init_sqm_write_delay before rechecking whether messages are waiting to be added to the block. If no messages exist, SQM Writer doubles the init_sqm_write_delay time. The SQM Writer continues to double the delay time until it reaches the value of init_sqm_write_max_delay. At this point, SQM Writer writes the partially full block.
	See "Setting the amount of time SQM Writer waits" on page 142.
	Default: 1000 milliseconds
init_sqm_write_max_delay	The maximum amount of time an SQM Writer thread should wait for more messages before writing a partially full block of messages to the queue. See the description of init_sqm_write_delay for more information. See also "Setting the amount of time SQM Writer waits" on page 142.
	Default: 10,000 milliseconds
md_sqm_write_request_limit	Specifies the amount of memory available to the Distributor for messages waiting to be written to the outbound queue.
	Note In Replication Server 12.1, md_sqm_write_request_limit replaces md_source_memory_pool. md_source_memory_pool is retained for compatibility with older Replication Servers.
	Default: 1MB
	Minimum: 16K
	Maximum: 2GB
memory_limit	The maximum total memory the Replication Server can use.
	Values for several other configuration parameters are directly related to the amount of memory available from the memory pool indicated by memory_limit. These include exec_sqm_write_request_limit, md_sqm_write_request_limit, queue_dump_buffer_size, sqt_max_cache_size, sre_reserve, and sts_cachesize.
	Default: 80 MB

Configuration parameter	Description
rec_daemon_sleep_time	Specifies the sleep time for the recovery daemon, which handles "strict" save interval messages in warm standby applications and certain other operations. See "Setting wake up intervals" on page 144.
	Default: 2 minutes
smp_enable	Enables symetric multiprocessing (SMP). Specifies whether Replication Server threads should be scheduled internally by Replication Server or externally by the operation system. When Replication Server threads are scheduled internally, Replication Server is restricted to one machine processor, regardless of how many may be available. Values are "on" and "off."
	See "Making SMP more effective" on page 148.
	Default: off
sqm_cache_enable	Indicates whether to enable SQM caching and large I/O in a stable device.
	Default: on
sqm_cache_size	Indicates the number of pages in cache where size of a page is specified by sqm_page_size.
	Default: 16
sqm_page_size	Indicates the number of 16K blocks in a page.
	Default: 4
sqm_recover_segs	Specifies the number of stable queue segments Replication Server allocates before updating the RSSD with recovery QID information.
	See "Specifying the number of stable queue segments allocated" on page 147.
	Default: 1
	Minimum: 1
	Maximum: 2,147,483,648
sqm_seg_prealloc	Enables or disables the segments preallocation settings.
	Default: on
sqm_write_flush	Specifies whether or not writes to memory buffers are flushed to the disk before the write operation completes. Values are "on" ,"off" and "dio".
	See also "Stable devices: considerations."
	Default: on
sqt_init_read_delay	The length of time an SQT thread sleeps while waiting for an SQM read before checking to see if it has been given new instructions in its command queue. With each expiration, if the command queue is empty, SQT doubles its sleep time up to the value set for sqt_max_read_delay.
	Default: 1 milliseconds (ms)
	Minimum: 0 ms
	Maximum: 86,400,000 ms (24 hours)

Configuration parameter	Description
sqt_max_cache_size	Maximum SQT cache memory, in bytes. See "Sizing the SQT cache" on page 145.
	Default: 1,048,576 bytes
sqt_max_read_delay	The maximum length of time an SQT thread sleeps while waiting for an SQM read before checking to see if it has been given new instructions in its command queue.
	Default: 1 ms
	Minimum: 0 ms
	Maximum: 86,400,000 ms (24 hours)
sts_cachesize	The total number of rows that are cached for each cached RSSD system table. Increasing this number to the number of active replication definitions prevents Replication Server from executing expensive table lookups.
	Monitor whether the STS cache is too small by reviewing counter 11008 – STSCacheExceed or examing the Replication Server log for warnings that rows have been removed from the STS cache. See "Caching system tables" on page 142.
	Default: 100
sts_full_cache_*table_name*	Specifies an RSSD system table that is to be fully cached. Fully cached tables do not require access to the RSSD for simple select statements.
	See "Caching system tables" on page 142 for a list of RSSD tables that can be fully cached.
sub_daemon_sleep_time	Number of seconds the subscription daemon sleeps before waking up to recover subscriptions. The range is 1 to 31,536,000.
	See "Setting wake up intervals" on page 144.
	Default: 120 seconds
sub_sqm_write_request_limit	Specifies the memory available to the subscription materialization or dematerialization thread for messages waiting to be written to the outbound queue.
	Default: 1MB
	Minimum: 16K
	Maximum: 2GB

Stable devices: considerations

Like any application, Replication Server is subject to standard I/O and I/O device best practices. You should consider the impact of contention for disk Read/Write heads and I/O channels when planning how your stable devices will be used to support your stable queues. To the extent that you can dedicate one or more devices to each queue, I/O will be less of a performance issue. This includes guarding the devices from use by other processes such as primary or replicate databases or RSSDs. You can use the database connection parameter disk_affinity to establish affinities between queues and specific partitions that are supported by dedicated devices.

For stable queues initialized on UNIX operating system files, the sqm_write_flush configuration parameter controls whether or not writes to memory buffers are flushed to the disk before the write operation completes.

When sqm_write_flush is on, Replication Server opens stable queues using the O_DSYNC flag. This flag ensures that writes are flushed from memory buffers to the disk before write operations complete. Because the data is stored on physical media, Replication Server can always recover the data in the event of a system failure. This is the default setting.

When sqm_write_flush is off, writes may be buffered in the UNIX file system. If subsequent writes fail, automatic recovery is not guaranteed. Testing has shown that when comparing the write rates of the various options for partition types and I/O flushing that writing to a buffered file system with sqm_write_flush on is up to five times slower than writes to raw partitions.

Further, testing has shown that writes to raw partitions are up to seven times slower than writes to buffered file systems with sqm_write_flush off. Turning sqm_write_flush off when using UNIX Buffered file systems for stable devices provides peak I/O performance but with an increased risk of data loss. Provided you keep primary database transaction log backups, that risk can be reduced or eliminated.

For file system partitions, direct I/O reduces the I/O latency as compared to the synchronous I/O, DSYNC. Configure direct I/O using:

```
configure replication server set sqm_write_flush to
"dio"
```

This command enables direct I/O and is effective only when the stable queue is on the file system. The direct I/O method allows the Replication Server to read or write directly to the disk without the buffering of the file system. Adjust the stable queue cache properly. A proper cache size ensures that most read transactions are completed within the cache.

Note Direct I/O is supported only on Solaris and Linux platforms for Replication Server 15.1.

This command is static, which means you must restart the server for it to take effect.

Note The sqm_write_flush setting is ignored for stable queues initialized on raw partitions or Windows files. In these cases, write operations always take place directly to media.

To improve I/O performance, Replication Server 15.1 supports caching for stable device. See "Caching stable queue" for detailed information.

Connection parameters that affect performance

Table 4-3 describes the database connection parameters that can affect performance. See Chapter 7, "Managing Database Connections," in the *Replication Server Administration Guide Volume 1* for a complete list of connection parameters.

Table 4-3: Connection parameters that affect performance

Configuration parameter	Description
batch	The default, "on," allows command batches to a replicate database.
	Default: on
db_packet_size	The maximum size of a network packet. During database communication, the network packet value must be within the range accepted by the database.
	Maximum: 16384 bytes
	Default: 512-byte network packet for all Adaptive Server databases
disk_affinity	Specifies an allocation hint for assigning the next partition. Enter the logical name of the partition to which the next segment should be allocated when the current partition is full. Values are "*partition_name*" and "off."
	Default: off

Configuration parameter	Description
dsi_cmd_batch_size	The maximum number of bytes that Replication Server places into a command batch. Default: 8192 bytes
dsi_commit_check_locks_intrvl	The number of milliseconds (ms) the DSI executor thread waits between executions of the rs_dsi_check_thread_lock function string. Used with parallel DSI. See "Using parallel DSI threads" on page 151. Default: 1000 ms (1 second) Minimum: 0 Maximum: 86,400,000 ms (24 hours)
dsi_commit_check_locks_max	The maximum number of times the DSI executor thread executes the rs_dsi_check_thread_lock function string before rolling back and retrying a transaction. Used with parallel DSI. See "Using parallel DSI threads" on page 151. Default: 400 Minimum: 1 Maximum: 1,000,000
dsi_commit_control	Specifies whether commit control processing is handled internally by Replication Server using internal tables (on) or externally using the rs_threads system table (off). Used with parallel DSI. See "Using parallel DSI threads" on page 151. Default: on
dsi_isolation_level	Specifies the isolation level for transactions. ANSI standard and Adaptive Server supported values are: • 0 – ensures that data written by one transaction represents the actual data. • 1 – prevents dirty reads and ensures that data written by one transaction represents the actual data. • 2 – prevents nonrepeatable reads and dirty reads, and ensures that data written by one transaction represents the actual data. • 3 – prevents phantom rows, nonrepeatable reads, and dirty reads, and ensures that data written by one transaction represents the actual data. Through the use of custom function strings, Replication Server can support any isolation level the replicate data server may use. Support is not limited to the ANSI standard only. Default: the current transaction isolation level for the target data server
dsi_large_xact_size	The number of commands allowed in a transaction before the transaction is considered to be large. Default: 100 Minimum: 4 Maximum: 2147483647

Configuration parameter	Description
dsi_max_xacts_in_group	Specifies the maximum number of transactions in a group. Larger numbers may reduce commit processing at the replicate database, and thereby improve throughput. Range of values: 1 – 100.
	See "Specifying the number of transactions in a group" on page 149.
	Default: 20
dsi_num_large_xact_threads	The number of parallel DSI threads to be reserved for use with large transactions. The maximum value is one less than the value of dsi_num_threads.
	Default: 0
dsi_num_threads	The number of parallel DSI threads to be used. The maximum value is 255.
	Default: 1
dsi_partitioning_rule	Specifies the partitioning rules (one or more) the DSI uses to partition transactions among available parallel DSI threads. Values are origin, origin_sessid, none, time, user, and name. See also "Partitioning rules: reducing contention and increasing parallelism" on page 162.
	Default: none
dsi_serialization_method	Specifies the method used to determine when a transaction can start, while still maintaining consistency. In all cases, commit order is preserved.
	These option methods are ordered from most to least amount of parallelism. Greater parallelism can lead to more contention between parallel transactions as they are applied to the replicate database. To reduce contention, use the dsi_partition_rule option.
	• no_wait – specifies that a transaction can start as soon as it is ready—without regard to the state of other transactions.
	• wait_for_start – specifies that a transaction can start as soon as the transaction scheduled to commit immediately before it has started.
	• wait_for_commit (default) – specifies that a transaction cannot start until the transaction scheduled to commit immediately preceding it is ready to commit.
	These options are retained only for backward compatibility with older versions of Replication Server:
	• none – same as wait_for_start.
	• single_transaction_per_origin – same as wait_for_start with dsi_partitioning_rule set to origin.
	• isolation_level_3 – same as wait_for_start with dsi_isolation_level set to 3.
dsi_sqt_max_cache_size	Maximum SQT (Stable Queue Transaction) interface cache memory for the database connection, in bytes. The default, 0, means the current setting of the sqt_max_cache_size parameter is used as the maximum cache size for the connection.
	Default: 0

Configuration parameter	Description
dsi_xact_group_size	The maximum number of bytes, including stable queue overhead, to place into one grouped transaction. A grouped transaction is a set of transactions that the DSI applies as a single transaction. –1 means no grouping.
	Sybase recommends that you set dsi_xact_group_size to the maximum value and use dsi_max_xacts_in_group to control the number of transactions in a group.
	Default: 65,536 bytes
	Maximum: 2,147,483,647
exec_cmds_per_timeslice	Specifies the number of LTL commands an LTI or RepAgent Executor thread can process before it must yield the CPU to other threads. The range is 1 to 2,147,483,648.
	Default: 5
exec_sqm_write_request_limit	Specifies the amount of memory available to the LTI or RepAgent Executor thread for messages waiting to be written to the inbound queue.
	Default: 1MB
	Minimum: 16K
	Maximum: 2GB
md_sqm_write_request_limit	Specifies the amount of memory available to the Distributor for messages waiting to be written to the outbound queue.
	Note In Replication Server 12.1, md_sqm_write_request_limit replaces md_source_memory_pool. md_source_memory_pool is retained for compatibility with older Replication Servers.
	Default: 1MB
	Minimum: 16K
	Maximum: 2GB
parallel_dsi	A shorthand method for configuring parallel DSI to default values. A value of "on" sets dsi_num_threads to 5, dsi_num_large_xact_threads to 2, dsi_serialization_method to wait_for_commit, and dsi_sqt_max_cache_size to 1 million bytes. A value of "off" sets the parallel DSI values to their defaults. You can set this parameter to "on" and then set individual parallel DSI configuration parameters to fine-tune your configuration.
	Default: off

Configuration parameter	Description
use_batch_markers	If use_batch_markers is set to on, the function strings rs_batch_start and rs_batch_end will be executed.
	Note This parameter will only need to be set to on for replicate data servers that require additional SQL translation to be sent at the beginning and end of a batch of commands that are not contained in the rs_begin and rs_commit function strings.
	Default: off

Route parameters that affect performance

Table 4-4 describes the route configuration parameters that affect performance. See Chapter 6, "Managing Routes," in the *Replication Server Administration Guide Volume 1* for a complete list of route parameters.

Table 4-4: Route parameters that affect performance

Configuration parameter	Description
rsi_batch_size	The number of bytes sent to another Replication Server before a truncation point is requested.
	Default: 256K
	Minimum: 1K
	Maximum: 128MB
rsi_packet_size	Packet size, in bytes, for communications with other Replication Servers. The range is 1024 to 16384.
	Default: 2048 bytes
rsi_sync_interval	The number of seconds between RSI synchronization inquiry messages. The Replication Server uses these messages to synchronize the RSI outbound queue with destination Replication Servers. The value must be greater than 0.
	Default: 60 seconds

Suggestions for using tuning parameters

This section provides basic recommendations for improving Replication Server performance. Whether or not changing these configuration values improves your system performance depends on your system configuration and how Replication Server is used at your site.

Setting the amount of time SQM Writer waits

Replication Server configuration parameters: init_sqm_write_delay and init_sqm_write_max_delay

In a low-volume system, set init_sqm_write_delay and init_sqm_write_max_delay to a low value so that the SQM Writer need not wait long before writing a partially full block. In a high-volume system, set these parameters higher because the SQM Writer rarely waits to fill a block.

Monitor how often the SQM Writer waits by reviewing counter 6038 – WritesTimerPop.

Determine the number of full or partially full blocks that have been written by reviewing these counters:

- 6002 – BlocksWriten

- 6041 – BlocksFullWrite

If counter 62006 – SleepsWriteQ is relatively high compared to counter 62002 – BlocksRead, SQM Readers must too often wait for the next block of messages to deliver downstream—which causes latency. Decrease the values of init_sqm_write_delay and init_sqm_write_max_delay so that SQM Writer does not wait to long before writing a partially full block.

Ideally, the ratio of counter 62004 – BlocksReadCached to counter 62002 – BlocksRead should be high, and counter 62006 – SleepsWriteQ should be relatively low. Such numbers would indicate that the SQM Writer is working approximately as fast as the SQM Reader, handing off blocks from the former to the latter without reading from disk. However, these are Replication Server–wide parameters, adjusting them to make one queue more efficient may decrease the efficiency of another.

Caching system tables

Replication Server configuration parameters: sts_cache_size and sts_full_cache_*table_name*.

You can fully cache certain system tables so that simple select statements on those tables do not require access to the RSSD. By default, rs_repobjs and rs_users are fully cached. Sybase recommends that you cache rs_objects, rs_columns, and rs_functions. Depending on the number of replication definitions and subscriptions used, fully caching these tables may significantly reduce RSSD access requirements. However, if the number of unique rows in rs_objects is approximately equal to the value for sts_cachesize, these tables may already be fully cached.

Table 4-5 lists those tables that can be fully cached.

Table 4-5: System tables that can be cached

Tables

rs_classes	rs_dbsubsets	rs_version	rs_datatype
rs_databases	rs_columns	rs_config	rs_routes
rs_objects	rs_diskaffinity	rs_functions	rs_users
rs_sites	rs_queues	rs_repdbs	rs_dbreps
rs_repobjs	rs_systext	rs_publications	

Caching stable queue

Replication Server uses a simple caching mechanism to optimize I/O. This mechanism reduces write latency and improves reader speed, since data can usually be read quickly from the cache.

A cache is made up of multiple pages and each page is made up of multiple adjoining blocks. A cache is allocated for each queue at start-up time. Changing the page size changes the size of I/O in the stable queue devices. When a page is full, the entire page is written in one single write operation.

In stable queue caching, the page pointer moves forward and rotates back at the end of the cache. SQM flushes the current page if the writer has filled the message queue and is blocked when waiting for messages. Only blocks with data are written to a disk when flushing a page that is not full.

Configuring stable queue cache parameters

Set the server-wide caching default value using:

```
configure replication server set sqm_cache_enable to
"on|off"
```

Enable or disable the caching for a queue and override the server-level setting using:

```
alter queue q_number, q_type, set sqm_cache_enable to
"on|off"
```

When `sqm_cache_enable` parameter is disabled, SQM module returns back to the earlier mechanism, which maintains a fixed 16K; one-block buffer.

Set the server-wide page size default value using:

```
configure replication server set sqm_page_size to
"num_of_blocks"
```

Set the page size for a specified queue using:

```
alter queue q_number, q_type, set sqm_page_size to
"num_of_blocks"
```

num_of_blocks specifies the number of 16K blocks in a page. Configuring the page size also sets the I/O size of Replication Server. For example, if you set the page size to 4, this instructs the Replication Server to write to stable queue in 64K chunks.

Set the server-wide cache size default value using:

```
configure replication server set sqm_cache_size to
"num_pages"
```

Set the cache size for a specified queue using:

```
alter queue q_number, q_type, set sqm_cache_size to
"num_pages"
```

num_pages specifies the number pages in the cache.

All SQM configuration commands are static, thus you must restart the server for these commands to take effect.

See the *Replication Server Reference Manual* for detailed information about these configuration parameters.

Setting wake up intervals

Replication Server configuration parameters: rec_daemon_sleep_time and sub_daemon_sleep_time

By default, the recovery and subscription daemons wake up every two minutes to check the RSSD for messages. In a typical production environment, the subscription daemon is used rarely. As a consequence, you may be able to set the subscription daemon wake-up interval to the maximum value: 31,536,000 seconds. Similarly, you can evaluate whether you want to set the recovery daemon to a longer wake-up interval.

Sizing the SQT cache

Replication Server configuration parameter: sqt_max_cache_size
Database connection configuration parameter: dsi_sqt_max_cache_size

Monitor SQT cache usage by reviewing counter 24005 – CacheMemUsed. Although this counter may indicate that the SQT cache is constantly full, sqt_max_cache_size may not need to be increased. Instead, monitor counter 24009 – TransRemoved. If TransRemoved remains zero, indicating that transactions are not being flushed from the cache to make room for others, you may not need to adjust sqt_max_cache_size.

However, sqt_max_cache_size can be set too high. Monitor counter 24019 – SQTCacheLowBnd to determine the minimum cache size before transactions are flushed. This value depends on the number and size (in terms of bytes requred to store in cache) of transactions; it varies as the transaction profile varies. Monitor this counter when there is a heavy transaction load, and set sqt_max_cache_size to the size reported by this counter plus 10 to 20%. At this size, you may see an occasional transaction removed from cache, but typically a frequency of no more than one transaction every five minutes does not introduce significant latency.

Warning! Setting the sqt_max_cache_size too high can cause the server to shutdown and can affect the overall resources of the Replication Server if the server memory_limit is not set high enough to accommodate the sqt cache sizing.

sqt_max_cache_size applies to all SQT caches supporting DIST clients, and provides a default value for SQT caches that support DSI clients. The DISTs can push through transactions rapidly; their SQT caches do not need to be as large as SQT caches for DSIs. Thus, it is advisable to set SQT cache sizes for DSIs individually using the connection configuration parameter dsi_sqt_max_cache_size, and using sqt_max_cache_size for DIST SQT caches only.

Note In versions of Replication Server earlier than 12.6, You must increase sqt_max_cache_size to ensure that many closed transactions were ready to be distributed or sent to the replicate database when resources became available. With Replication Server 12.6 and later, this advice no longer applies.

Controlling the number of network operations

Database connection configuration parameter: dsi_cmd_batch_size

dsi_cmd_batch_size controls the size of a DSI command batch. That is, it controls the size of the buffer a DSI uses to send commands to a replicate data server. When the DSI configuration batch is set on, the DSI places as many commands as will fit into a single command batch before sending it to the replicate. In some cases, increasing the value of dsi_cmd_batch_size improves throughput by providing the replicate database with more work per command batch.

You can monitor the average size of a batch by referring to counter 57076 – DSIEBatchSize. You can monitor the average amount of time taken to process a batch (the time from when the batch is created until it is flushed and the results processed) by referring to counter 57070 – DSIEBatchTime.

The following counters may also be useful in monitoring the effectiveness of batching and batch size:

57037 – SendTime	57079 – DSIEOCmdCount	57063 – DSIEResultTime
57070 – DSIEBatchTime	57092 – DSIEBFMaxBytes	57076 – DSIEBatchSize

Controlling the number of outstanding bytes

Database connection configuration parameters: exec_sqm_write_request_limit and md_sqm_write_request_limit

exec_sqm_write_request_limit controls the maximum number of outstanding bytes the RepAgent User thread can hold before it must wait for some of those bytes to be written to the inbound queue. Similarly, md_sqm_write_request_limit controls the number of outstanding bytes a DIST thread can hold before it must wait for some of those bytes to be written to the outbound queue.

Monitor the number of times and duration of time the RepAgent Executor sleeps while waiting for outstanding write requests to complete by reviewing this counter:

• 58019 – RAWriteWaitsTime

If the RepAgent Executor consistently reaches this threshold, review the StableDevice I/O.

Controlling the number of commands the RepAgent executor can process

Database connection configuration parameter: exec_cmds_per_timeslice

By default, the value of the exec_cmds_per_timeslice parameter is 5, which indicates that the RepAgent executor thread can process no more than five commands before it must yield the CPU to other threads. Depending on your environment, increasing or decreasing these values may improve performance.

If the in-bound queue is slow to be processed, try increasing these values to give the RepAgent executor thread and the DIST thread more time to perform their work. If, however, the out-bound queue is slow to be processed, try decreasing these parameter values so that the DSI has more time to work.

If CPU resources are limited with respect to the number of connections Replication Server supports, increasing the value of exec_cmds_per_timeslice may result in decreased overall performance. In this case, giving the RepAgent Executor more control of CPU resources may reduce resources to other Replication Server threads.

Monitor the number of times and duration of time the RepAgent executor thread yields CPU with this counter:

• 58016 – RAYieldTime

Specifying the number of stable queue segments allocated

Replication Server configuration parameter: sqm_recover_segs

sqm_recover_segs specifies the number of stable queue segments Replication Server allocates before updating the RSSD with recovery QID information.

If sqm_recover_segs is set low, more RSSD updates are performed, possibly slowing performance. If sqm_recover_segs is set high, fewer RSSD updates are performed, possibly improving performance at the expense of longer recovery times.

Monitor how often an SQM Writer makes updates to the rs_oqids table by reviewing counter 6036 – UpdsRsoqid. Typically, increasing the value of sqm_recover_segs improves performance by reducing the amount of time and system resources necessary to allocate segments. However, queue startup and restart take longer as the SQM Writer must scan more of the queue to determine the last message successfully written for each origin. Each segment requires 1MB of queue space; determine the value of sqm_recover_segs by calculating the number of megabytes the SQM Writer can afford to scan at startup or restart. For example, if the SQM Writer can scan 50MB of queue without slowing Replication Server startup or restart, set sqm_recover_segs to 50.

Selecting disk partitions for stable queues

Database connection configuration parameter: disk_affinity

The Replication Server partition affinity feature (see "Allocating queue segments" on page 182) allows you to choose the disk partition to which Replication Server allocates segments for stable queues. Sybase suggests that to improve overall throughput, you associate faster devices with stable queues that process more slowly.

Making SMP more effective

Replication Server configuration parameter: smp_enable

To determine the number of processors required to make effective use of SMP, establish a base of two processors plus one more for every four queues. Processor speed may determine whether these numbers are correct to meet your performance needs. If you have outbound queues supporting parallel DSI, and there are more than 12 DSI Executor threads, you may want to increase the processor/thread ratio for outbound queues—one processor for every three or even two outbound queues.

Replication Server always uses a finite number of threads based on the number of supported connections and routes. Even if all threads are to be kept always busy, making more and more processors available to Replication Server will eventually cause "CPU saturation"—beyond which more processors will not increase performance. At that point, any performance issues you experience as a result of CPU resources may best be addressed by introducing CPUs running at faster speeds.

In some cases, there is evidence that making too many processors available to Replication Server can actually decrease performance. In such cases, the issue seems to be the amount of time taken to force thread context switches among the available processors. Use your operating system (OS) monitoring utilities to monitor the OS's management of the Replication Server process and its threads. These utilities will help you determine if a reduction in CPUs made available to Replication Server reduces the number of such context switches.

Specifying the number of transactions in a group

You can use different configuration parameters to control the number of transactions in a group.

Database configuration parameter: *dsi_max_xacts_in_group*

dsi_max_xacts_in_group specifies the maximum number of transactions in a group. Larger numbers may reduce commit processing at the replicate database, and thereby improve throughput.

Monitor the average number of transactions placed in a group per DSI-E thread by reviewing counter 57001 – UnGroupedTransSched.

Monitor the average number of transactions per group for the total DSI connection by reviewing these counters:

- 5000 – DSIReadTranGroups

- 5002 – DSIReadTransUngrouped

Use dsi_max_xacts_in_group to control group size. Set dsi_xact_group_size to the maximum value of 2147483647 and do not change it. Contention among parallel transactions may be reduced by reducing the value of dsi_max_cacts_in_group to 1, which indicates no grouping.

Monitor why groups are being closed by reviewing these counters:

- 5042 – GroupsClosedBytes

- 5043 – GroupsClosedNoneOrig

- 5044 – GroupsClosedMixedUser

- 5045 – GroupsClosedMixedMode

- 5049 – GroupsClosedTranPartRule

- 5051 – UserRuleMatchGroup

- 5053 – TimeRuleMatchGroup
- 5055 – NameRuleMatchGroup
- 5063 – GroupsClosedTrans
- 5068 – GroupsClosedLarge
- 5069 – GroupsClosedWSBSpec
- 5070 – GroupsClosedResume
- 5071 – GroupsClosedSpecial
- 5072 – OriginRuleMatchGroup
- 5074 – OSessIDRuleMatchGroup
- 5076 – IgOrigRuleMarchGroup

Database configuration parameters: *dsi_xact_group_size* and *dsi_max_xacts_in_group*

Use these configuration parameters together to increase the number of transactions that can be grouped as a single transaction for application to the replicate database. If the average number of commands per transaction is small (five or fewer), you can use dsi_xact_group_size and dsi_max_xact_in_group to improve transaction application time.

Sybase recommends that you set dsi_xact_group_size to the maximum value, and use dsi_max_xact_in_group to control transaction group size.

Setting transaction size

For single DSI connections, set the value of dsi_large_xact_size to the maximum value of 21474836467. Even when parallel DSI is not configured, the DSI/S reads the statement limit set by dsi_large_xact_size and performs several tasks related to parallel DSI.

Using parallel DSI threads

You can configure a database connection so that transactions are applied to a replicate data server using parallel DSI threads rather than a single DSI thread. Applying transactions in parallel increases the speed of replication, yet maintains the serial commit order of the transactions that occurred at the primary site.

When parallel DSI threads are active, Replication Server normally starts processing a transaction *before* the preceding transaction has committed and *after* the DSI has seen the commit record for the next transaction. The commit is delayed until it is determined that all preceding transactions have committed. Replication Server can maintain the order in which transactions are committed and detect conflicting updates in transactions that are executing in parallel simultaneously, using either of these methods:

*   Internally, using Replication Server internal tables and function strings, or

*   Externally, using the rs_threads system table in the replicate database.

Replication Server can achieve additional parallelism in the way it processes transactions containing a large number of operations with parallel DSI threads. Large transactions begin processing *before* the DSI has seen the commit record. While this means a large transaction can be processed sooner, it also means that in a warm standby situation, Replication Server might start processing a transaction that is ultimately rolled back. However, with subscription replication, the rollback transaction would be caught by the DIST thread.

Replication Server provides other options for maximizing parallelism and minimizing contentions between transactions. For example:

*   Transaction serialization methods allow you to choose the degree of parallelism your system can handle without inducing conflicts.

*   Transaction partitioning rules provide additional tuning to affect how transactions are grouped and distributed to avoid contention in the replicate database.

Benefits and risks

For most primary databases, many users and applications can create transactions simultaneously. Funneling all of these transactions to the replicate through a single connection can create a serious bottleneck. This bottleneck can cause periods of unwanted latency between the primary and the replicate.

The benefit of enabling parallel DSI within Replication Server is to reduce this potential bottleneck by processing multiple transactions across multiple replicate databases at the same time.

The risk in enabling parallel DSI is the introduction of contention between the multiple replicate connections and their transactions. The simultaneous application of transactions against the replicate may introduce competition between the transactions for replicate resources, creating a different kind of bottleneck.

As a result, using parallel DSI threads successfully requires an in-depth knowledge of your replication environment and iterative testing to determine which of the parallel DSI tuning parameters are most beneficial. The objective is to provide high throughput while controlling the amount of contention introduced at the replicate.

For example, consider a body of work that includes 1000 transactions that must be replicated. It will take some time to send all 1000 transactions across a single replicate connection. However, attempting to configure and use 1000 connections, one for each transaction, will likely result in contentions and strained server resources. A successful configuration requires a balance between the two scenarios; it depends on both the transaction profile and the impact of issuing those transactions against the replicate using parallel DSI.

In a second example, two serial transactions issued at the primary each perform a single update operation to the same table row. If these two transactions are attempted in parallel at the replicate by two connections, the first transaction to access the table row is granted exclusive access. The second transaction must wait until the first transaction has either committed or rolled back and thus released the row. Although both transactions are ultimately applied, there is no benefit from the parallel DSI configuration. The transactions are processed serially in the same way they would have been processed without parallel DSI. The contention has nullified any benefit from using parallel DSI.

Parallel DSI parameters

You can customize the parallel DSI thread environment using the configuration parameters shown in Table 4-6. Use these configuration parameters with alter connection to tune parallel DSI threads for individual connections.

Table 4-6: Parallel DSI configuration parameters

Configuration parameter	Description
dsi_commit_check_locks_intrvl	The number of milliseconds (ms) the DSI executor thread waits between executions of the rs_dsi_check_thread_lock function string.
	Default: 1000 ms (1 second)
	Minimum: 0
	Maximum: 86,400,000 ms (24 hours)
dsi_commit_check_locks_log	The number of times the DSI executor thread executes the rs_dsi_check_thread_lock function string before logging a warning message.
	Default: 200
	Minimum: 1
	Maximum: 1,000,000
dsi_commit_check_locks_max	The maximum number of times the DSI executor thread executes the rs_dsi_check_thread_lock function string before rolling back and retrying a transaction.
	Default: 400
	Minimum: 1
	Maximum: 1,000,000
dsi_commit_control	Specifies whether commit control processing is handled internally by Replication Server using internal tables (on) or externally using the rs_threads system table (off).
	Default: on
dsi_ignore_underscore_names	When the dsi_partitioning_rule is set to "name," specifies whether or not Replication Server ignores transaction names that begin with an underscore. Values are "on" and "off."
	Default: on
dsi_isolation_level	Specifies the isolation level for transactions. ANSI standard and Adaptive Server supported values are:
	• 0 – ensures that data written by one transaction represents the actual data.
	• 1 – prevents dirty reads and ensures that data written by one transaction represents the actual data.
	• 2 – prevents nonrepeatable reads and dirty reads, and ensures that data written by one transaction represents the actual data.
	• 3 – prevents phantom rows, nonrepeatable reads, and dirty reads, and ensures that data written by one transaction represents the actual data.
	Through the use of custom function strings, Replication Server can support any isolation level the replicate data server may use. Support is not limited to the ANSI standard only.
	Default: the current transaction isolation level for the target data server

Configuration parameter	Description
dsi_large_xact_size	The number of statements allowed in a transaction before it is considered to be a large transaction. Default: 100 Minimum: 4
dsi_num_large_xact_threads	The number of parallel DSI threads to be reserved for use with large transactions. The maximum value is one less than the value of dsi_num_threads. Default: 0
dsi_num_threads	The number of parallel DSI threads to be used for a connection. A value of 1 disables the parallel DSI feature. Default: 1 Minimum: 1 Maximum: 255
dsi_partitioning_rule	Specifies the partitioning rules (one or more) the DSI uses to partition transactions among available parallel DSI threads. Values are origin, origin_sessid, time, user, name, none, and ignore_origin. See "Partitioning rules: reducing contention and increasing parallelism" on page 162 for detailed information. Default: none
dsi_serialization_method	Specifies the method used to determine when a transaction can start, while still maintaining consistency. In all cases, commit order is preserved. These option methods are ordered from most to least amount of parallelism. Greater parallelism can lead to more contention between parallel transactions as they are applied to the replicate database. To reduce contention, use the dsi_partition_rule option. • no_wait – specifies that a transaction can start as soon as it is ready, without regard to the state of other transactions. • wait_for_start – specifies that a transaction can start as soon as the transaction scheduled to commit immediately before it has started. • wait_for_commit (default) – specifies that a transaction cannot start until the transaction scheduled to commit immediately preceding it is ready to commit. These options are retained only for backward compatibility with earlier versions of Replication Server: • none – same as wait_for_start. • single_transaction_per_origin – same as wait_for_start with dsi_partitioning_rule set to origin. • isolation_level_3 – same as wait_for_start with dsi_isolation_level set to 3.

Configuration parameter	Description
dsi_sqt_max_cache_size	The maximum SQT cache size for the database connection. The default, 0, means the current setting of the sqt_max_cache_size parameter is used as the maximum cache size for the connection.
	See "Sizing the SQT cache" on page 145 for more information about setting the SQT cache size.
	Default: 0
parallel_dsi	A shorthand method for configuring parallel DSI to default values. A value of "on" sets dsi_num_threads to 5, dsi_num_large_xact_threads to 2, dsi_serialization_method to wait_for_commit, and dsi_sqt_max_cache_size to 1 million bytes. A value of "off" sets the parallel DSI values to their defaults. You can set this parameter to "on" and then set individual parallel DSI configuration parameters to fine-tune your configuration.
	Default: off

To configure a connection for parallel DSI, set the parallel_dsi parameter to on and then set individual parallel DSI configuration parameters to fine-tune your environment.

For example, to enable parallel DSI for the connection to the pubs2 database on the SYDNEY_DS data server, enter:

```
alter connection to SYDNEY_DS.pubs2
 set parallel_dsi to 'on'
```

See "Configuring parallel DSI for optimal performance" on page 173 for guidelines on configuring the parameters.

Components of parallel DSI

Figure 4-3 shows the components of parallel DSI.

Figure 4-3: Parallel DSI components
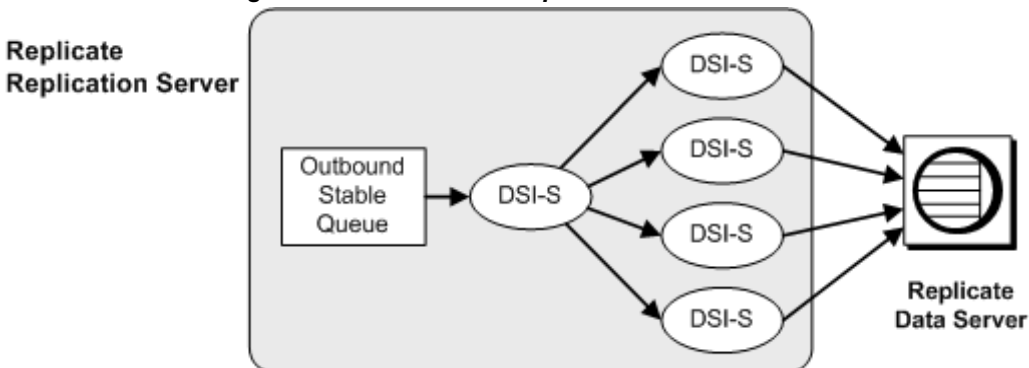
DSI scheduler thread

The DSI scheduler thread (shown as DSI-S in Figure 4-3) collects small transactions into groups by commit order. Once transactions are grouped, the DSI scheduler dispatches the groups to the next available DSI executor thread. The DSI scheduler attempts to dispatch groups for different origins in parallel, because they can commit in parallel. If contention between transactions from different origins is too high, set the ignore_origin option for the dsi_partitioning_rule parameter.

Transaction partitioning rules allow you to specify additional criteria the DSI scheduler can use to group transactions. See "Partitioning rules: reducing contention and increasing parallelism" on page 162.

DSI executor threads

The DSI executor threads (shown as DSI-E in Figure 4-3) map functions to function strings and execute the transactions on the replicate database. The DSI executor threads also take action on any errors the replicate data server returns.

Processing transactions with parallel DSI threads

You can define large and small transactions with the dsi_large_xact_size database connection configuration parameter. dsi_large_xact_size specifies the number of commands allowed in a transaction before the transaction is considered to be large. Replication Server normally processes small and large transactions differently.

Small transactions

Replication Server attempts to group similar transactions to process them as one, larger transaction. In this way, Replication Server can issue one commit for the group rather than committing each individual transaction. A group of transactions is complete and sent to the next available DSI executor thread when one of several criteria is met. For example:

- The next transaction has been issued from a different origin.

- The number of transactions in the group exceeds the value specified by dsi_max_xacts_in_group.

- The total size, in bytes, of the transactions in the group exceeds the value specified by dsi_xact_group_size.

- The next transaction is a large transaction, which is always grouped by itself.

- A transaction partitioning rule determines that the next transaction cannot be grouped with the existing group.

Once a group is complete, it can be sent to the next available DSI executor thread. Only committed transactions can be added to a group. That is, transactions are not added to the transaction group until their commit record is read.

Large transactions

Large transactions are submitted to the next available DSI executor thread that is reserved for a large transaction. The DSI executor thread sends the transaction to the replicate data server without waiting to see the commit record. If the transaction was rolled back at the primary data server, the DSI executor thread rolls it back at the replicate data server.

If Replication Server encounters a large transaction, and a dedicated large transaction thread is not available, the transaction is processed in the same way as a small transaction.

Selecting isolation levels

By selecting a transaction isolation level, you can control the degree to which data can be accessed by other users during a transaction. The ANSI SQL standard defines four levels of isolation for transactions. Each isolation level specifies the kinds of actions that are not permitted while concurrent transactions are processing. Higher levels include the restrictions imposed by lower levels. For more information about isolation levels, see the *Adaptive Server Enterprise Transact-SQL Guide*.

Note Replication Server supports not just the ANSI standard values, but all values needed to replicate to any supported data servers.

- Level 0 – prevents other transactions from changing data that has already been modified by an uncommitted transaction. However, other transactions can still read the uncommitted data, which results in dirty reads.

- Level 1 – prevents dirty reads, which occur when one transaction modifies a row, and a second transaction reads that row before the first transaction commits the change.

- Level 2 – prevents nonrepeatable reads, which occur when one transaction reads a row and a second modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.

- Level 3 – ensures that data read by one transaction is valid until the end of the transaction. It prevents "nonrepeatable reads" and "phantom rows" by applying an index page or table lock until the end of the transaction.

  Select isolation level 3 if you are using triggers to enforce referential integrity of data across a database. Isolation level 3 prevents phantom rows from occurring in a table while a trigger is executing.

You can set the isolation level using create connection or configure connection with the dsi_isolation_level option. For example, to change the isolation level to 3 for the connection to the pubs2 database on the SYDNEY_DS data server, enter:

```
alter connection to SYDNEY_DS.pubs2
```

```
set dsi_isolation_level to '3'
```

Note Isolation levels may vary depending on the replicate data server. The rs_set_isolation_level function string must be edited for non-Sybase replicate data servers, and include the *rs_isolation_level* system-defined variable. See the *Replication Server Reference Manual* for more information about rs_set_isolation_level.

Replication Server sets the isolation-level value to the rs_set_isolation_level function string using the *rs_isolation_level* system variable. rs_set_isolation_level executes when Replication Server establishes the connection with the replicate data server. If no value has been set, Replication Server does not execute rs_dsi_isolation_level, and instead uses the isolation level of the data server. The default isolation level for Adaptive Server is 1.

If you are using a data server other than Adaptive Server, make sure you include the *rs_isolation_level* variable when you modify the rs_set_isolation_level function string for your data server.

Transaction serialization methods

Replication Server provides four different serialization methods for specifying the level of parallelization. The serialization method you choose depends on the amount of contention you expect between parallel threads and your replication environment. Each serialization method defines how much of a transaction can start before it must wait for the previous transaction to commit.

Use the dsi_partitioning_rule parameter to reduce the probability of contention without reducing the degree of parallelism assigned by the serialization method. See "Partitioning rules: reducing contention and increasing parallelism" on page 162.

The serialization methods are:

• no_wait

• wait_for_start

• wait_for_commit

Use the alter connection command with the dsi_serialization_method parameter to select the serialization method for a database connection. For example, enter the following command to select the wait_for_commit serialization method for the connection to the pubs2 database on the SYDNEY_DS data server:

```
alter connection to SYDNEY_DS.pubs2
 set dsi_serialization_method to 'wait_for_commit'
```

A transaction contains three parts:

• The beginning

• The body of the transaction, consisting of operations such as insert, update, or delete

• The end of the transaction, consisting or a commit or a rollback

While providing commit consistency, the serialization method defines whether the beginning of the transaction waits for the previous transaction to become ready to commit or if the beginning of the transaction can be processed earlier.

no_wait

This method instructs the DSI to initiate the next transaction without waiting for the previous transaction to commit. It assumes that your primary applications are designed to avoid conflicting updates, or that dsi_partitioning_rule is used effectively to reduce or eliminate contention. Adaptive Server does not hold update locks unless dsi_isolation_level has been set to 3. The method assumes little contention between parallel transactions and results in the nearly parallel execution shown in Figure 4-6.

Figure 4-4: Thread timing with the no_wait serialization method

Transaction A

| | Body | End |

Transaction B

| | Body | End |

Transaction C

| | Body | End |
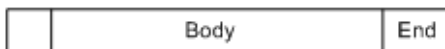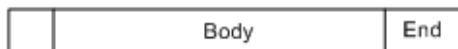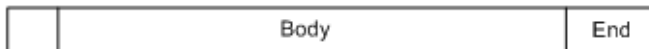
Time

no_wait provides the better opportunity for increased performance, but also provides the greater risk of creating contentions.

wait_for_start

wait_for_start specifies that a transaction can start as soon as the transaction scheduled to commit immediately before it has started. See Figure 4-5.

Sybase recommends that you do not concurrently set dsi_serialization_method to wait_for_start and dsi_commit_control to off.

Figure 4-5: Thread timing with wait_for_start serialization method
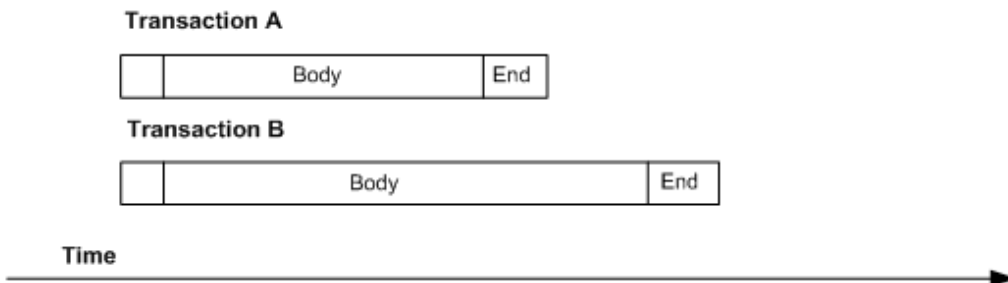
wait_for_commit

In this method, the next thread's transaction group is not sent for processing until the previous transaction has processed successfully and the commit is being sent. This is the default setting. It assumes considerable contention between parallel transactions and results in the staggered execution shown in Figure 4-6.

Figure 4-6: Thread timing with wait_for_commit serialization method
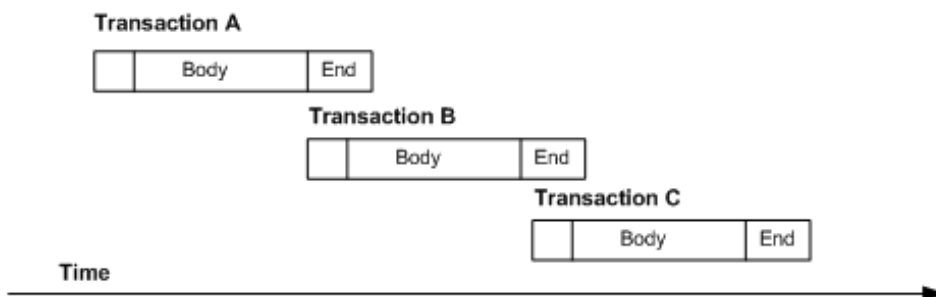
This method maintains transaction serialization by instructing the DSI to wait until a transaction is ready to commit before initiating the next transaction. The next transaction can be submitted to the replicate data server while the first transaction is committing, since the first transaction already holds the locks that it requires.

Partitioning rules: reducing contention and increasing parallelism

Another parallel DSI tuning parameter is dsi_partitioning_rule. Partitioning rules set using dsi_partitioning_rule allow the parallel DSI feature to make decisions about transaction groups and parallel execution based on transactions having common names, users, overlapping begin/commit times, or a combination of these. Partitioning rules allow the parallel DSI feature to more closely mimic processing order at the primary, and are intended to be used in reducing contention at the replicate.

Each of the parallel DSI parameters provides a method for fine-tuning the feature based on conditions at your installation. dsi_num_threads controls the number of DSI threads available for a connection. dsi_serialization_method controls the amount of parallelism for the connection, but must balance increased parallelism with the potential for contentions at the replicate. dsi_partitioning_rule provides a method for reducing contentions without reducing the overall capabilities of the parallel DSI feature.

Using transaction-partitioning rules

Replication Server allows you to partition transactions for each connection according to one or more of these attributes:

- Origin

- Origin and session ID

- None, in which no partitioning rule is applied

- User name

- Origin begin and commit times

- Transaction name

- Ignore origin

Note If partitioning rules are to be used to improve performance, dsi_serialization_method must not be wait_for_commit. wait_for_commit removes contention by reducing parallelism.

To select partition rules, use the alter connection command with the dsi_partitioning_rule option. The syntax is:

    alter connection to *data_server.database*
        set dsi_partitioning_rule to '{ none|*rule*[, *rule*] }'

Values for *rule* are user, time, origin, origin_sessid, name, and ignore_origin.

For example, to partition transactions according to user name and origin begin and commit times, enter:

```
alter connection to TOKYO_DS.pubs2
    set dsi_partitioning_rule to 'user,time'
```

Partitioning rule: origin

origin causes transactions from the same origin to be serialized when applied to the replicate database.

Partitioning rule: origin and process ID

origin_sessid causes transactions with the same origin *and* the same process ID to be serialized when applied to the replicate database. Sybase recommends that when first trying partitioning rules start with a setting of origin_sessid,time.

Note The process ID for Application Server is the Session Process ID (SPID).

Partitioning rule: none

none is the default behavior, in which the DSI scheduler assigns each transaction group or large transaction to the next available parallel DSI thread.

Partitioning rule: user

If you choose to partition transactions according to user name, transactions entered by the same primary database user ID are processed serially. Only transactions entered by different user IDs are processed in parallel.

Use of this partitioning rule avoids contentions, but may in some cases cause unnecessary loss of parallelism. For example, consider a DBA who is running multiple batch jobs. If the DBA submits each batch job using the same user ID, Replication Server processes each one serially.

The user name partitioning rule is most useful if each user connection at the primary has a unique ID. It is less useful if multiple users log on using the same ID, such as "sa." In such cases, orig_sessid may be a better option.

Partitioning rule: origin time begin and commit times

If the time partitioning rule is used, the DSI scheduler looks at the origin begin and commit times of transactions to determine which transactions could *not* have been executed by the same process at the primary database. A transaction whose origin begin time is earlier than the commit time of the preceding transaction can be processed by a different DSI executor thread.

Suppose the origin begin and commit times partitioning rule has been selected, and the transactions and processing times shown in Figure 4-7 are all from the same primary database.

Figure 4-7: Transaction origin begin and commit times
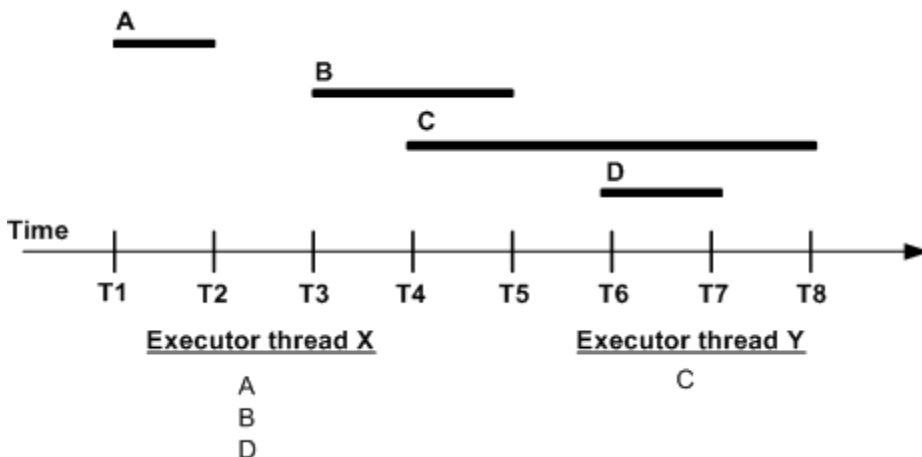
In this example, the DSI scheduler gives transaction A to DSI executor thread X. The scheduler then compares the begin time of transaction B and the commit time of transaction A. As transaction A has committed before transaction B begins, the scheduler gives transaction B to executor thread X. That is, transactions A and B may be grouped together and may be processed by the same DSI executor thread. Transaction C, however, begins before transaction B commits. Therefore, the scheduler assumes that transactions B and C were applied by different processes at the primary, and gives transaction C to executor thread Y. Transactions B and C are not allowed in the same group and may be processed by different DSI executor threads. Because transaction D begins before transaction C commits, the scheduler can safely give transaction D to executor thread X.

Note Use of the origin begin and commit times partitioning rule may lead to contentions when large transactions are processed, as they are scheduled before the commits are seen.

Partitioning rule: name

The DSI scheduler can use transactions names to group transactions for serial processing. When creating a transaction on Adaptive Server, you can use the begin transaction command to assign a transaction name.

If the transaction name partitioning rule is applied, the DSI scheduler assigns transactions with the same name to the same executor thread. Transactions with different transaction names are processed in parallel. Transactions with a null or blank name are ignored by the name parameter. Their processing is determined by other DSI parallel processing parameters or the availability of other executor threads.

Note This partitioning rule is available to non-Sybase data servers only if they support transaction names.

Default transaction names

By default, Adaptive Server always assigns a name to each transaction. If a name has not been assigned explicitly using begin transaction, Adaptive Server assigns a name that begins with the underscore character and includes additional characters that describe the transaction. For example, Adaptive Server assigns a single insert command the default name "_ins."

Use the dsi_ignore_underscore_name option with alter connection to specify whether or not Replication Server ignores these names when partitioning transactions based on transaction name. By default, dsi_ignore_underscore_name is on, and Replication Server treats transactions with names that begin with an underscore in the same way it treats transactions with null names.

Partitioning rule: ignore origin

All partitioning rules, except ignore_origin, allow transactions from different origins to be applied in parallel, regardless of other specified partitioning rules. For example:

```
alter connection dataserver.db
    set dsi_partitioning_rule to "name"
```

In this case, transactions with different origins are applied in parallel, whether or not they have the same name.

The name partitioning rule only affects transactions from the same origin. Thus, transactions with the same origin and name are applied serially, and transactions with the same origin and different names are applied in parallel.

ignore_origin overrides the default handling of transactions from different origins, and allows them to be partitioned as if they all came from the same origin.

If ignore_origin is listed first in the alter connection statement, Replication Server partitions transactions with the same or different origins according to the second or succeeding rules in the statement. For example:

```
alter connection dataserver.db
    set dsi_partitioning_rule to "ignore_origin, name"
```

In this case, all transactions with the same name are applied serially and all transactions with different names are applied in parallel. The origin of the transaction is irrelevant.

If ignore_origin is listed in the second or a succeeding position in the alter connection statement, Replication Server ignores it.

Using multiple transaction rules

You can set multiple transaction rules for a single connection. For example, applying both origin session ID and origin begin and commit times best approximates the processing environment at the primary database.

When more than one transaction rule is specified, Replication Server applies the rules in the order in which they are entered in the alter connection set dsi_partitioning_rule syntax.

For example, if dsi_partitioning_rule is set to "time, user," Replication Server checks origin begin and commit times before checking user ID. If no conflict exists for origin begin and commit times, Replication Server checks user ID. If there is a conflict involving begin and commit times, Replication Server applies the time rule without checking the user ID. Thus, two transactions will be assigned to different parallel DSI threads if the origin begin time of the later transaction is earlier than the commit time—even if both transactions have the same user ID.

Grouping logic and transaction partitioning rules

Partitioning rules can affect grouping as well as scheduling decisions. When no partitioning rule is applied, a group is complete when, for example, the maximum size for a group is reached or a large transaction is encountered.

If a partitioning rule determines that two transactions occurred at overlapping times (time rule), have different transaction names (name rule), or are from different users (user rule), the two transactions are not allowed in the same group. Otherwise, normal group-size decisions are applied, based on transaction size, origin, and so forth. See "Small transactions" on page 157.

Resolving conflicting updates

Parallel DSI processing must duplicate the commit order of transactions at the primary database, yet allow transaction updates to process simultaneously. It must then resolve any transaction contentions that occur as a result. Commit order deadlock transaction contentions—or contention deadlocks—can occur when a transaction cannot commit because it must wait for an earlier transaction to commit, and the earlier transaction cannot commit because needed resources are locked by the later transaction.

For example, DSI threads A and B are processing transactions in parallel. Thread A's transaction must commit before thread B's transaction. Thread B's transaction locks resources needed by thread A. Thread B's transaction cannot commit until thread A's transaction commits, and thread A's transaction cannot commit because needed resources are locked by thread B.

Replication Server provides two methods for resolving commit order deadlocks:

- Internally, using Replication Server internal tables and a function string, or

- Externally, using the rs_threads system table in the replicate database and several function strings.

The internal method is handled primarily within Replication Server, and uses the rs_dsi_check_thread_lock function string for commit order deadlock detection. The external method requires both Replication Server and the replicate database, and uses the rs_threads system table for both commit order validation and commit order deadlock detection.

Sybase recommends the internal method, which is the default, for both Sybase and non-Sybase data servers. This method requires less network I/O than the external method, and, if a commit order deadlock occurs, may require the rollback of only a single transaction. The external method requires more network I/O and results in the rollback of several transactions. The external method is included for compatibility with earlier versions of Replication Server.

If Replication Server encounters commit order deadlock and dsi_commit_control is on, Replication Server rolls back and retries one transaction. If, however, Replication Server encounters commit order deadlock and dsi_commit_control is off, Replication Server rolls back and retries all transactions serially.

To select a method, enter the alter connection command with the dsi_commit_control option. For example, to choose the internal method for the pubs2 database on the TOKYO_DS data server, enter:

```
alter connection to TOKYO_DS.pubs2
set dsi_commit_control to 'on'
```

Setting dsi_commit_control to "on" specifies the internal method; setting dsi_commit_control to "off" specifies the external method.

Resolving conflicts internally using the *rs_dsi_check_thread_lock* function string

To preserve transactional integrity, Replication Server must maintain transaction commit order and resolve commit order consistency deadlocks. Figure 4-8 describes the logic Replication Server uses to resolve commit order deadlocks using the rs_dsi_check_thread_lock function string.

Figure 4-8: Conflict resolution logic using the rs_dsi_check_thread_lock function string

---
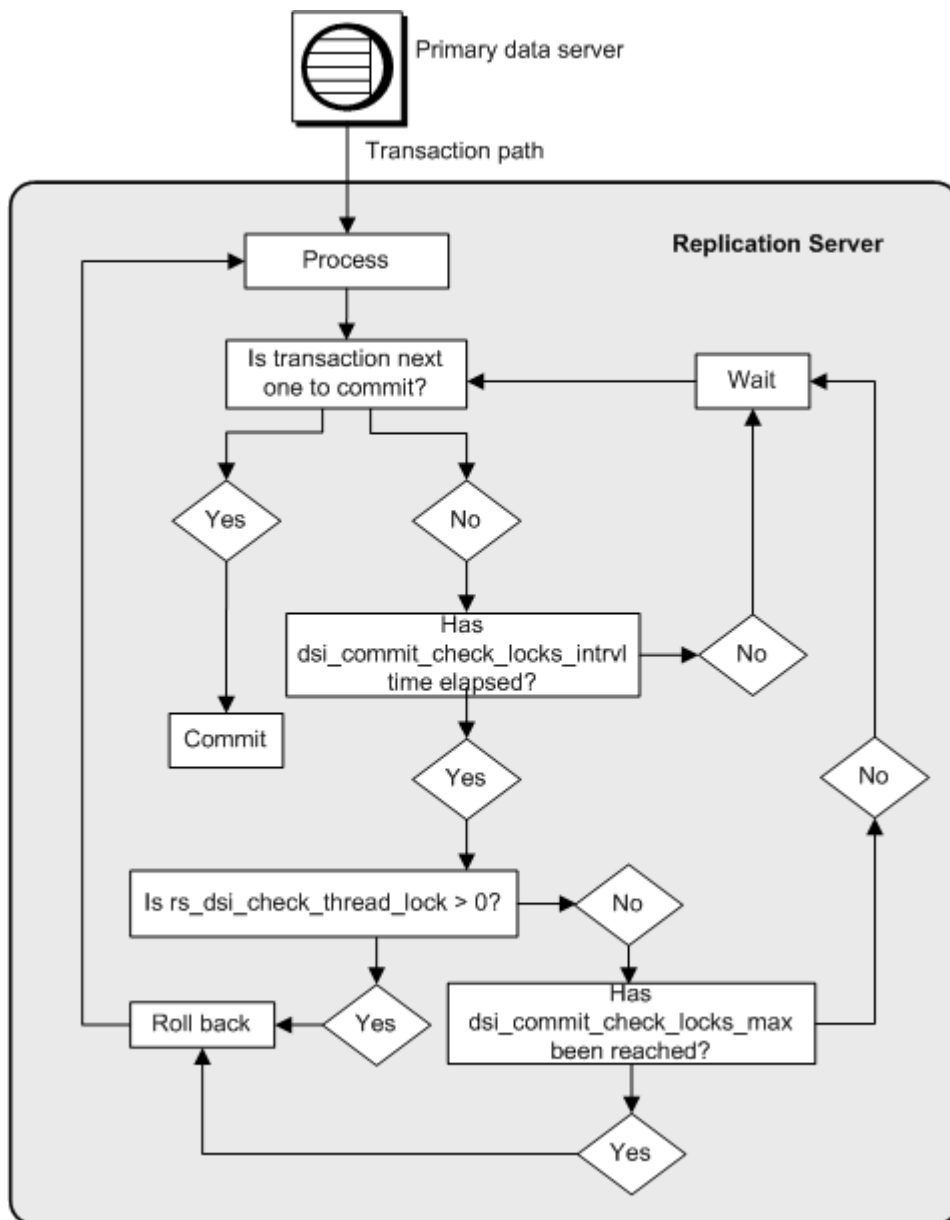
Note The internal method resolves commit order deadlocks that Replication Server detects and resolves conflicting updates only within Replication Server. If a deadlock is detected by the replicate database, the replicate chooses a transaction to roll back. To guarantee commit order, Replication Server must roll back all transactions currently executing against the replicate database. Replication Server then reapplies the transactions serially.

Maintaining commit order

Replication Server reads the commit information sent from the primary database and uses this information to define and maintain the transaction commit order at the replicate database.

If a DSI executor thread's transaction processing is complete and it is expected to be the "next" transaction to commit, it is allowed to do so. If a thread's transaction processing is complete and it is *not* the "next" transaction expected to commit, the thread must await its turn to commit.

Resolving commit consistency deadlocks

If a thread's transaction processing is complete and it is not the next transaction expected to commit, the transaction could be holding resources required by a transaction scheduled to commit earlier. See Figure 4-8 on page 169. After waiting the amount of time specified in the dsi_commit_check_locks_intrvl parameter, a DSI executor thread executes the rs_dsi_commit_check_thread_lock function string to determine if the thread holds a lock on resources needed by the earlier transaction:

- If the thread is blocking another transaction (rs_dsi_check_thread_lock > 0), the current transaction rolls back, which resolves the commit order deadlock and allows the earlier transaction to commit. Only the blocking transaction rolls back; other transactions process normally.

- If the thread is not blocking another transaction, it checks to see if it has executed rs_dsi_check_thread_lock more times than is defined by the dsi_commit_check_locks_max parameter.

  - If the thread has not executed rs_dsi_check_thread_lock more times than is defined in dsi_commit_check_locks_max, the transaction commits if it is next, or it waits again the amount of time specified in dsi_commit_check_locks_intrvl.

  - If the thread has executed rs_dsi_check_thread_lock more times than is defined in dsi_commit_check_locks_max, the current transaction rolls back.

Function strings for internal commit control

Replication Server uses the rs_dsi_check_thread_lock function to check whether the current DSI executor thread is blocking another replicate database process. This function has function-string-class scope. It is called only if the DSI executor thread is ready to commit but cannot because it is not next to commit, and the amount of time specified for dsi_commit_check_locks_intrvl has elapsed. If commit order contention occurs frequently, consider decreasing the wait time specified by dsi_commit_check_locks_intrvl.

Table 4-7: System functions that support internal commit control

Function	Description
rs_dsi_check_thread_lock	Determines whether or not the DSI executor thread is holding a lock that blocks a replicate database process. A return value greater than 0 indicates that the thread is holding resources required by another database process, and that the thread should roll back and retry the transaction.

Note Replication Server automatically creates function strings for the above function in function-string classes in which Replication Server generates default function strings. For other function-string classes, you must create these function strings before you can use parallel DSI features with dsi_commit_control set on.

Using *rs_threads* to resolve conflicts externally

The rs_threads table is located in the replicate database. It contains a row for each DSI executor thread. To simulate row-level locking, it has two columns, id and seq, and enough dummy columns so that only one row fits on a page. The id column is used as a unique clustered index.

At the beginning of a transaction, the DSI executor thread updates its row in the rs_threads table with the next available sequence number. When it is ready to commit the transaction, the thread sends a select statement to the replicate data server to select, from the rs_threads table, the sequence number of the transaction that should have committed prior to the transaction.

Because the preceding transaction holds a lock on this row in rs_threads, this thread is blocked until the preceding transaction commits.

If the sequence number that is returned is less than the expected value, the thread determines whether it should roll back the transaction or retry the select operation. Because the DSI formats many commands into a single batch before submitting it to the Adaptive Server, a thread may be ready to commit before the preceding transaction has submitted any commands to the Adaptive Server. In this case, the select in the rs_threads table may be submitted several times.

If the sequence number that is returned matches the expected value, the transaction can commit.

Handling deadlocks

If a transaction is ready to commit, but cannot because it is not next in proper commit order, and this transaction is holding locks on resources that are needed by a transaction that must commit before this one, a database resource deadlock occurs at the replicate database. The database resource deadlock consists of the lock on rs_threads held by the next transaction in commit order, and the locks held on resources needed by that transaction. The database resource deadlock is detected by the replicate database, which chooses a transaction to roll back. Since Replication Server must guarantee commit order, when this rollback is forced by the replicate database, Replication Server rolls back all transactions executing against the replicate database and reapplies them serially in commit order.

Function strings for commit control using *rs_threads*

Replication Server manipulates the rs_threads system table with the system functions listed below. These functions have function-string-class scope. They are executed only when more than one DSI thread is defined for a connection.

Table 4-8: System functions that modify the rs_threads system table

Function	Description
rs_initialize_threads	Sets the sequence of each entry in the rs_threads system table to 0. This function is executed during the initialization of a connection.
rs_update_threads	Updates the sequence number for the specified entry in the rs_threads system table.
rs_get_thread_seq	Returns the current sequence number for the specified entry in the rs_threads system table.
rs_get_thread_seq_noholdlock	Returns the current sequence number for the specified entry in the rs_threads system table, using the noholdlock option. This thread is used when dsi_isolation_level is 3.

Note The function strings described in Table 4-8 are needed only when the external, rs_threads method is used for commit control.

Configuring parallel DSI for optimal performance

The following guidelines can help you configure parallel DSI to achieve optimal performance. The objective is to tune parallel DSI processing to provide the best replication performance, balancing parallel processing with acceptable levels of contention. Contentions will always occur. The only way to eliminate contentions is to turn off parallel DSI processing. At the same time, setting all parallel DSI parameters for maximum parallelism may cause Replication Server to spend more time recovering from contentions than actually applying transactions to the replicate. Optimal performance is achieved through a clear understanding of your operating environment so that you can successfully balance parallel processing with acceptable contention levels.

Before you begin

Before you begin tuning for performance:

* *Understand your transaction profile.* What kinds of transactions are being replicated? Do these transactions affect the same rows and tables? Are these transactions liable to conflict if applied in parallel? Is the transaction profile constant, or does it change, perhaps with the time of day or month. A clear understanding of your transaction profile helps you select those parameters and settings that will be most useful.

* *Tune the replicate database to handle contentions.* Most primary databases have been tuned to minimize contentions through the use of clustered indexes, partitioning, row-level locking, and so on. Make sure that your replicate database has been tuned similarly.

* *Define a set of repeatable transactions that accurately reflect your replication environment.* Tuning your parallel DSI environment is an iterative process. You will need to set parameters, run a test, measure performance, compare against previous measurements, and repeat until you have maximized your results.

- *First, reset the dsi_serialization_method parameter.* Set the dsi_serialization_method parameter to no_wait to enable maximum parallelism. Then attempt to reduce contentions by testing other parameters. Because the wait_for_commit (the default) setting supplies minimal parallelism and therefore minimal benefit, only reset dsi_serialization_method to wait_for_commit after all attempts to reduce contention using the no_wait setting have failed to increase performance.

- *Set the dsi_num_threads parameter correctly.* The dsi_num_threads parameter defines the total number of DSI executor threads; the dsi_num_large_xact_threads parameter defines the total number of DSI executor threads reserved for large transactions. Thus, the total number of DSI executor threads (dsi_num_threads) equals the number of DSI threads reserved for large transactions plus the number of threads available for small transactions.

  To begin, try setting dsi_num_threads to 5, and dsi_num_large_xact threads to 2. After selecting a dsi_serialization_method and a dsi_partitioning_rule:

  - Increase dsi_num_threads if contention does not increase, or

  - Decrease dsi_num_threads if contention does not decrease.

  Make sure that dsi_num_threads is greater than the default, and that the value for dsi_num_threads is greater than that for dsi_num_large_xact_threads.

Reducing contention

Start tuning parallel DSI parameters to reduce contention when you have completed the tasks described in "Before you begin" on page 173, and performance tests indicate that contentions are affecting performance. For example:

- The replicate is blocking activity.

- Replication Server is rolling back and reapplying a large percentage of transactions due to deadlock conditions. Refer to counter 5060 – TrueCheckThrdLock.

Start by tuning the dsi_max_xacts_in_group parameter, which determines the number of transactions grouped in a single begin/commit block. By reducing the value of dsi_max_xacts_in_group, you cause the DSI executor threads to commit more frequently. Thus, the DSI executor threads hold fewer replicate resources for shorter periods of time and contentions should decrease.

Adjusting the dsi_num_threads parameter also affects contention. The larger the number of DSI executor threads available, the more likely contentions will arise among the threads. Try decreasing the value of dsi_num_threads even to 3 with one reserved for large transactions. Finding the values that provide best performance is iterative. Remember that some contention is acceptable if overall performance improves.

Using partitioning rules

Partitioning rules can also reduce contention, but require a clear understanding of your transaction profile.

The transaction name rule

Do transactions have transaction names? Is the contention caused by transactions with the same name? Try setting the transaction name rule, which forces transactions with the same name to be sent to the replicate one-by-one.

If transactions are not named, you could change the application so that names are added. Then use the name rule to serialize only specified transactions. Suppose a particular type of large transaction always causes problems if the DSI executor threads attempt to process two or more in parallel. By giving the problem transactions the same name, and applying the name rule, you can ensure that the problem transactions are processed serially. Remember, however, that the name rule is applied to all transactions, and all transactions with the same name will be processed serially.

The user name rule

Setting the user name rule may help reduce contentions caused by transactions processed in parallel from the same user ID. Like the transaction name rule, the user name rule, if set, is applied to all transactions, and every transaction from the same user ID will be processed serially.

The origin begin and commit times rule

The time rule forces serial execution of transactions with nonoverlapping commit/begin times. That is, if the commit time of the first transaction comes before the begin time of the next transaction, these two transactions must execute serially.

Combining partition rules

You can combine rules. The first rule to be satisfied takes precedence. Thus, if, for example, the origin_sessid, time rule is specified, two transactions with the same origin session ID will be forced to run serially, and the time rule is not applied.

Frequent conflicting updates

If your transactions conflict with each other frequently, set the parallel DSI configuration parameters as follows:

*   dsi_serialization_method – set this parameter to wait_for_commit.

*   dsi_num_large_xact_threads – set this parameter to 2. If you are configuring parallel DSI in a warm standby application, set the dsi_num_larg_xact_threads parameter for the standby database to one more than the number of simultaneous large transactions executed at the active database.

*   dsi_num_threads – set this parameter to 3 plus the value of the dsi_num_large_xact_threads parameter. If your transactions are usually small, such as one or two statements, set dsi_num_threads to 1 plus the value of dsi_num_large_xact_threads.

Setting the parallel_dsi configuration parameter on provides a shorthand method for configuring parallel DSI as described above. It also sets the dsi_sqt_max_cache_size parameter to 1 million bytes.

Infrequent conflicting updates

If your transactions conflict with each other only occasionally, set the parallel DSI configuration parameters as follows:

*   dsi_isolation_level – set this parameter to isolation level 3 if your replicate data server is Adaptive Server. For non-Sybase data servers, set to the level that corresponds to ANSI standard level 3.

*   dsi_num_large_xact_threads – set this parameter to 2. If you are configuring parallel DSI in a warm standby application, set the dsi_num_larg_xact_threads parameter for the standby database to one more than the number of simultaneous large transactions executed at the active database.

*   dsi_num_threads – set this parameter to 3 plus the value of the dsi_num_large_xact_threads parameter.

Using isolation levels

Use DSI isolation levels to prevent loss of parts of transactions when parallel DSI is enabled, and the replicate table is configured for row-level locking. In these cases, the order of individual operations within transactions may not match that seen at the primary, even if the transactions themselves are committed in proper order.

For example, if the second transaction to commit updates a row inserted by the first transaction to commit, the update may take place before the commit. In this case, the transactions commit correctly, but the update is lost, even though the insert remains.

To avoid out-of-sequence DML operations, set dsi_isolation_level to 3. In the example, if dsi_isolation_level is 3, the second transaction to commit acquires a range lock on the as-yet nonexistent row it intends to update, which causes a deadlock with the first transaction to commit. The data server declares a database resource deadlock. Replication Server rolls back all open transactions and serially reapplies them, and the update is not lost.

Setting the size for large transactions

Setting dsi_large_xact_size to a large number, even the maximum (2147483647), to remove the overhead of handling large transactions may give better performance than allowing large transactions to start before their commit point is read.

Parallel DSI and the *rs_origin_commit_time* system variable

The value of the *rs_origin_commit_time* system variable depends on whether you are using the parallel DSI feature.

- If you are not using parallel DSI to process large transactions, the value of *rs_origin_commit_time* contains the time when the last transaction in the transaction group committed at the primary site.

- If you are using parallel DSI to process large transactions (before their commit has been read from the DSI queue), when the DSI threads start processing one of these transactions, the value of *rs_origin_commit_time* is set to the value of *rs_origin_begin_time*.

When the commit statement for the transaction is read, the value of *rs_origin_commit_time* is set to the actual commit time. Therefore, when the configuration parameter dsi_num_large_xact_threads is set to a value greater than zero, the value for *rs_origin_commit_time* is not reliable for any system function other than rs_commit.

Dynamic SQL for enhanced Replication Server performance

Dynamic SQL in Replication Server enhances replication performance by allowing Replication Server Data Server Interface (DSI) to prepare dynamic SQL statements at the target user database and to execute them repeatedly. Instead of sending SQL language commands to the target database, only the literals are sent on each execution, thereby eliminating the overheads brought by SQL statement syntax checks and optimized query plan builds.

If turned on, dynamic SQL will be used in a user database connection instead of a language command if:

- The command is insert, update, or delete.

- There are no text, image, java or opaque columns in the command.

- There are no NULL values in the where clause for update or delete command.

- There are no more than 255 parameters in the command:

    - insert commands can have no more than 255 columns.

    - update commands can have no more than 255 columns in the set clause and where clauses combined.

    - delete commands can have no more than 255 columns in the where clause.

- The command does not use user-defined function strings.

Setting up the configuration parameters to use dynamic SQL

Dynamic SQL is off by default at a server and connection level. Configure dynamic SQL at a server or a connection level by issuing these commands:

```
configure replication server
set { dynamic_sql |
      dynamic_sql_cache_size |
      dynamic_sql_cache_management }
```

```
to value

alter connection to server.db
set { dynamic_sql |
      dynamic_sql_cache_size |
      dynamic_sql_cache_management }
to value
```

Table-level dynamic
SQL control

create/alter replication definition commands allow you to control the application of dynamic SQL on each table through replication definition.

See the *Replication Server Reference Manual* for information about create/alter replication definition commands.

You can change the dynamic SQL execution at the table level for a specific replicate database by using:

```
set dynamic_sql {on | off}
for replication definition with replicate at
data_server.database
```

At the replication definition level, the default is to use dynamic SQL. You only need to use these commands to change the dynamic SQL usage if you want to exclude tables from dynamic SQL. To check for dynamic SQL usage, turn on stats_sampling and run admin stats, dsi command and look for DSIEDsqlPrepared, DSIEDsqlExecuted and other dynamic SQL related counters.

Use stored procedures rs_helprep, rs_helpsub, and rs_helppubsub to display dynamic SQL setting for each replication definition.

See "rs_helprep", "rs_helpsub", and "rs_helppubsub," in Chapter 6, "Adaptive Server Stored Procedures" in the *Replication Server Reference Manual* for information about using these stored procedures.

The server-level configurations provide the default values for the connections created or started in the future. For database-level configurations:

- dynamic_sql – turns dynamic SQL on or off for a replicate connection. Other dynamic SQL related configuration parameters take effect only if this parameter is set to on.

- dynamic_sql_cache_size – tells the Replication Server how many database objects may use the dynamic SQL for a connection. This parameter is provided to limit the resource demand on the data server.

- dynamic_sql_cache_management – manages the dynamic SQL cache for a connection. Once the dynamic SQL statements reaches dynamic_sql_cache_size for a connection, it either stops allocating new dynamic SQL statements if the value is fixed, or it keeps the most recently used statements and deallocates the rest to allocate new statements if the value is mru.

Limitations

Dynamic SQL has these limitations:

- If a table is replicated to a standby or MSA connection using an internal replication definition, and dynamic SQL is enabled for the connection, any new replication definition for the table should define the column order consistent with the column order in the primary database. Otherwise, the existing prepared statements may be invalidated, and may require the standby or MSA connection to be restarted.

- Replication Server converts user-defined datatypes to Open Client/Server™ (OCS) datatype in a dynamic SQL command.

  If data falls outside Sybase ranges that cause dynamic SQL to fail, DSI logs an error message and resends dynamic SQL using the language command. DSI shuts down only if the language command also fails.

  If this condition happens frequently, disable dynamic SQL from the table replication definition or use the set dynamic_sql off command.

  Use any of these commands to turn off dynamic_sql:

  - alter connection... set dynamic_sql off – turns dynamic SQL off for all commands in this connection.

  - create/alter replication definition...without dynamic_sql – turns dynamic SQL off for all commands using this replication definition.

  - set dynamic_sql off for replication definition with replicate at... – turns dynamic SQL off for all commands using this replication definition at this replicate connection.

Using multiprocessor platforms

You can run Replication Server on symmetric multiprocessor (SMP) or single-processor platforms. Replication Server multithreaded architecture supports both hardware configurations. On a single processor platform, Replication Server threads run serially. On a multiprocessor platform, Replication Server threads can run in parallel, thereby improving performance and efficiency.

Note The SMP option in Linux Edition is not available.

Replication Server is an Open Server application. Replication Server support for multiple processors is based on Open Server support for multiple processors. Both servers use the POSIX thread library on UNIX platforms and the WIN32 thread library on Windows platforms. For detailed information about Open Server support for multiple processing machines, see the *Open Server Server-Library/C Reference Manual*.

When Replication Server is in single-processor mode, a server-wide mutual exclusion lock (mutex) enforces serial thread execution. Serial thread execution safeguards global data, server code, and system routines, ensuring that they remain thread-safe.

When Replication Server is in multiprocessor mode, the server-wide mutex is disengaged and individual threads use a combination of thread management techniques to ensure that global data, server code, and system routines remain secure.

Enabling multiprocessor support

To specify whether Replication Server takes advantage of a multiprocessor machine, use configure replication server with the smp_enable option. For example:

```
configure replication server set smp_enable to 'on'
```

Setting smp_enable "on" specifies multiprocessor support; setting smp_enable "off" specifies single-processor support. The default is "off."

smp_enable is a static option. You must restart Replication Server after changing the status of smp_enable.

Monitoring thread status

You can verify Replication Server thread status using these commands:

- admin who – provides information on all Replication Server threads

- admin who_is_up or admin who_is_down – lists Replication Server threads that are running, or not running.

- sp_help_rep_agent – provides information on the RepAgent thread and the RepAgent User thread.

See Chapter 1, "Verifying and Monitoring Replication Server" for more information about monitoring thread status.

Monitoring performance

Replication Server provides monitors and counters for monitoring performance. See "Using Counters to Monitor Performance" on page 187.

Allocating queue segments

You can choose the disk partition to which Replication Server allocates segments for stable queues. By choosing the stable queue placement, you can enhance load balancing and read/write distribution.

Replication Server stores messages destined for other sites on partitions. It allocates space in partitions to stable queues and operates in 1MB chunks called segments. Each stable queue holds messages to be delivered to another Replication Server or to a data server. The queues hold data until it is sent to its destination.

rs_init assigns Replication Server initial partition. You may need additional partitions, depending on the number of databases and remote Replication Servers to which the Replication Server distributes messages.

A Replication Server can have any number of partitions of varying sizes. The sum of the partition sizes is the Replication Server capacity for queued transactions.

By default, Replication Server assigns queue segments to the first partition in an ordered list of partitions. See Figure 4-9. When the first partition becomes full, the first partition becomes the last partition, and the next queue segment is allocated to the new first partition. When the default method is used, the rolling allocation of segments is automatic and cannot be controlled by the user.

Figure 4-9: Default allocation mechanism
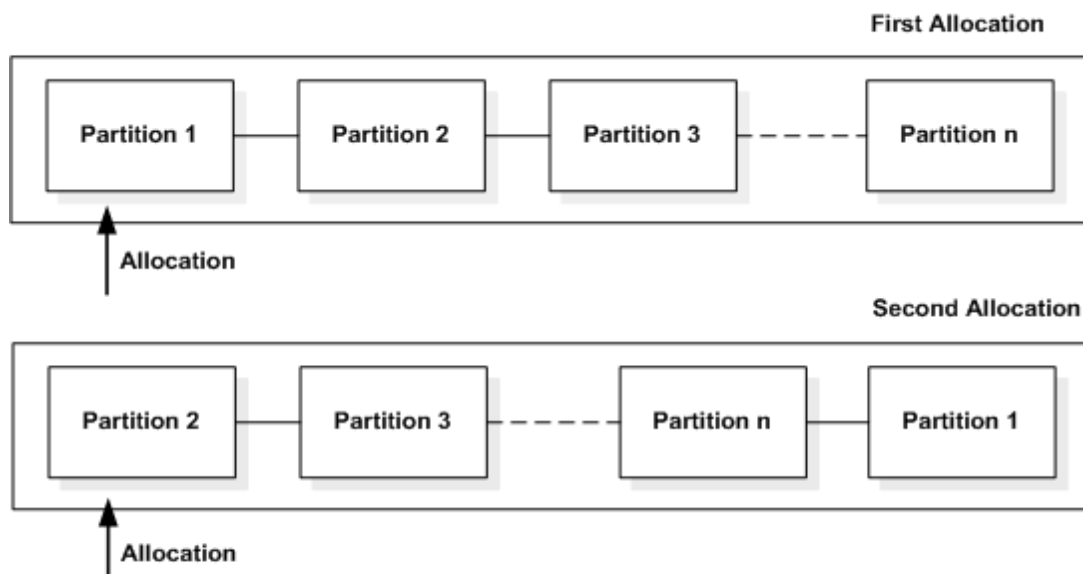
Choosing disk allocations

To choose the segment allocation, use the alter connection or alter route command with the set disk_affinity option. The syntax is:

```
alter connection to dataserver.database
    set disk_affinity to [ 'partition' | 'off' ]
```

```
alter route to replication_server
    set disk_affinity to [ 'partition' | 'off' ]
```

partition is the logical name of the partition to which you want to allocate the next segment for the connection or route.

Each allocation directive is called a "hint" because Replication Server can override the allocation if, for example, the allocated partition is full or has been dropped. If Replication Server overrides the hint, it allocates segments according to the default mechanism described in Figure 4-9.

Replication Server checks for an allocation hint each time it allocates a new segment for a queue. Each hint is stored in the rs_diskaffinity system table. Each partition may have many hints, but each stable queue can have only one hint.

Successfully using disk allocation to improve performance depends on the architecture and other characteristics of your site. One way to improve overall throughput is to associate faster devices with those stable queues that process more slowly.

In addition, if new partitions are added after all connections are in place, the new partitions are not used until the existing ones are filled. You can force a connection to use the new partition by adding allocation hints.

An example

You can allocate different disk partitions to different stable queues. You could, for example, make partitions of different sizes available to different database connections. In this example, we add partitions of 10MB and 20MB to the Replication Server and specify allocation hints for the TOKYO_DS and SEATTLE_DS data servers. The procedure is:

1   Make the partitions P1 and P2 on the device named /dev/rds0a available to Replication Server, enter:

```
create partition P1 on '/dev/rds0a' with size 20
create partition P2 on '/dev/rds0a' with size 10
```

2   Suspend the connection to the TOKYO_DS and SEATTLE_DS data servers, enter:

```
suspend connection to TOKYO_DS
suspend connection to SEATTLE_DS
```

3   Specify allocation hints for the connection to the TOKYO_DS and SEATTLE_DS data servers, enter:

```
alter connection to TOKYO_DS.db1
set disk_affinity to 'P1'

alter connection to SEATTLE_DS.db5
set disk_affinity to 'P2'
```

4   Resume the connections to the TOKYO_DS and SEATTLE_DS data servers, enter:

```
resume connection to TOKYO_DS
resume connection to SEATTLE_DS
```

Preallocating segments

Replication Server 15.1 preallocates segments in the background to reduce segment allocation latency. Segment allocation imposes significant latency to writer threads especially when the RSSD is on a remote Adaptive Server.

When a writer thread needs a new segment, it checks whether the preallocated segment is available, if it is not, the thread requests to allocate the segment. Once the writer thread gets the new segment, a preallocation request is made so that the segment is allocated in the background. By the time the writer thread needs a new segment, it is already available.

Enable or disable segment preallocation using:

```
configure replication server set sqm_seg_prealloc to
"on|off"
```

This command is static, which means you must restart the server for it to take effect. It supports only the server-level configuration.

Dropping hints and partitions

You can remove an allocation hint using the alter connection or alter route command with the set disk_affinity to 'off' parameter. For example:

```
alter connection to TOKYO_DS.db1
set disk_affinity to 'P1' to 'off'
```

This command deletes the allocation hint for P1 from the rs_diskaffinity table.

You can remove a partition from Replication Server using the drop partition command. If the partition you are dropping has one or more allocation hints in the rs_diskaffinity table, Replication Server marks the allocation hints for deletion, but does not delete them until all data stored on the partition has been successfully delivered and the partition has been dropped.

Using the heartbeat feature in RMS

To view latency information, use the heartbeat feature in the command line service, Replication Monitoring Services (RMS). The heartbeat feature uses the stored procedure rs_ticket to generate latency information, which is the amount of time it takes a transaction to move from the primary to the replicate database. At a specified interval, the RMS executes rs_ticket at a primary database. The latency information that has been generated is stored in a table in the replicate database.

RMS provides commands to set up the heartbeat process and to retrieve that latency information from the replicate database. The heartbeat feature is available only through RMS. See the *Replication Server Reference Manual* for more information about the heartbeat commands.

Using Counters to Monitor Performance

This chapter describes how to use Replication Server counters to monitor performance. To monitor performance using the RepAgent counters, see "Using counters to monitor RepAgent performance" on page 128 in the *Replication Server Administration Guide Volume 1*.

Introduction

Replication Server has several hundred different counters that can monitor performance at various points and areas in the replication process. By default, counters are not active until you choose to activate them—with the exception of a few counters that are always active.

You can view current counter values and other performance information at any time using these commands:

- admin stats – displays current values for specified counters.

- admin stats, backlog – displays the current backlog in the Replication Server stable queues.

- admin stats, { tps | cps | bps } – displays throughput in terms of transactions per second, commands per second, or bytes per second.

- admin stats, { md | mem | mem_in_use } – displays message and memory information

Counter values can also be saved (or flushed) to the RSSD, where averages and rates can be calculated and viewed using standard Transact-SQL statements or the rs_dump_stats stored procedure.

Modules and counters: an overview

In Replication Server, a module is a group of components that work together to perform specific services. For example, the Stable Queue Manager (SQM) consists of logically related components that provide stable queue services. Replication Server provides counters that can track activity at each instance (occurrence) of each module.

Some modules have exactly one instance in Replication Server. Instances of those modules can be identified by the module name alone. Examples of this type of module are:

- System Table Services (STS)

- Connection Manager (CM)

Other modules can have multiple instances in Replication Server. To uniquely identify each instance of the module, you must include both the module name and the instance ID. Examples include:

- Replication Server Interface (RSI)

- Distributor (DIST)

- Data Server Interface, scheduler thread (DSI/S)

Still other modules require three identifiers to differentiate them: the module name, the instance ID, and an instance value. Examples include:

- Stable Queue Transaction thread (SQT)

- Stable Queue Manager (SQM)

- Data Server Interface, executor thread (DSIEXEC)

Table 5-1 lists the most commonly used modules. Counters for independent modules can be addressed directly using Replication Server commands. To access counters for dependent modules, use the name of their parent modules.

Table 5-1: Replication Server modules

Module name	Acronym	Independent/dependent
Connection Manager	CM	Independent
Distributor	DIST	Independent
Data Server Interface	DSI	Independent
DSI Executor	DSIEXEC	Dependent of DSI
RepAgent thread	REPAGENT	Independent
Replication Server Interface	RSI	Independent
RSI User	RSIUSER	Independent
Replication Server Global	SERV	Independent
Stable Queue Manager	SQM	Independent
SQM Reader	SQMR	Dependent of SQM
SQM Transaction Manager	SQT	Independent
System Table Services	STS	Independent
Thread Synchronization	SYNC	Independent
SYNC Element	SYNCELE	Dependent of SYNC

Counters

To view descriptive and status information about Replication Server counters, use the rs_helpcounter stored procedure. See "Viewing information about the counters" on page 199.

Each counter has a descriptive name and a display name that you use to identify the counter when you enter RCL commands and when you view displayed information.

Different kinds of counters provide different types of information. Although not all counters can be divided into discrete categories, when Replication Server displays counter information it uses these categories:

- Observers – collect the number of occurrences of an event over a time period. For example, observers might collect the number of times a message is read from a queue. Replication Server reports the number of occurrences and the number of occurrences per second.

- Monitors – collect measurements at a given time or times. For example, monitors might collect the number of operations per transaction. Replication Server reports the number of observations, the last value collected, the maximum value, and the average value.

- Counters – collect a variety of measurements. Counters that measure duration are in this group as are counters that collect total numbers of bytes. For this category, Replication Server can report number of observations, total value, last value, maximum value, an average, and rate per second.

Sampling

You have several options for gathering data. You can choose whether to sample data over a long period of time, a short period of time (seconds), or a single occurrence.

You can collect counter statistics in either of two ways:

- By executing admin stats with the display option, which instructs Replication Server to collect information for a specified time period and then, at the end of that time period, to display the information collected on the computer screen.

- By executing admin stats with the save option, which instructs Replication Server to collect information for a specified number of observations within a specified time period, and save that information to the RSSD.

By default, information is not collected from the counters until you turn them on. You can turn them on for a specific time period when you execute admin stats. You can also turn on sampling for an indefinite time period by setting the stats_sampling configuration parameter on.

Turning on sample collection activates all counters. However, you can display or save statistics only for those counters or modules that are of interest.

Statistics shown on the computer screen record the number of events and computed values—such as averages and rates—for a single observation period. When statistics are sent to the RSSD, Replication Server saves raw values—such as observations, totals, last value, and maximum value—for multiple consecutive observation periods. You can then compute averages and rates from these stored values.

Collecting statistics for a specific time period

The syntax for admin stats is:

admin { stats | statistics } [, sysmon | "all"
       | *module_name* [, inbound | outbound] [, *display_name*]]
       [, *server*[, *database*] | *instance_id*]
       [, display |, save [, *obs_interval*]]
       [, *sample_period*]

admin stats lets you specify:

* The counters to be sampled

* The length of the observation interval and the sample period

* Whether to save statistics to the RSSD or display them on the computer
  screen

Note admin stats also supports the cancel option. This stops the currently
running command.

By default, Replication Server does not report counters that show 0 (zero)
observations for the sample period. You can change that behavior by setting the
stats_show_zero_counter configuration on using configure replication server.
See the *Replication Server Reference Manual* for complete syntax and usage
information.

Specifying the counters to be sampled

You can specify all counters or as few as a single instance of a counter.

* sysmon – samples all counters marked by Sybase as most important to
  performance and tuning. This is the default value.

  To view a list of the sysmon counters, enter:

      rs_helpcounter sysmon

* "all" – samples all counters.

* module_name – samples all counters for a particular module. See
  "Modules and counters: an overview" on page 188 for a list of modules.

* module_name, display_name – samples all instances of a particular
  counter. Use sp_helpcounter for a list of counters.

- module_name, display_name, instance_id – samples a particular instance of a counter. To find the numeric ID for an instance, execute admin_who and see the Info column.

> **Note** If the instance ID is specified and the module is either SQT or SQM, you can specify whether you want information supplied by the inbound or outbound queue for the counter instance.

For example, to collect statistics for the sysmon counters for one second and send the information to the computer screen, enter:

```
admin stats, sysmon, display, 1
```

Specifying the sample period

You specify a sampling period in numbers of seconds. Replication Server collects statistics for the named counters for that number of seconds and reports to the screen or the RSSD. The default value is 0 (zero) seconds—which causes all counters to report their current value.

For example, to collect statistics for all counters for one minute and display them on the computer screen, enter:

```
admin stats, "all", display, 60
```

Specifying how statistics are to be reported

Statistics can be sent to the computer screen or to the RSSD.

Displaying statistics on the computer screen

To send statistics to the computer screen, include the display option. In this case, Replication Server makes a single observation at the end of the specified time period. The observed statistics are sent only to the computer screen.

For example, to report the number of blocks read from all queues and by all readers over a five-minute interval, enter:

```
admin stats, sqm, blocksread, display, 300
```

When you execute admin stats with a nonzero time period using the display option, Replication Server:

1 Resets all counters to zero.

2 Turns on all counters.

3   Puts your session to sleep for the specified time period.

4   Turns off all counters.

5   Reports the requested data.

Saving statistics in the RSSD

To save statistics in the RSSD, include the save option, which immediately returns the session.

When you send statistics to the RSSD, you can specify the length for each observation interval (*obs_interval*) during the specified sampling period. *obs_interval* can be a numeric value in seconds, or a quoted time format string hh:mm[:ss].

For example, to start sampling and saving statistics to the RSSD for one hour and thirty minutes at 20-second intervals, enter:

```
admin stats, "all", save, 20, "01:30:00"
```

To collect statistics for the outbound SQT for connection 108 for two minutes at 30-second intervals, enter:

```
admin stats, sqt, outbound, 108, save, 30, 120
```

Replication Server determines the number of observation intervals by dividing the sampling period by the observation interval. The remainder in seconds, if any, is added to the last observation interval.

Sampling period (*sample_period*)	Observation interval (*obs_interval*)	Number of observation intervals
60 seconds	15	Four 15-second intervals
75 seconds	5	Not allowed – observation interval must be => 15 seconds
60 seconds	30	Two 30-second intervals
130 seconds	20	Five 20-second intervals and a final 30-second interval
10 seconds	Not specified	One 10-second interval

When you execute admin stats with a nonzero time period using the save option, Replication Server starts a background thread to collect sampling data and returns your session immediately. Once the session is returned, you can use admin stats, status command to check the sampling progress. The background thread:

1   Truncates the rs_statrun and rs_statdetail system tables if the configuration
     parameter stats_reset_rssd is set to on.

2   Resets all counters.

3   Turns on all counters.

4   Writes the requested counters to the RSSD at the end of each observation
     period.

5   Turns off all counters.

Note To keep old sampling data, set the configuration parameter
stats_reset_rssd to off or make sure that you have dumped any needed
information from rs_statrun and rs_statdetail before executing admin stats with
the save option. See "Using the rs_dump_stats procedure" on page 198 for
information about dumping information from these tables.

Collecting statistics for an indefinite time period

To turn on sampling for an indefinite period, configure Replication Server
using the stats_sampling parameter. Enter:

```
configure replication server
  set stats_sampling to "on"
```

Replication Server continues to collect data until you reconfigure Replication
Server to turn sampling off.

```
configure replication server
   set stats_sampling to "off"
```

Then, when you want to view data on the computer screen or send the collected
data to the RSSD, use admin stats.

Note Use admin stats with care when stats_sampling is on. If you execute
admin stats and specify a nonzero time period, Replication Server clears the
counters, executes the command, and turns stats_sampling off.

For example, to collect statistics for two consecutive 24-hour periods,
reporting results to the computer screen, you might follow this sequence:

Day 1, 8am

1   Clear existing statistics, enter:

```
admin statistics, reset
```

2   Turn on sampling:

```
configure replication server
set stats sampling to "on"
```

Day 2, 8am

1   Turn off sampling to ensure Replication Server does not collect statistics
    as statistics are dumped to the screen.

```
configure replication server
set stats sampling to "off"
```

2   Dump statistics to the screen:

```
admin statistics, "all"
```

3   Clear all statistics:

```
admin statistics, reset
```

4   Turn on sampling:

```
configure replication server
set stats_sampling to "on"
```

Day 3, 8am

1   Turn off sampling to ensure Replication Server does not collect statistics
    as statistics are dumped to the screen.

```
configure replication server
set stats sampling to "off"
```

2   Dump statistics to the screen:

```
admin statistics, "all"
```

3   Clear all statistics:

```
admin statistics, reset
```

Viewing statistics on screen

admin stats displays statistics on the computer screen from a single sample run.
You can display statistics for a single counter instance, a single counter, all
counters for a particular module, the generally most useful or "sysmon"
counters, or all counters.

You choose whether to display statistics on the screen when you configure the sample run using admin stats—see "Collecting statistics for a specific time period" on page 190.

See the *Replication Server Reference Manual* for example output and complete syntax and usage information.

Viewing throughput rates

Use admin stats with the tps, cps, or bps option to view the current throughput in terms of transactions, commands, or bytes per second.

Transactions per second

Replication Server calculates the transaction rate based on the number of processed transactions and the number of elapsed seconds since the counters were last reset. The data is obtained from several modules, including the SQT, DIST, and DSI modules.

To view throughput in transactions per second, enter:

```
admin stats, tps
```

Commands per second

The number of commands per second is calculated from the number of commands processed and the number of elapsed seconds since the last reset. The data is obtained from the REPAGENT, RSIUSER, RSI, SQM, DIST, and DSI modules.

To view throughput in commands per second, enter:

```
admin stats, cps
```

Bytes per second

The number of bytes per second is calculated from the number of bytes processed and the number of elapsed seconds since the last reset. The data is obtained from the REPAGENT, RSIUSER, SQM, DSI, and RSI modules.

To view throughput in bytes per second, enter:

```
admin stats, bps
```

Viewing statistics about messages and memory use

Use admin stats with the md option to view information about the number of messages. Use admin stats with the mem, or mem_in_use options to view information about memory use.

- To view statistics for message delivery, which is associated with Distributors and RSI users, enter:

```
admin stats, md
```

- To view current segment usage according to segment size, enter:

    ```
    admin stats, mem
    ```

- To view current memory use in bytes, enter:

    ```
    admin stats, mem_in_use
    ```

Viewing the number of transactions in the stable queues

You can view the number of transactions in both the inbound and outbound stable queues awaiting distribution. Replication Server reports the data in terms of segments and blocks, where one segment is equal to 1MB, and one block is equal to 16K. The data is obtained from the SQMRBacklogSeg and the SQMRBacklogBlock counters.

To view the stable queue backlog, enter:

```
admin stats, backlog
```

Viewing statistics saved in the RSSD

Statistics sent to the RSSD are stored in these system tables:

- rs_statcounters – contains descriptive information for each counter

- rs_statdetail – contains observed metrics for each sampling run for each counter

- rs_statrun – describes each sampling run

See the *Replication Server Reference Manual* for detailed information about these tables.

You can view statistics stored in these tables using:

- select and other Transact-SQL commands

- rs_dump_stats

- rs_helpcounter to display information from rs_statcounters

Using the *rs_dump_stats* procedure

rs_dump_stats dumps the contents of the rs_statrun and rs_statdetail system tables to a CSV file that can be loaded into a spreadsheet for analysis. For complete syntax and usage information for rs_dump_stats, see the *Replication Server Reference Manual*.

To use rs_dump_stats, log in to the RSSD and execute the stored procedure. For example:

```
1> rs_dump_stats
2> go
```

Note Comments to the right of the output are included to explain the example. They are not part of the rs_dump_stats output.

```
Comment:  Sample of rs_dump_stats output
Nov  5 2005 12:29:18:930AM                      *Start time stamp*
Nov  5 2005 12:46:51:350AM                      *End time stamp*
16                                              *No of obs intervals*
1                                               *No of min between obs*
16384                                           *SQM bytes per block*
64                                              *SQM blocks per segment*
CM                                              *Module name*
13                                              *Instance ID*
-1                                              *Instance value*
dCM                                             *Module name*
CM: Outbound database connection request        *Counter external name*
CMOBDBReq                                       *Counter display name*
13003           , , 13,  -1                     *Counter ID, instance ID,
                                                    instance value*

ENDOFDATA                                       *EOD for counter*

CM: Outbound non-database connection requests   *Counter external name*
CMOBNonDBReq                                     *Counter display name*
13004           , , 13,  -1                     *Counter ID, instance ID,
                                                    instance value*
Nov  5 2005 12:29:18:930AM,  103,  103,  1,  1  *Dump ts, obs, total,
                                                    last, max*
Nov  5 2005 12:30:28:746AM,  103,  103,  1,  1
Nov  5 2005 12:31:38:816AM,  107,  107,  1,  1
Nov  5 2005 12:32:49:416AM,  104,  104,  1,  1
Nov  5 2005 12:33:58:766AM,  114,  114,  1,  1
...
Nov  5 2005 12:46:51:350AM,  107,  107,  1,  1
```

```
ENDOFDATA                                        *EOD for counter*

CM: Outbound 'free' matching connections found   *Counter external name*
CMOBFreeMtchFound                                *Counter display name*
13005           , , 13,  -1                      *Counter ID, instance ID,
                                                     instance value*

Nov  5 2005 12:29:18:930AM,  103,  103,  1,  1   *Dump ts, obs, total,
                                                     last, max*

Nov  5 2005 12:30:28:746AM,  103,  103,  1,  1
...
Nov  5 2005 12:46:51:350AM,  2,  2,  1,  1
ENDOFDATA                                        *EOD for counter*
```

Viewing information about the counters

You can view descriptive information about the counters stored in the rs_statcounters table using the rs_helpcounter system procedure. See "rs_helpcounter" in the *Replication Server Reference Manual* for detailed syntax and usage information.

• To view a list of modules that have counters and the syntax of the rs_helpcounter procedure, enter:

        rs_helpcounter

• To view descriptive information about all counters for a specified module, enter:

        rs_helpcounter *module_name*[, short | long]

    If you enter short, Replication Server prints the display name, module name, and counter descriptions for each counter.

    If you enter long, Replication Server prints every column in rs_statcounters for each counter.

    If you do not enter a second parameter, Replication Server prints the display name, the module name, and the external name of each counter.

• To list all counters that match a keyword, enter:

        rs_helpcounter *keyword* [, short |, long]

• To list counters with a specified status, the syntax is:

```
rs_helpcounter { sysmon | internal | must_sample
         | no_reset | old | configure }
```

Resetting counters

You can reset all counters, except those that are never reset, to 0 (zero) by issuing the admin stats, reset command:

```
admin stats, reset
```

If sampling has not been enabled using the stats_sampling parameter, counter values are zero. Running admin stats with a nonzero sample period sets the counters to zero, turns on sampling, turns off counter sampling after the sampling run is completed, and resets the counters to zero. If the sampling period is zero, current counter values are reported.

If sampling has been enabled, use admin stats with care. With sampling enabled using the stats_sampling configuration, counter values are accumulating. Issuing admin stats and specifying a sample period causes Replication Server to clear all counters and disable sampling (stats_sampling off) after the sampling run.

Generating performance reports

You must load this script into the RSSD after upgrading to Replication Server 15.1:

$SYBASE/$SYBASE_REP/scripts/rs_install_statreport_v1510_[ase|asa].sql

After loading the script, run rs_stat_populate and rs_stat_genreport stored procedures to generate these performance reports:

• Replication Server performance overview – overview information about your Replication Servers, such as DIST processing, DSI processing, and so on.

• Replication Server performance analysis – performance analysis and tuning suggestions based on critical Replication Server counters. The detailed description is available in the script file.

- Active object identification result – lists the active table and procedure names, owner names, execution times, and so on.

For more information about rs_stat_populate and rs_stat_genreport, see the script file, which contains syntax, examples, and so on.

CHAPTER 6   **Handling Errors and Exceptions**

This chapter describes various error handing methods for Replication Server.

Topic	Page
General error handling	203
Error log files	204
Data server error handling	208
Exceptions handling	213
DSI duplicate detection	219
Duplicate detection for system transactions	220

Refer to the *Replication Server Troubleshooting Guide* for information about resolving specific errors.

General error handling

Replication Server passes messages to data servers and other Replication Servers while they are accessible and queues messages when connections are down. Using Sybase Central, you can monitor the status of the replication system and troubleshoot problems as they arise.

Normally, short-term failures of networks and data servers do not require special error handling or intervention. When the failure is corrected, replication system components resume their work automatically. Lengthier failures may require intervention if there is not enough disk space to queue up messages or if it is necessary to reconfigure the replication system to work around the failure.

Failures of some system components, such as Replication Server partitions or primary databases, also require user intervention. Refer to Chapter 7, "Replication System Recovery" for more information about recovery procedures.

A Replication Server response to errors depends on the kind of error, source of the error, and how the Replication Server is configured. Replication Server handles errors in these ways:

- Logs errors in its error log file.

- Responds to data server errors based on configuration settings.

- If transactions fail to commit in a database, writes the transactions to the exceptions log for manual resolution.

- Detects duplicate transactions after system restart.

Error log files

This section describes error log files in the replication system. You can access log files to help you troubleshoot Replication Server and RepAgent. To view skipped transactions that are written to system tables, you can access the Adaptive Server for the Replication Server managing a specified database. Refer to the *Replication Server Troubleshooting Guide* for details on troubleshooting errors.

Replication Server allows user-definable error processing in response to data server errors. For details, see "Data server error handling" on page 208.

Replication Server error log

The Replication Server error log is a text file where Replication Server writes informational and error messages.

By default, the Replication Server error log file name is *repserver.log*, and resides in the directory where you started the Replication Server. You can specify the name and location of the error log file by using the -E command line flag when you start the Replication Server or in a Replication Server run file

Each log message begins with a letter to indicate the message type. Table 6-1 lists the possible message types.

Table 6-1: Message types in the Replication Server error log

Error code	Description
I	An informational message.

Error code	Description
W	A warning about a condition that has not yet caused an error, but may require attention. An example is running out of a resource.
E	An error that does not prevent further processing, such as a site that is unavailable.
H	A Replication Server thread has died. An example is a lost network connection.
F	Fatal. A serious error caused Replication Server to exit. An example is starting the Replication Server with an incorrect configuration.
N	Internal error. These errors are caused by anomalies in the Replication Server software. Report these errors to Sybase Technical Support.

Informational messages

For informational messages in the error log, the format is as follows:

```
I. date: message
```

The letter "I" at the beginning of a message means that the message is provided for information. It does not mean that an error occurred. For example, Replication Server outputs the following messages as it drops a subscription:

```
I. 95/11/01 05:41:54. REPLICATE RS: Dropping
subscription authors_sub for replication definition
authors with replicate at <SYDNEY_DS.pubs2>
I. 95/11/01 05:42:02. SQM starting: 104:-2147483527
authors.authors_sub
I. 95/11/01 05:42:12. SQM Stopping: 104:-2147483527
authors.authors_sub
I. 95/11/01 05:42:20. REPLICATE RS: Dropped
subscription authors_sub for replication definition
authors with replicate at <SYDNEY_DS.pubs2>
```

Error and warning messages

For messages other than informational messages, the format is as follows:

```
severity, date. ERROR #error_number
thread_name(context) - source_file(line) message
```

If the message is a warning, "ERROR" in the above format becomes "WARNING."

The *severity* is either W, E, H, F, or N, as listed in Table 6-1. The *date* is the date and time that the error occurred. The *error_number* is the Replication Server error number.

The *thread_name* is the name of the Replication Server thread that received the error. See Chapter 2, "Replication Server Technical Overview" in the *Replication Server Administration Guide Volume 1* and Chapter 4, "Performance Tuning" for details about Replication Server threads. The *context* provides some information about the thread's context at the time the error occurred.

The *source_file* and *line* point to the program file and line number in the Replication Server source code where the error was reported.

The *message* is the full text of a message from a Replication Server. It is in the language specified in the *RS_language* configuration parameter. Some messages also include a message from a data server, or one of the component libraries that Replication Server uses.

Note Replication Server puts question marks (?) in messages where more specific information is not available. For example, if an error occurs during initialization, Replication Server may not yet have completed some internal structures, so it prints question marks in place of information it has not yet collected.

The following example shows the Replication Server error log entry for a data server:

```
E. 95/11/01 05:30:52. ERROR #1028 DSI(SYDNEY_DS.pubs2)
- dsiqmint.c(3522)Message from server:
 Message: 2812, State: 4, Severity: 16 --
 'Stored procedure 'upd_authors' not found.
H. 95/11/01 05:30:53. THREAD FATAL ERROR #5049
DSI(SYDNEY_DS.pubs2) - dsiqmint.c(3529)
The DSI thread for database 'SYDNEY_DS.pubs2' is being
shutdown because of error action mapped from data server
error '2812'. The error was caused by output command '1'
mapped from source command '2' of the transaction.
```

The messages indicate that Adaptive Server returned error number 2812, causing Replication Server to take the stop_replication action. See "Assigning actions to data server errors" on page 212.

Finding the name of the Replication Server error log

Use the admin log_name command to find the name of the current Replication Server error log file. Replication Server displays the path to the log file, as the following UNIX example shows:

```
Log File Name
-----------------------------
 /work/sybase/SYDNEY_RS/SYDNEY_RS.log
```

Changing to a new Replication Server log file

To begin a new error log file, use the admin set_log_name command. This command closes the current log file and opens a new one. Subsequent messages are written in the new log file.

Following is an example of the admin set_log_name command for UNIX:

```
admin set_log_name,
'/work/sybase/SYDNEY_RS/951101.log'
```

The previous log remains active if Replication Server fails to create and open the new log file.

RepAgent error log messages

All RepAgent error, trace, and information messages are logged in the Adaptive Server error log file. Each message identifies the RepAgent that logged the error in the string "RepAgent (*dbid*)", which appears in the first line of the message. *dbid* is the database identification number of the RepAgent that logged the error.

Here is an example of an information message:

```
RepAgent(dbid): Recovery of transaction log is
complete. Please load the next transaction log dump and
then start up the Rep Agent Thread with
sp_start_rep_agent, with 'recovery' specified.
```

The Adaptive Server error log is a text file. The messages are printed in the language specified at Adaptive Server. RepAgent records errors and informational messages that occur when transferring replicated objects from the Adaptive Server transaction log and converting them into commands. RepAgent errors are generally in the 9200 to 9299 range.

Adaptive Server performs actions based on the severity and recoverability of an error. Some errors are for information only, others cause Adaptive Server to retry the operation that caused the error until it succeeds, and still others indicate an error too severe to continue and RepAgent shuts down. For more information about the Adaptive Server error log file, refer to the *Adaptive Server Enterprise System Administration Guide*.

Sample error messages

This section describes some common RepAgent error messages and possible solutions.

- In this example, the RepAgent login name is not present on the Replication Server.

```
RepAgent(6): Failed to connect to Replication
Server. Please check the Replication Server,
username, and password specified to
sp_config_rep_agent. RepSvr = repserver_name, user =
RepAgent_username
RepAgent(6): This Rep Agent Thread is aborting due
to an unrecoverable communications or Replication
Server error.
```

You must either add RepAgent's login name to Replication Server or change RepAgent's login name.

- In this example, RepAgent cannot connect to Replication Server.

```
RepAgent(7): The Rep Agent Thread will retry the
connection to the Replication Server every 60
second(s). (RepSvr = repserver_name.)
```

Check Replication Server status. If Replication Server is down, resolve the problem and restart. Otherwise, wait for possible network problem to resolve.

Data server error handling

Replication Server allows user-definable error processing for data server errors. This is accomplished by creating an error class for a database and specifying responses for each error that the data server returns when the error is encountered in the database. The data server returns the defined errors to Replication Server. Table 6-2 lists the RCL commands and Adaptive Server system procedures that manage errors and error classes.

Table 6-2: RCL commands and system procedures for error processing

Command	Description
rs_helpclass	Adaptive Server system procedure that displays the name of each existing error class, function-string class, and their primary Replication Server
create error class	Creates a new error class

Command	Description
drop error class	Drops an existing error class
assign action	Specifies an error processing action for one or more data server errors
create connection	Associates an error class with a new database connection
alter connection	Associates an error class with an existing database connection

A default error class, rs_sqlserver_error_class, is provided for Adaptive Server databases.

Creating an error class

You can define a single error class to use with all databases managed by the same type of data server. For example, you can use the default Adaptive Server error class, rs_sqlserver_error_class, with any Adaptive Server database. There is no need to create another error class unless a database has special error-handling requirements.

An error class is a name used to group error action assignments. The syntax for the create error class command is:

> create error class *error_class*

For example, to create an error class named pubs2_error_class, use this command:

```
create error class pubs2_error_class
```

Initially, rs_sqlserver_error_class, the default error class that is predefined to work with Adaptive Server databases, does not have a primary site. Since you can only create server-wide error classes at a primary site for a class, you need to designate one of the Replication Servers as a primary site for a Adaptive Server error class.

You must specify a primary site before you can modify a default error class. You can designate a site as primary by executing the create error class command for a Adaptive Server error class at that site. To do this, execute create error class rs_sqlserver_error_class at the primary site. Make sure all other Replication Servers have direct or indirect routes from the primary site.

The default error action for all errors returned by a data server is
stop_replication. This is also the most serious action: it suspends replication for
the database, as if you entered the suspend connection command. To assign less
severe actions to errors you want to handle differently, use the assign action
command. See "Assigning actions to data server errors" on page 212 for more
information.

Initializing a new error class

After you have created a new error class, you can initialize it with error actions
from an error class such as the system-provided rs_sqlserver_error_class. To do
this, use the rs_init_erroractions stored procedure:

> rs_init_erroractions *new_error_class*, *template_class*

For example, to create the error class pubs2_error_class, based on the template
error class rs_sqlserver_error_class, enter:

```
rs_init_erroractions pubs2_error_class,
rs_sqlserver_error_class
```

Then use the assign action command to change the actions for individual errors.

Dropping an error class

The drop error class command drops an error class and all actions associated
with it. The error class must not be in use with an active database connection
when you drop it. The syntax for drop error class is:

> drop error class *error_class*

For example, to drop the pubs2_error_class error class, use this command:

```
drop error class pubs2_error_class
```

You cannot drop the rs_sqlserver_error_class error class.

Changing the primary Replication Server for an error class

Use the move primary command to change the primary site for an error class. This is necessary when you are changing the primary site from one Replication Server to another so that error actions can be distributed through new routes. For example, you must use this command if you are dropping from the replication system the Replication Server that is the current primary site for an error class.

Before you execute move primary, make sure that a route exists from:

- The new primary site to each Replication Server that will use the error class

- The current primary to the new primary site

- The new primary to the current primary site

The syntax for the move primary command, for error classes, is:

> move primary of error class *class_name*
> to *replication_server*

Execute the move primary command at the Replication Server that you want to designate as the new primary site for the error class.

- *class_ name* – the name of the error class whose primary Replication Server is to be changed.

- *replication_server* – specifies the new primary Replication Server for the error class.

The following command changes the primary site for the pubs2_error_class error class to the TOKYO_RS Replication Server where the command is entered:

```
move primary of error class pubs2_error_class
 to TOKYO_RS
```

For the default error class, rs_sqlserver_error_class, no Replication Server is the primary site until you assign one as the primary site. You must specify a primary site before you can use the assign action command to change default error actions.

To specify a primary site for the default error class, execute the following command in that Replication Server:

```
create error class rs_sqlserver_error_class
```

After you have executed this command, you can use the move primary command to change the primary site for the error class.

Displaying error class information

The Adaptive Server rs_helpclass stored procedure displays the names of existing error classes and function-string classes in the replication system and their primary Replication Servers. For example:

```
rs_helpclass error_class
Error Class(es)               PRS for class
 --------------               ---------
rs_sqlserver_error_class      Not Yet Defined
```

Refer to Chapter 6, "Adaptive Server Stored Procedures," in the *Replication Server Reference Manual* for more information about rs_helpclass command.

Assigning actions to data server errors

The assign action command specifies the action to take for errors that a data server can return to Replication Server. The syntax for the assign action command is:

```
assign action {ignore | warn | retry_log | log |
 retry_stop | stop_replication}
 for error_class
 to data_server_error [, data_server_error]...
```

For example, to instruct Replication Server to ignore Adaptive Server errors 5701 and 5703:

```
assign action ignore
 for rs_sqlserver_error_class
 to 5701, 5703
```

The default error class provided for Adaptive Server databases is rs_sqlserver_error_class. You must create this error class at a primary site before you can use the assign action command to change default error actions. The *data_server_error* parameter is the data server error number.

Enter one of the six possible error actions at the Replication Server where the error class was created. These actions are listed in Table 6-3, in order of severity: ignore is the least severe action and stop_replication is the most severe.

When a transaction causes multiple errors, Replication Server chooses just one action—the most severe action assigned to any of the errors that occurred. To return an error to the default error action, stop_replication, you must reassign it explicitly.

Table 6-3: Replication Server actions for data server errors

Action	Description
ignore	Assume that the command succeeded and that there is no error or warning condition to process. This action can be used for a return status that indicates successful execution.
warn	Log a warning message, but do not roll back the transaction or interrupt execution.
retry_log	Roll back the transaction and retry it. The number of retry attempts is set with the configure connection command. If the error continues after retrying, write the transaction into the exceptions log, and continue, executing the next transaction.
log	Roll back the current transaction and log it in the exceptions log; then continue, executing the next transaction.
retry_stop	Roll back the transaction and retry it. The number of retry attempts is set with the configure connection command. If the error recurs after retrying, suspend replication for the database.
stop_replication	Roll back the current transaction and suspend replication for the database. This is equivalent to using the suspend connection command. This action is the default.
	Since this action stops all replication activity for the database, it is important to identify the data server errors that can be handled without shutting down the database connection, and assign them to another action.

Displaying assigned actions for error numbers

Execute the rs_helperror stored procedure to display the action assigned for an error number. The syntax for the rs_helperror stored procedure is:

    rs_helperror *server_error_number* [, v]

where *server_error_number* parameter is the data server error number of the error you want information for. The v parameter specifies "verbose" reporting. When you supply this option, rs_helperror also displays the Adaptive Server error message text, if available. Refer to Chapter 6, "Adaptive Server Stored Procedures," in the *Replication Server Reference Manual* for more details on using rs_helperror command.

Exceptions handling

When a transaction submitted by Replication Server fails, Replication Server records the transaction in the exceptions log in the RSSD. The replication system administrator at the site must resolve the transactions in the exceptions log. See "Accessing the exceptions log" on page 215.

Transactions can fail due to errors such as duplicate keys, column value checks, and insufficient disk space. They may also be rejected for reasons such as insufficient permissions, version control conflicts, and invalid object references.

Because skipping a transaction causes inconsistency and can have an adverse affect on the system, you should review on a regular basis any transactions that have been recorded in the exceptions log and resolve them. The best resolution for a transaction may depend on the client application that originated it. For example, if a failed transaction corresponds to a real-world event, such as a cash withdrawal, the transaction must somehow be applied.

Refer to the *Replication Server Troubleshooting Guide* for more information on the implications of skipping a transaction.

Handling failed transactions

This section outlines the recommended process for handling failed transactions that require manual intervention.

Suspend database connection

When a data server begins rejecting transactions because of a temporary failure, such as lack of space in a database or log file, you can suspend the database connection until the error is corrected.

If the database connection is not suspended, Replication Server writes the transactions into the exceptions log for the database. Since these transactions must then be resolved manually, you can save time by shutting down the connection until the error condition is corrected.

While a database connection is suspended, Replication Server stores transactions in a stable queue. When the connection is resumed, the stored transactions are sent to the data server.

To stop the flow of transactions from a Replication Server to a database, use the suspend connection command:

```
suspend connection to data_server.database
```

The command requires sa permission and must be entered at the Replication Server that manages the database.

Analyze and resolve the problem

You then need to determine why the transaction failed, make corrections or adjustments, and resubmit the transaction. For example, if a transaction failed because the maintenance user had insufficient permissions, grant the maintenance user the needed permissions and retry the transaction.

If you are resolving transactions in the exceptions log:

1   Retrieve a list of the transactions from the exceptions log. See "Accessing the exceptions log" on page 215.

2   Investigate the transactions to determine the cause of failure and the best method for resolution.

3   Resolve the transactions according to your plan. For example, you might correct a permissions problem and then resubmit a transaction.

4   Delete resolved transactions from the exceptions log. See "Deleting transactions from the exceptions log" on page 218.

Resume the connection

Restart the flow of transactions for a suspended database connection with the resume connection command. The same command is used whether you suspended the connection intentionally, using the suspend connection command, or whether it was suspended by Replication Server as the result of an error action. The syntax for resume connection is:

resume connection to *data_server.database*
[skip transaction]

The command requires sa permission and must be entered at the Replication Server that manages the database.

Use the skip transaction clause to instruct Replication Server to ignore the first transaction in the queue. You may need to do this if a transaction continues to fail each time you resume the connection.

Accessing the exceptions log

Replication Manager provides a graphical interface to view and manage the transactions in the exeptions log.

Displaying transactions in the exceptions log

You can display a summary list of all transactions in the exceptions log using the rs_helpexception stored procedure. The syntax for the rs_helpexception stored procedure is:

    rs_helpexception [*transaction_id*, [, v]]

If you supply a valid *transaction_id* and v for "verbose" reporting, rs_helpexception displays a detailed description of a transaction. Use rs_helpexception with no parameters to obtain *transaction_id* numbers for all transactions in the exceptions log.

Querying the exceptions log system tables

You can join the rs_exceptshdr and rs_exceptscmd system tables on the sys_trans_id column.

You can also join the rs_exceptscmd and rs_systext system tables to retrieve the text of a transaction. To do this, join the cmd_id column in rs_exceptscmd to the parentid column in rs_systext.

Figure 6-1 illustrates the exceptions log system tables.

Figure 6-1: Exceptions log system tables

rs_exceptshdr

sys_trans_id	rs_id
rs_trans_id	binary
apps_trans_name	varchar
orig_siteid	int
orig_site	varchar
orig_db	varchar
orig_time	datetime
orig_user	varchar
error_siteid	int
error_site	varchar
error_db	varchar
long_time	datetime
ds_error	int
ds_errmsg	varchar
error_src_line	int
error_proc	int
err_output_line	int
log_reason	char
trans_status	smallint
retry_status	smallint
app_usr	varchar
app_pwd	varchar

rs_exceptscmd

sys_trans_id	rs_id
src_cmd_line	int
output_cmd_index	int
cmd_type	char
cmd_id	rs_id

rs_systex

prsid	int
parentid	rs_id
texttype	char
sequence	int
textval	varchar

The rs_exceptshdr system table contains descriptive information about the transactions in the exceptions log, including the following:

• User-assigned transaction name

• Site and database where the transaction originated

• User at the origin site who submitted the transaction

• Information about the error that caused the transaction to be recorded in the exceptions log

To retrieve a list of the excepted transactions for a given database, use, for example, the following query:

```
select * from rs_exceptshdr
 where error_site = 'data_server'
 and error_db = 'database'
 order by log_time
```

To retrieve the source and output text for a transaction with a given system transaction ID, use:

```
select t.texttype, t.sequence,
 t.textval
 from rs_systext t, rs_exceptscmd e
 where e.sys_trans_id = sys_trans_id
and t.parentid = e.cmd_id
 order by e.src_cmd_line, e.output_cmd_index,
 t.sequence
```

Refer to Chapter 8, "Replication Server System Tables," in the *Replication Server Reference Manual* for a list of all of the columns in these Replication Server system tables.

Deleting transactions from the exceptions log

To delete a transaction from the exceptions log, use the rs_delexception stored procedure.

rs_delexception [*transaction_id*]

With no parameters, rs_delexception displays a summary of transactions in the exceptions log. If you supply a valid *transaction_id*, rs_delexception deletes a transaction. You can find the *transaction_id* for a transaction by using either rs_helpexception or rs_delexception with no parameters.

See Chapter 3, "Managing Replication Server with Sybase Central" in the *Replication Server Administration Guide Volume 1* for information about viewing queue data.

DSI duplicate detection

The DSI records the last transaction committed or written into the exceptions log so that it can detect duplicates after a system restart. Each transaction is identified by a unique origin database ID and an origin queue ID that increases for each transaction.

The last transaction committed from each origin database is recorded at a data server by executing the function strings defined for the data server's function-string class. For the system-defined classes, this is done in the function string for a commit command, that is, the rs_commit function. Every function-string class supports the rs_get_lastcommit function, which returns the origin_qid and secondary_qid for each origin database. The secondary_qid is the ID of the queue used for subscription materialization or dematerialization.

The origin_qid and secondary_qid for the last transaction written into the exceptions log from each origin is recorded into the rs_exceptslast system table. However, transactions logged explicitly by the sysadmin log_first_tran command are not recorded in this system table. These transactions are logged, but they are not skipped.

When a DSI is started or restarted, it gets the origin_qid returned by the rs_get_lastcommit function and the one stored in the rs_exceptslast system table. It assumes that any transaction in the queue with an origin_qid less than the larger of these two values is a duplicate and ignores it.

If the origin_qid values stored in a data server or the rs_exceptslast system table are modified by mistake, non-duplicate transactions may be ignored or duplicate transactions may be reapplied. If you suspect that this is happening in your system, check the values stored and compare them with the transactions in the database's stable queue to determine the validity of the values. If the values are wrong, you must modify them directly.

Refer to the *Replication Server Troubleshooting Guide* for details on how to dump transactions in a queue.

Duplicate detection for system transactions

truncate table and certain supported DDL commands are not logged, although they can be replicated to standby and replicate databases. Refer to "Supported DDL commands and system procedures" on page 64 for a list of DDL commands supported for replication. Refer to the *Adaptive Server Enterprise Reference Manual* for information about each DDL command.

Replication Server copies these commands as system transactions, in which Replication Server "sandwiches" the truncate table or similar command between two complete transactions. Execution of the first transaction is recorded in the replicate database in the secondary_qid column of the rs_lastcommit table and in the origin_qid column of that table. If Replication Server records the second transaction, the system transaction has completed, and Replication Server clears the secondary_qid column.

If there is a system failure, and you see the following error message when the system restarts:

```
5152 DSI_SYSTRAN_SHUTDOWN,"There is a system
transaction whose state is not known. DSI will be
shutdown."
```

a system command has not completed, and the connection shuts down. You must verify whether the command within the system transaction has executed at the replicate database.

- If the command has executed, or if you choose to execute the command yourself, you can skip the first transaction in the queue and continue with the second transaction when you resume the connection. At the replicate Replication Server, enter:

  ```
  resume connection to data_server.database
  skip transaction
  ```

- If the command has not executed, you can fix the problem and then execute the first command in the queue. At the replicate Replication Server, enter:

  ```
  resume connection to data_server.database
  execute transaction
  ```

You *must* include the skip transaction or execute transaction clause with resume connection. Otherwise, Replication Server does not reset the secondary_qid correctly, and the error message reappears.

This chapter describes how to prevent or recover from certain kinds of system failures in a replication system.

Topic	Page
How to use recovery procedures	222
Configuring the replication system to support Sybase Failover	222
Configuring the replication system to prevent data loss	226
Recovering from partition loss or failure	231
Recovering from truncated primary database logs	235
Recovering from primary database failures	238
Recovering from RSSD failure	241
Recovery support tasks	256

While Replication Server tolerates most failure conditions and recovers from them automatically, some failures require user intervention. This chapter identifies those failures and provides procedures for recovery. These procedures are designed to maintain the integrity of the replication system by recovering lost and corrupted data and restoring that data to its previous state.

You should design, install, and administer your replication system with backup and recovery in mind. We assume that dumps are performed on a regular basis and that appropriate tools and settings for handling recovery are in place. See "Creating coordinated dumps" on page 230 for details on performing dumps.

In this chapter, the **current** Replication Server refers to the one with a database (for example, RSSD) that you are recovering. An **upstream** Replication Server has a direct or indirect route to the current Replication Server. A **downstream** Replication Server is one to which the current Replication Server has a direct or indirect route.

How to use recovery procedures

When using recovery procedures in this chapter, always write down or check off recovery steps as you perform them. Such information can help Sybase Technical Support determine where you are in the recovery procedure, if necessary.

Table 7-1 lists failure conditions described in this chapter, and indicates where to find information on corresponding failure symptoms and recovery procedures.

Table 7-1: Overview of available recovery procedures

Failure condition	For symptoms and recovery procedures
Replication Server partition loss or failure	"Recovering from partition loss or failure" on page 231
Truncated primary database logs	"Recovering from truncated primary database logs" on page 235
Primary database failure	"Recovering from primary database failures" on page 238
RSSD failure	"Recovering from RSSD failure" on page 241

Recovery procedures are *only* intended for the specific situations noted in this chapter. Do not use recovery procedures for replication system problems such as failure to replicate data.

Warning! Use recovery procedures in this chapter only for the failure condition specific to the procedure. Attempting to use recovery procedures on conditions other than those specified can complicate your problem and require more drastic recovery actions.

Refer to the *Replication Server Troubleshooting Guide* for help in diagnosing and correcting problems.

Configuring the replication system to support Sybase Failover

This section describes how Replication Server version 12.0 and later supports Sybase Failover available in Adaptive Server Enterprise version 12.0 and later.

Overview

Sybase Failover allows you to configure two version 12.0 and later Adaptive Servers as companions. If the primary companion Adaptive Server fails, that server's devices, databases, and connections can be taken over by the secondary companion Adaptive Server.

You can configure a high availability system either asymmetrically or symmetrically.

An *asymmetric configuration* includes two Adaptive Servers that are physically located on different machines, but share the same system devices, system/master databases, user databases, and user logins. These two servers are connected so that if one of the servers is brought down, the other assumes its workload. The secondary Adaptive Server acts as a "hot standby" and does not perform any work until failover occurs.

A *symmetric configuration* also includes two Adaptive Servers running on separate machines, but each Adaptive Server is fully functional with its own system devices, system/master databases, user databases, and user logins. If failover occurs, either Adaptive Server can act as a primary or secondary companion for the other Adaptive Server.

In either setup, the two machines are configured for dual access, which makes the disks visible and accessible to both servers.

In a replication system, where Replication Server makes many connections to Adaptive Servers, you can enable or disable Failover support of the database connections initiated by a Replication Server to Adaptive Servers. When you enable Failover support, Replication Servers connected to an Adaptive Server that fails are automatically switched to the second companion machine, reestablishing network connections.

See the Adaptive Server Enterprise documentation for more detailed information about how Sybase Failover works in Adaptive Server. See Appendix B, "High Availability on Sun Cluster 2.2" for information about Failover support for Replication Server.

Enabling Failover support in Replication Server

You enable Failover support for each Replication Server in your system; once for the RSSD connection, and once for all other database connections from the specified Replication Server to Adaptive Servers.

You cannot enable Failover support for individual connections, except the RSSD connection.

The default for Failover support in Replication Server is "off" for all connections from a Replication Server to Adaptive Servers.

For continuing replication, you should enable Failover support for all connections. However, in some cases you may want to disable connection Failover when the secondary server's workload exceeds its capacity.

How Sybase Failover works with Replication Server

To configure Sybase Failover from Replication Server to Adaptive Server, the Adaptive Server must be configured to allow connection failover.

When Adaptive Servers are in failover companion mode and the primary companion fails, the secondary companion takes over the workload. Incomplete transactions or operations that require updates to the RSSD fail. Replication Server retries existing connections, but new connections are failed over.

For Data Server Interface (DSI) connections, the DSI retries failed transactions after a brief sleep.

For RSSD connections, user commands that are executed during failover do not succeed. Internal operations (such as updates to locator, disk segment, and so on) should not fail. Replication of RSSD objects should be covered by the DSI.

Asynchronous commands (for example, subscription, routing, and standby commands) may be rejected or encounter errors and require recovery if the commands have been accepted but not completed. For example, a create subscription command may have been accepted, but the subscription may still be being created.

Note Failover support is not a substitute for warm standby. While warm standby keeps a copy of a database, Failover support accesses the same database from a different machine. Failover support works the same for connections from Replication Server to warm standby databases.

Requirements

- To enable Failover support, a Replication Server must connect to Adaptive Servers that are version 12.0 or later and configured for Failover.

- Failover of Replication Server System Databases (RSSDs) and user databases is configured directly through the Adaptive Server.

- Failover support responds only to failover of the Adaptive Servers; that is, failover of Replication Servers is not supported.

- Adaptive Server is responsible for the RepAgent thread failover and its reconnection to Replication Server after failover/failback.

- Each Replication Server configures its own connections.

Enabling Failover support for an RSSD connection

To enable Failover support for an RSSD connection, use either of the following methods:

- Use rs_init when you install a new Replication Server.

  For instructions, refer to Chapter 2, "Configuring Replication Server and Adding New Databases," in the *Replication Server Configuration Guide* for your platform.

- Edit the Replication Server configuration file after you have installed the Replication Server.

  a Use a text editor to open the Replication Server configuration file. The default file name is the Replication Server name with a "*.cfg*" extension.

  The configuration file contains one line per entry.

  b Find the line "RSSD_ha_failover=no" and change it to:

      RSSD_ha_failover=yes

  c To disable Failover support for an RSSD connection, change the "RSSD_ha_failover=yes" to:

      RSSD_ha_failover=no

  These changes take affect immediately; that is, you do not have to restart Replication Server to enable Failover support.

Enabling Failover support for non-RSSD database connections

You can enable Failover support for new database connections from the Replication Server to Adaptive Servers using the procedure in this section.

For more information about Sybase Failover, refer to the Adaptive Server Enterprise book *Using Sybase Failover in a High Availability System*.

❖ **Enabling Failover support using *configure replication server***

1   If necessary, start the Replication Server, as described in the section "Starting Replication Server" in Chapter 4.

2   Log in to the Replication Server:

```
isql -Uuser_name -Ppassword -Sserver_name
```

where *user_name* must have Administrator privileges. Specify the name of the Replication Server using the -S flag.

When your login is accepted, isql displays a prompt:

```
1>
```

3   Enter the following RCL command:

```
configure replication server
set ha_failover to 'on'
```

Configuring the replication system to prevent data loss

This section contains recommended measures for preventing data loss in the event of an irrecoverable database error. If used properly, these measures allow you to restore replicated data using the system recovery procedures.

Save interval for recovery

Replication Servers are designed to store messages from their source and forward them to their destinations. To increase the chances of recovering online messages after rebuilding stable queues, you can set save intervals, measured in minutes, for routes between Replication Servers. A save interval is the amount of time that a message is stored after it has been forwarded. You can also set save intervals for a physical or logical database connection from a Replication Server, allowing Replication Server to save messages in a DSI outbound queue.

To find the current save interval for a route or connection, use the admin who, sqm command. The Save_Int:Seg column holds two values. The value preceding the colon is the save interval. The value after the colon is the first saved segment in the stable queue.

Details on setting save intervals for routes and connections are described in the following sections.

Routes between Replication Servers

If the Replication Server has suspended routes, or if a network or data server connection is down, a backlog of messages may accumulate in the Replication Server stable queues. The chance of recovering these messages decreases with time. Source Replication Servers may already have deleted messages from their stable queues and database logs may already have been truncated.

When you set the *save_interval* for each route between Replication Servers, you allow each Replication Server to retain messages for a minimum period of time after the next site in the route acknowledges that it has received the messages. The availability of these messages increases the chance of recovering online messages after queues are rebuilt.

For example, in Figure 7-1 on page 228, Replication Server TOKYO_RS maintains a direct route to MANILA_RS, and MANILA_RS maintains a direct route to SYDNEY_RS.

TOKYO_RS retains messages for a period of time after MANILA_RS has received them. If MANILA_RS experiences a partition failure, it requires that TOKYO_RS to resend the backlogged messages. MANILA_RS can also retain messages to allow SYDNEY_RS to recover from failures.

When all of the messages stored on a stable queue segment are at least as old as the *save_interval* setting, Replication Server deletes the segment so it can be reused.

Figure 7-1: Save interval example
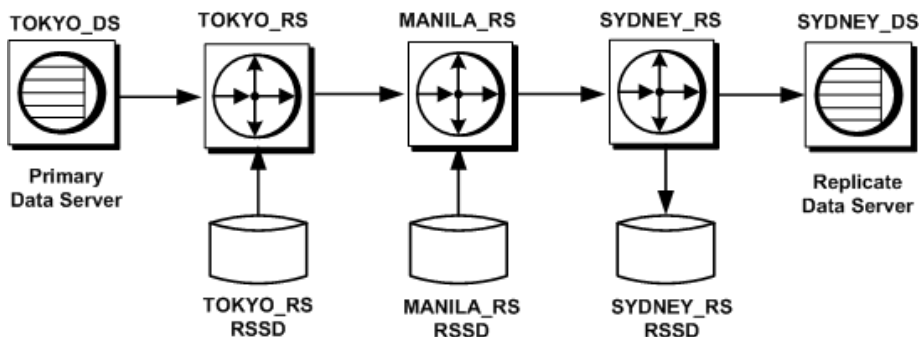
Setting the save interval for routes

To set the *save_interval* for a route, execute the alter route command at the source Replication Server. Using as an example the replication system in Figure 7-1, here is the command to set Replication Server TOKYO_RS to save for one hour any messages destined for MANILA_RS:

```
alter route to MANILA_RS
    set save_interval to '60'
```

By default, the *save_interval* is set to 0 (minutes). For systems with low volume, this may be an acceptable setting for recovery, since Replication Server does not delete messages immediately after receiving acknowledgment from destination servers. Rather, messages are deleted periodically in large chunks.

However, to accommodate the volume and activity of sites that receive distributions from the Replication Server and to increase the chance of full recovery from database or partition failures, you may want to change the *save_interval* setting.

In case of a partition failure on the stable queues, be sure your setting allows adequate time to restore your system. Consider also the size of the partitions that are allocated for backlogged messages. Partitions must be large enough to hold the extra messages.

Refer to the *Replication Server Design Guide* capacity planning guidelines for help in determining queue space requirements.

Connections between Replication Servers and data servers

When you set the *save_interval* for a physical or logical connection between a Replication Server and a data server and database, you allow Replication Server to save transactions in the DSI queue. You can restore the backlogged transactions using the sysadmin restore_dsi_saved_segments command. Refer to the *Replication Server Reference Manual* for more information.

You can use these saved transactions to resynchronize a database after it has been loaded to a previous state from transaction dumps and database dumps.

For example, in Figure 7-1, if the replicate data server SYDNEY_DS that is connected to Replication Server SYDNEY_RS experiences a failure, it can obtain the messages saved in the DSI queue at SYDNEY_RS to resynchronize the replicate database after it has been restored.

You can also use the *save_interval* for setting up a warm standby of a database that holds some replicate data or one that receives applied functions.

Setting the save interval for connections

To set the *save_interval* for a database connection, execute the alter connection command at the Replication Server. For example, here is the command to set Replication Server SYDNEY_RS to save for one hour any messages destined for its replicate data server SYDNEY_DS.

```
alter connection to SYDNEY_DS.pubs2
    set save_interval to '60'
```

By default, the *save_interval* is set to 0 (minutes).

You can also configure the save intervals for the DSI queue and the materialization queue for a logical connection. See "Configuring logical connection save intervals" on page 109 for details.

Backing up the RSSDs

If you cannot recover an RSSD's most recent state, RSSD recovery can be complex. The procedure you use depends on how much RSSD activity there has been since the last dump. See Table 7-3 on page 242 for a list of possible recovery procedures.

You should perform a dump of your RSSDs following any replication DDL, such as changing routes or adding subscriptions.

Creating coordinated dumps

When you must recover a primary database by restoring backups, you must also make sure that replicate data in the affected databases at other sites is consistent with the primary data. To provide for consistency after a restore on multiple data servers, Replication Server provides a method for coordinating database dumps and transaction dumps at all sites in a replication system.

You initiate a database dump or transaction dump from the primary database. RepAgent retrieves the dump record from the log and submits it to Replication Server so that the dump request can be distributed to the replicate sites. The method ensures that all of the data can be restored to a known point of consistency.

You can only use a coordinated dump with databases that store either primary data or replicated data but not both. You initiate a coordinated dump from within a primary database.

The process for coordinating dumps works as follows:

- In each function-string class assigned to the databases involved, the Replication System Administrator at each site creates function strings for the rs_dumpdb and rs_dumptran system functions. The function strings should call stored procedures that execute the dump database and dump transaction or equivalent commands and update the rs_lastcommit system table. Refer to the *Replication Server Reference Manual* for examples.

- You must be using a function-string class, such as a derived class, in which you can create and modify function strings. See "Managing function-string classes" on page 26 for more information.

- Using the alter connection command, the replication system administrator at each replicate site configures the Replication Servers to enable a coordinated dump.

- When a dump is started in a primary database, the RepAgent transfers the dump database or dump transaction log record to the Replication Server.

- Replication Server distributes an rs_dumpdb or rs_dumptran function call to sites that have subscriptions for the replicated tables in the database.

- The rs_dumpdb and rs_dumptran function strings at the replicate sites execute the customized stored procedures at each replicate site.

Recovering from partition loss or failure

When a Replication Server detects a failed or missing partition, it shuts down the stable queues that are using the partition and logs messages about the failure. Restarting Replication Server does not correct the problem. You must drop the damaged partition and rebuild the stable queues.

Complete recovery depends on the volume of messages cleared from the queue and on how soon you apply the recovery procedure after the failure occurs. If a Replication Server maintains minimal latency in the replication system, only the most recent messages are lost when its queues are rebuilt.

If a partition fails in a primary Replication Server, you can usually resend lost messages from their source using an off-line database log. If partitions fail in a replicate Replication Server, you need to recover from the stable queue of the upstream Replication Server.

In some cases, using an off-line log may be the only way you can recover your messages. If the Replication Server has suspended routes or connections, or if a network or data server connection goes down, a backlog may have accumulated in the Replication Server stable queues. Unless you have specified a save interval setting that can cover the backlog, your chance of recovering these messages decreases with time. Source Replication Servers may have already deleted messages from their stable queues and may have truncated the database logs.

Note For details on setting and displaying the save interval for recovery purposes, see "Recovering from partition loss or failure" on page 231.

Table 7-2 summarizes when to use and where to locate the appropriate recovery procedure for partition loss or failure.

Table 7-2: Overview of symptoms and procedures

Symptom	Use this procedure
Replication Server detects lost, damaged, or failed stable queue.	"Procedure for recovering from partition loss or failure" on page 232.
Message loss occurred because a backlog existed in the failed Replication Server and there were insufficient messages saved at the previous site.	"Message recovery from off-line database logs" on page 233.
In addition to message loss, database logs have been truncated. Either the secondary truncation point is invalid or the dbcc settrunc('ltm', 'ignore') command, was executed to truncate log records that have not been transferred by RepAgent to the Replication Server.	Use "Truncated message recovery from the database log" on page 236 to recover the database log. Then use "Message recovery from off-line database logs" on page 233 to rebuild the stable queues and recover lost messages.

Procedure for recovering from partition loss or failure

To recover from Replication Server partition loss or failure, perform the following steps:

1   Log in to the Replication Server and drop the failed partition:

```
drop partition logical_name
```

Replication Server does not immediately drop a partition that was in use. If the partition is undamaged, Replication Server drops it only after all of the messages it holds are delivered and deleted.

Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about drop partition command.

2   If the failed partition was the only one available to the Replication Server, add another one to replace it:

```
create partition logical_name
on 'physical_name' with size size
[starting at vstart]
```

Refer to the *Replication Server Reference Manual* for more information.

3   Since the partition is damaged, you must rebuild the stable queues:

```
rebuild queues
```

See "Rebuilding queues online" on page 257 for a description of this process.

When all stable queues on the partition are removed, Replication Server drops the failed partition from the system and rebuilds the queues using the remaining partitions.

4   After rebuilding the queues, check the Replication Server logs for loss detection messages.

See "Loss detection after rebuilding stable queues" on page 260 for background and details.

5   If Replication Server detected message loss, you can:

*   Perform "Message recovery from off-line database logs" on page 233, or

- Request that Replication Server ignore the loss by executing the ignore loss command for the database on the Replication Server where the loss was detected.

Note If you specify that Replication Server ignore message losses and you have rebuilt the queues of a Replication Server that is part a route, you must re-create subscriptions at the destination or use the rs_subcmp program with the -r flag to reconcile primary and replicate data.

Message recovery from off-line database logs

If the online log does not contain all the data needed to recover, you must load an older version of the primary database into a separate database and start RepAgent for the database. Although RepAgent is accessing a different database, it submits messages as if they were from the database whose messages you are recovering.

To recover messages from off-line logs after a partition failure:

1   Restart Replication Server in standalone mode, using the -M flag.

2   Log in to the Replication Server, and enter:

```
rebuild queues
```

See "Rebuilding queues online" on page 257 for a description of this process.

3   Inspect the Replication Server logs at each site for "Checking Loss" messages.

See "Determining which dumps to load" on page 266 for background and details on examining these messages.

4   Use the date and time in the error log messages to determine which dumps to load.

5   Enable RepAgent for a temporary recovery database, using the sp_config_rep_agent system procedure.

```
sp_config_rep_agent temp_dbname, 'enable', \
'rs_name', 'rs_user_name', 'rs_password'
```

See "Setting Replication Server configuration parameters" on page 90 in the *Replication Server Administration Guide Volume 1* for information about configuring RepAgent.

6   Load the database dump and the first transaction log dump in to a temporary recovery database.

7   Start RepAgent in recovery mode for the temporary database:

```
sp_start_rep_agent temp_dbname, 'recovery', \
'connect_dataserver', 'connect_database', \
'rs_name', 'rs_user_name', 'rs_password'
```

where *"connect_dataserver"* and *"connect_database"* specify the original primary data server and database.

RepAgent transfers data in the transaction log of the temporary recovery database to the original primary database. When RepAgent completes scanning the transaction log, it shuts down.

8   Verify that RepAgent has replayed the transaction log of the temporary database. Use either of these methods:

• Check the Adaptive Server log for a message similar to the following:

```
Recovery of transaction log is complete. Please
load the next transaction log dump and then start
up the Rep Agent Thread with sp_start_rep_agent,
with 'recovery' specified.
```

Then, perform the appropriate actions.

• From Adaptive Server, execute the sp_help_rep_agent system procedure for recovery:

```
sp_help_rep_agent dbname, 'recovery'
```

This procedure displays RepAgent's recovery status. If the recovery status is "not running" or "end of log," then recovery is complete. You can load the next transaction log dump. If the recovery status is "initial" or "scanning," either the log has not been replayed, or the replay is not complete.

9   If you have performed another recovery procedure since you performed the last database dump, you may need to change the database generation number after loading a transaction log dump. See "Determining database generation numbers" on page 267.

10   If there are more transaction log dumps to load, repeat the following three steps for each dump:

a   Load the next transaction log dump. (Be sure to load the dumps in the correct order.)

b   Restart RepAgent in recovery mode.

    c   Watch the Adaptive Server log for the completion message or use sp_help_rep_agent.

11   Check the Replication Server logs for loss detection messages.

No losses should be detected unless you failed to load the database to a state old enough to retrieve all of the messages.

See "Loss detection after rebuilding stable queues" on page 260 for background and details.

12   Restart the Replication Server in normal mode.

13   Restart RepAgent for the original primary data server and database in normal mode.

Message recovery from the online database log

To recover messages that are still in the online log at the primary database, perform the following steps:

1   Stop all client activity.

2   Restart RepAgent for the primary database in recovery mode.

This process causes RepAgent to scan the log from the beginning so that it retrieves all messages.

Recovering from truncated primary database logs

This section describes how to recover from failures caused by truncating a primary transaction log before Replication Server has received the messages.

This situation typically occurs if RepAgent, a Replication Server (managing a primary database), or a network between them is down for a long time and RepAgent or Replication Server is unable to read records from the transaction log. The secondary truncation point cannot be moved, which prevents Adaptive Server from truncating the log and causes the transaction log of the primary database to fill up. You can then remove the secondary truncation point by executing sp_stop_rep_agent followed by dbcc settrunc (ltm, ignore).

When a failed component returns to service, messages are missing at the Replication Server. Depending on the status of the lost messages, use one of the following procedures:

*   If messages are still in the online log at the primary database (which is unlikely), see "Message recovery from the online database log" on page 235.

*   If messages have been truncated from the online database log, see "Truncated message recovery from the database log" on page 236.

Truncated message recovery from the database log

In this procedure, you must load a previous database dump and transaction log dumps into a temporary recovery database. Then connect a RepAgent to that database to transmit the truncated log to the Replication Server. After the missing log records are recovered, you can restart the system using the regular primary database.

Using a temporary recovery database permits transaction recovery from clients that continued to use the primary database after its log was truncated.

Note Use the temporary database exclusively for recovering messages. Any modification to the database prevents you from loading the next transaction log dump. Also limit the activity on the original primary database so that the recovery can be completed before the transaction log on the original primary database must be dumped and truncated again.

To replay off-line transaction logs, follow these steps:

1   Create a temporary database such that the sysusages tables are similar in both the original and the temporary databases. To do this, you must use the same sequence of create database and alter database commands when creating the temporary database as were used to create the original database.

2   Shut down Replication Server.

3   Restart Replication Server in standalone mode, using the -M flag.

4   Log in to the Replication Server and execute the set log recovery command for each primary database you are recovering.

    See "Setting log recovery for databases" on page 264.

This command puts the Replication Server into loss detection mode for the databases. Replication Server logs a message similar to the following:

```
Checking Loss for DS1.PDB from DS1.PDB
    date=Nov-01-1995 10:35am
    qid=0x01234567890123456789
```

5   Execute the allow connections command to allow Replication Server to accept connections only from other Replication Servers and from RepAgents in recovery mode.

> **Note** If you attempt to connect to this Replication Server by automatically restarting RepAgent in normal mode with scripts, the Replication Server rejects the connection. You must restart RepAgent in recovery mode while pointing to the correct off-line log. This step allows you to resend old transaction logs before current transactions are processed.

6   Load the database dump into the temporary primary database.

7   Load the first or next transaction log dump into the temporary primary database.

8   Start the RepAgent for the temporary database in recovery mode:

```
sp_start_rep_agent temp_dbname, 'recovery',
'connect_dataserver', 'connect_database',
'repserver_name', 'repserver_username',
'repserver_password'
```

where *connect_dataserver* and *connect_database* specify the original primary data server and database.

RepAgent transfers data in the transaction log of the temporary recovery database to the original primary database. When RepAgent completes scanning the current transaction log, it shuts down.

9   Verify that RepAgent has replayed the transaction log of the temporary database.

a   Check the Adaptive Server log for the following message:

```
Recovery of transaction log is complete. Please
load the next transaction log dump and then start
up the Rep Agent Thread with sp_start_rep_agent,
with 'recovery' specified.
```

and perform the appropriate actions, or

b   Execute admin who_is_down.

If the RepAgent reports "down," load the next transaction log.

10 Repeat steps 7 through 9 until all transaction logs have been processed.

You are now ready to resume normal replication from the primary database.

11 Shut down Replication Server, which is still in standalone mode.

12 Execute the following commands:

```
rs_zeroltm data_server, database
dbcc settrunc('ltm', 'valid')
```

Note You may need to execute rs_zeroltm to clear the locator information.

13 Restart Replication Server in normal mode.

14 Restart RepAgent for both the primary database and RSSD using sp_start_rep_agent.

15 If you have performed another recovery procedure since you performed the last database dump, you may need to change the database generation number after loading a transaction log dump. See "Determining database generation numbers" on page 267.

Recovering from primary database failures

Most database failures are recovered without losing any committed transactions. No special Replication Server recovery procedure is needed if the database recovers on restart—Replication Server performs a handshake with the database, ensuring that no transactions are lost or duplicated in the replication system.

If a primary database fails and you are unable to recover all committed transactions, you must load the database to a previous state and follow a recovery procedure designed to restore consistency at the replicate sites.

Here are two possible scenarios for recovering from primary database failures:

• Recovering with coordinated dumps

If you have coordinated dumps of primary and replicate databases, you can use them to load all databases in the replication system to a consistent state.

See "Loading from coordinated dumps" on page 239 for details.

- Recovering with primary dumps only

  If you do not have coordinated dumps, you can load the failed primary database and then verify the consistency of the replicate databases with the restored primary database.

  See "Loading a primary database from dumps" on page 240 for details.

Loading from coordinated dumps

Use this procedure only if you have coordinated dumps of both primary and replicate databases. To load a primary database and all replicate databases to the same state, follow this procedure:

1   Perform steps 1 through 10 from "Loading a primary database from dumps" on page 240.

2   Suspend connections to the replicate databases that must be restored.

3   For each replicate database, log in to its managing Replication Server and execute the suspend connection command:

```
suspend connection to data_server.database
```

4   Load the replicate databases from the coordinated dumps that correspond to the restored primary database state.

5   For each replicate database, log in to its managing Replication Server and execute a sysadmin set_dsi_generation command to set the generation number for the database to the same generation number used in step 1:

```
sysadmin set_dsi_generation, 101,
    primary_data_server, primary_database,
    replicate_data_server, replicate_database
```

The parameters *primary_data_server* and *primary_database* specify the primary database for loading. The parameters *replicate_data_server* and *replicate_database* specify the replicate database for loading.

Setting the generation numbers in this manner prevents Replication Servers from applying to the replicate databases any old messages that may be in the queues.

6   For each replicate database, log in to its managing Replication Server and execute the resume connection command to restart the DSI for the database:

```
resume connection to data_server.database
```

7   Restart the primary Replication Server in normal mode.

8   Restart RepAgent for the primary database in normal mode.

Note If any subscriptions were materializing when the failure occurred, drop them and re-create them.

Loading a primary database from dumps

Use this procedure if you are loading only a primary database in a replication system. To load the database to a previous state and resolve any inconsistencies with replicate databases, follow this procedure:

1   Log in to the primary Replication Server and use the admin get_generation command to get the database generation number for the primary database:

```
admin get_generation, data_server, database
```

Write down the generation number so you have it for step 7.

2   Shut down the RepAgent for the primary database. To do this execute sp_stop_rep_agent system procedure.

```
sp_stop_rep_agent database
```

3   Suspend the DSI connection to the primary database (for exclusive use).

4   Load the database to the most recent or previous state.

This step entails loading the most recent database dump and all subsequent transaction log dumps.

Refer to the *Adaptive Server Enterprise System Administration Guide* for instructions.

5   Resume the DSI connection.

6   Enter the following commands to dump the transaction log:

```
use database
go
dbcc settrunc('ltm', 'ignore')
go
dump tran database with truncate_only
go
dbcc settrunc('ltm', 'valid')
```

```
go
```

7   Execute the dbcc settrunc command in the restored primary database to set
    the generation number to the next higher number. For example, if the
    admin get_generation command in step 1 returned 0, enter the following
    commands:

```
use database
go
dbcc settrunc('ltm', 'gen_id', 1)
```

8   Run the following command to clear the locator information:

```
rs_zeroltm data_server, database
```

9   Start RepAgent for the primary database. To do this, execute the following
    command:

```
sp_start_rep_agent database
```

10 Run the rs_subcmp program for each subscription at the replicate sites.
    Use the -r flag to reconcile the replicate data with the restored primary
    data, or drop all the subscriptions and re-create them.

    See Chapter 11, "Managing Subscriptions" in the *Replication Server
    Administration Guide Volume 1* for more information on using rs_subcmp.
    Also refer to Chapter 7, "Executable Programs," in the *Replication Server
    Reference Manual* for more information about rs_subcmp command.

Recovering from RSSD failure

If you cannot recover the RSSD's most recent database state, recovering from
an RSSD failure is a complex process. In this case, you must load the RSSD
from old database dumps and transaction log dumps.

Note It is not possible to migrate an RSSD database across platforms using
cross platform dump and load XPDL or using bcp, and so on, the only way is
to rebuild the replication system on the new platform.

The procedure for recovering an RSSD is similar to that for recovering a primary database. However, it requires more steps, since the RSSD holds information about the replication system itself. RSSD system tables are closely associated with the state of the stable queues and of other RSSDs in the replication system.

If a Replication Server RSSD has failed, you first need to determine the extent of recovery required. To do this, perform one or more of the following actions:

- When the RSSD becomes available, log in to the Replication Server and execute admin who_is_down. Some Replication Server threads may have shut down during the RSSD's period of inactivity.

  - If an SQM thread for an inbound or outbound queue or an RSI outbound queue is down, restart the Replication Server.

  - If a DSI thread is down, resume the connection to the associated database.

  - If an RSI thread is down, resume the route to the destination database.

- Check all connecting RepAgents to see if they are running with the sp_help_rep_agent system procedure. (RepAgents may have shut down in response to errors resulting from RSSD shutdown.) Restart them if necessary.

- If you cannot recover the RSSD's most recent database state, you must load it from old database dumps and transaction log dumps. See "Recovering an RSSD from dumps" on page 242.

Recovering an RSSD from dumps

The procedure you use to recover an RSSD depends on how much RSSD activity there has been since the last RSSD dump. There are four increasingly severe levels of RSSD failure, with corresponding recovery requirements. Use Table 7-3 to locate the RSSD recovery procedure you need.

Table 7-3: Recovering from RSSD failures

Activity since last RSSD dump	Use this procedure
No DDL activity	"Basic RSSD recovery procedure" on page 243
DDL activity, but no new routes or subscriptions created	"Subscription comparison procedure" on page 246
DDL activity, but no new routes created	"Subscription re-creation procedure" on page 252
New routes created	"Deintegration/reintegration procedure" on page 256

Basic RSSD recovery procedure

Use the basic RSSD recovery procedure to restore the RSSD if you have executed no DDL commands since the last RSSD dump. DDL commands in RCL include those for creating, altering, or deleting routes, replication definitions, subscriptions, function strings, functions, function-string classes, or error classes.

Certain steps in this procedure are also referenced by other RSSD recovery procedures in this chapter.

> **Warning!** Do not execute any DDL commands until you have completed this recovery procedure.

To perform basic RSSD recovery, follow these steps:

1   Shut down all RepAgents that connect to the current Replication Server.

2   Since its RSSD has failed, the current Replication Server is down. If for some reason it is not down, log in to it and use the shutdown command to shut it down.

> **Note** Some messages may still be in the Replication Server stable queues. Data in those queues may be lost when you rebuild these queues in later steps.

3   Restore the RSSD by loading the most recent RSSD database dump and all transaction dumps.

4   Restart the Replication Server in standalone mode, using the -M flag.

   You must start the Replication Server in standalone mode, because the stable queues are now inconsistent with the RSSD state. When the Replication Server starts in standalone mode, reading of the stable queues is not automatically activated.

5   Log in to the Replication Server, and get the generation number for the RSSD, using the admin get_generation command:

```
admin get_generation, data_server, rssd_name
```

   For example, the Replication Server may return a generation number of 100.

6   In the Replication Server, rebuild the queues with the following command:

```
rebuild queues
```

See "Rebuilding queues online" on page 257 for a description of this process.

7 Start all RepAgents (except the RSSD RepAgent) that connect to the current Replication Server in recovery mode:

```
sp_start_rep_agent dbname, recovery
```

Wait until each RepAgent logs a message in the Adaptive Server log that it is finished with the current log.

8 Check the loss messages in the Replication Server log, and in the logs of all the Replication Servers with direct routes *from* the current Replication Server.

• If all your routes were active at the time of failure, you probably will not experience any real data loss.

• However, loss detection may indicate real loss. Real data loss may be detected if the database logs were truncated at the primary databases, so that the rebuild process did not have enough information to recover. If you have real data loss, reload database logs from old dumps. See "Recovering from truncated primary database logs" on page 235.

• See "Loss detection after rebuilding stable queues" on page 260 for background and details on loss detection.

9 Shut down RepAgents for all primary databases managed by the current Replication Server:

```
sp_stop_rep_agent dbname
```

10 Shut down Replication Server.

11 Execute the dbcc settrunc command at the Adaptive Server for the restored RSSD. Move up the secondary truncation point.

```
use rssd_name
go
dbcc settrunc('ltm', 'ignore')
go
dump tran rssd_name with truncate_only
go
begin tran commit tran
```

```
go 40
```

Note The begin tran commit tran go 40 command moves the Adaptive Server log onto the next page.

After completing step 10 and before continuing with step 11, run the following command to clear the locater information.

```
rs_zeroltm rssd_server, rssd_name
go
```

12 Execute the dbcc settrunc command at the Adaptive Server for the restored RSSD to set the generation number to one higher than the number returned by admin get_generation in step 5.

```
dbcc settrunc ('ltm', 'gen_id', generation_number)
go
dbcc settrunc('ltm', 'valid')
go
```

Make a record of this generation number and of the current time, so that you can return to this RSSD recovery procedure, if necessary. Or, you can dump the database after setting the generation number.

13 Restart the Replication Server in normal mode.

If you performed this procedure as part of the subscription comparison or subscription re-creation procedure, the upstream RSI outbound queue may contain transactions, bound for the RSSD of the current Replication Server, that have already been applied using rs_subcmp. If this is the case, after starting the Replication Server, the error log may contain warnings referring to duplicate inserts. You can safely ignore these warnings.

14 Restart RepAgents for the RSSD and for user databases in normal mode.

If you performed this procedure as part of the subscription comparison or subscription re-creation RSSD recovery procedure, you should expect to see messages regarding RSSD losses being detected in all Replication Servers that have routes from the current Replication Server.

Subscription comparison procedure

Follow this RSSD recovery procedure if you have executed some DDL commands since the last transaction dump but you have not created any new subscriptions or routes. DDL commands in RCL include those for creating, altering, or deleting routes, replication definitions, subscriptions, function strings, functions, function-string classes, or error classes.

Warning! Do not execute any DDL commands until you have completed this recovery procedure.

Following this procedure makes the failed RSSD consistent with upstream RSSDs or consistent with the most recent database and transaction dumps (if there is no upstream Replication Server). It then makes downstream RSSDs consistent with the recovered RSSD.

If DDL commands have been executed at the current Replication Server since the last transaction dump, you may have to re-execute them.

Warning! This procedure may fail if you are operating in a mixed-version environment; that is, the Replication Servers in your replication system are not all at the same version level.

To restore an RSSD with subscription comparison, follow these steps:

1   To prepare the failed RSSD for recovery, perform steps 1 through 4 of "Basic RSSD recovery procedure" on page 243.

2   To prepare all upstream RSSDs for recovery, execute the admin quiesce_force_rsi command at each upstream Replication Server.

   •   This step ensures that all committed transactions from the current Replication Server have been applied before you execute the rs_subcmp program.

   •   Execute this command sequentially, starting with the Replication Server that is furthest upstream from the current Replication Server.

   •   Make sure that RSSD changes have been applied, that is, that the RSSD DSI outbound queues are empty.

   •   The Replication Server that is directly upstream from the current Replication Server cannot be quiesced.

3   To prepare all downstream RSSDs for recovery, execute the admin
    quiesce_force_rsi command at each downstream Replication Server.

    *   This step ensures that all committed transactions bound for the current
        Replication Server have been applied before you execute the
        rs_subcmp program.

    *   Execute this command sequentially, starting with Replication Servers
        that are immediately downstream from the current Replication Server.

    *   Make sure that RSSD changes have been applied, that is, that the
        RSSD DSI outbound queues are empty.

4   Reconcile the failed RSSD with all upstream RSSDs, using the rs_subcmp
    program.

    *   First execute rs_subcmp without reconciliation to get an idea of what
        operations it will perform. When you are ready to reconcile, use the
        -r flag to reconcile the replicate data with the primary data.

    *   You must execute rs_subcmp as the maintenance user. See Chapter 8,
        "Managing Replication Server Security" in the *Replication Server
        Administration Guide Volume 1* for more information on the
        maintenance user.

    *   In each instance, specify the failed RSSD as the replicate database.

    *   In each instance, specify the RSSD of each upstream Replication
        Server as the primary database.

    *   Start with the furthest upstream Replication Server, and proceed
        downstream for all other Replication Servers with routes (direct or
        indirect) to the current Replication Server.

    *   Reconcile each of the following RSSD system tables: rs_articles,
        rs_classes, rs_columns, rs_databases, rs_erroractions, rs_functions,
        rs_funcstrings, rs_objects, rs_publications, rs_systext, and
        rs_whereclauses.

    *   When you execute rs_subcmp on replicated RSSD tables, the where
        and order by clauses of the select statement must include all rows to
        be replicated. See "Using rs_subcmp on replicated RSSD system
        tables" on page 249 for more information.

        The failed RSSD should now be recovered.

5   Reconcile all downstream RSSDs with the RSSD for the current
    Replication Server, which was recovered in the previous step, using the
    rs_subcmp program.

- First execute rs_subcmp without reconciliation to get an idea of what operations it will perform. When you are ready to reconcile, use the -r flag to reconcile the replicate data with the primary data.

- You must execute rs_subcmp as the maintenance user. See Chapter 8, "Managing Replication Server Security" in the *Replication Server Administration Guide Volume 1* for more information on the maintenance user.

- In each instance, specify as the primary database the recovered RSSD.

- In each instance, specify as the replicate database the RSSD of each downstream Replication Server.

- Start with the Replication Servers that are immediately downstream, then proceed downstream for all other Replication Servers with routes (direct or indirect) from the current Replication Server.

- Reconcile each of the following RSSD system tables: rs_articles, rs_classes, rs_columns, rs_databases, rs_erroractions, rs_functions, rs_funcstrings, rs_objects, rs_publications, rs_systext, and rs_whereclauses.

- When you execute rs_subcmp on replicated RSSD tables, the where and order by clauses of the select statement must select all rows to be replicated. See "Using rs_subcmp on replicated RSSD system tables" on page 249 for more information.

  All downstream RSSDs should now be fully recovered.

6 If the recovering Replication Server is an ID Server, you must restore the Replication Server and database IDs in its RSSD.

   a For every Replication Server, check the rs_databases and rs_sites system tables for their IDs.

   b Insert the appropriate rows in the recovering RSSD's rs_idnames system table if they are missing.

   c Delete from the recovering RSSD's rs_idnames system table any IDs of databases or Replication Servers that are no longer part of the replication system.

   d To ensure that the rs_ids system table is consistent, execute the following stored procedure in the RSSD of the current Replication Server:

```
rs_mk_rsids_consistent
```

7   If the recovering Replication Server is *not* an ID Server, *and* a database connection was created at the recovering Replication Server after the last transaction dump, delete the row corresponding to that database connection from the rs_idnames system table in the ID Server's RSSD.

8   Perform steps 5 through 13 of "Basic RSSD recovery procedure" on page 243.

9   To complete RSSD recovery, re-execute any DDL commands executed at the current Replication Server since the last transaction dump.

Using *rs_subcmp* on replicated RSSD system tables

When executing rs_subcmp on replicated RSSD tables during RSSD recovery procedures, formulate the where and order by clauses of the select statement to select all rows that must be replicated for each system table.

Table 7-4 illustrates the general form of these select statements.

> **Note** You may need to adjust these select statements in a mixed-version environment.

Table 7-4: select statements for rs_subcmp procedure

RSSD table name	select statement
rs_articles	select * from rs_articles where prsid in *sub_select* order by *primary_key*
rs_classes	select * from rs_classes where prsid in *sub_select* order by *primary_keys*
rs_columns	select * from rs_columns where prsid in *sub_select* and rowtype = 1 order by *primary_keys*
rs_databases	select * from rs_databases where prsid in *sub_select* and rowtype = 1 order by *primary_keys*
rs_erroractions	select * from rs_erroractions where prsid in *sub_select* order by *primary_keys*
rs_funcstrings	select * from rs_funcstrings where prsid in *sub_select* and rowtype = 1 order by *primary_keys*
rs_functions	select * from rs_functions where prsid in *sub_select* and rowtype = 1 order by *primary_keys*
rs_objects	select * from rs_objects where prsid in *sub_select* and rowtype = 1 order by *primary_keys*
rs_publications	select * from rs_publications where prsid in *sub_select* order by *primary_key*
rs_systext	select * from rs_systext where prsid in *sub_select* and texttype in ('O', 'S') order by *primary_keys*
rs_whereclauses	select * from rs_whereclauses where prsid in *sub_select* order by *primary_key*

In the select statements in Table 7-4, *sub_select* represents the following sub-selection statement, which selects all site IDs that are the source Replication Servers for the current Replication Server:

```
(select source_rsid from rs_routes
    where
      (through_rsid = PRS_site_ID
       or through_rsid = RRS_site_ID)
    and
    dest_rsid = RRS_site_ID)
```

where *PRS_site_ID* is the site ID of the Replication Server managing the primary RSSD, and *RRS_site_ID* is the site ID of the Replication Server managing the replicate RSSD for the rs_subcmp operation.

For the rs_columns, rs_databases, rs_funcstrings, rs_functions, and rs_objects system tables, if rowtype = 1, then the row is a replicated row. Only replicated rows need be compared using rs_subcmp.

For each system table, the *primary_keys* are its unique indexes.

Classes and system tables

The system-provided function-string classes and error class do not initially have a designated primary site, that is, their site ID equals 0. The classes rs_default_function_class and rs_db2_function_class cannot be modified, and thus can never have a designated primary site. The classes rs_sqlserver_function_class and rs_sqlserver_error_class may be assigned a primary site and modified. The primary site of a derived function-string class is the same as its parent class.

If the recovering Replication Server was made the primary site for a function-string class or error class since the last transaction dump, the rs_subcmp procedure described earlier in this section would find orphaned rows in downstream RSSDs.

In that event, run rs_subcmp again on the rs_classes, rs_erroractions, rs_funcstrings, and rs_systext system tables. Set prsid = 0 in order to repopulate these tables with the necessary default settings. For example, use the following select statement for the rs_classes table:

```
select * from rs_classes where prsid = 0
    order by primary_keys
```

Example

Suppose you have the following Replication Server sites in your replication system, where an arrow (→) indicates a route. Site B is the failed site, and there are no indirect routes.

- A > B

- C > B

- C > D

- B > E

These Replication Servers have the following site IDs:

- A = 1

- B = 2

- C = 3

- D = 4

- E = 5

In this example, to bring the RSSDs to a consistent state, you would perform the following tasks, in the order presented, on the rs_classes, rs_columns, rs_databases, rs_erroractions, rs_funcstrings, rs_functions, rs_objects, and rs_systext system tables.

Reconciling with upstream RSSDs

1   Run rs_subcmp against the above tables, specifying site B as the replicate and site A as the primary, with prsid = 1 in the where clauses. For example, the select statement for rs_columns should look like the following:

```
select * from rs_columns where prsid in
    (select source_rsid from rs_routes
    where
        (through_rsid = 1 or through_rsid = 2)
         and dest_rsid = 2)
       and rowtype = 1
      order by objid, colname
```

2   Run rs_subcmp against the above tables, specifying site B as the replicate and site C as the primary, with prsid = 3 in the where clauses. For example, the select statement for rs_columns should look like the following:

```
select * from rs_columns where prsid in
        (select source_rsid from rs_routes
         where
          (through_rsid = 3 or through_rsid = 2)
           and dest_rsid = 2)
         and rowtype = 1
        order by objid, colname
```

Reconciling downstream RSSDs

1   Run rs_subcmp against the above tables, specifying site B as the primary and site E as the replicate, with prsid = 2 in the where clauses. For example, the select statement for rs_columns should look like the following:

```
select * from rs_columns where prsid in
       (select source_rsid from rs_routes
        where
         (through_rsid = 2 or through_rsid = 5)
          and dest_rsid = 5)
        and rowtype = 1
       order by objid, colname
```

Refer to Chapter 7, "Executable Programs," in the *Replication Server Reference Manual* for more information on rs_subcmp. Refer to Chapter 8, "Replication Server System Tables," in the *Replication Server Reference Manual* for more information on the RSSD system tables.

Subscription re-creation procedure

Follow this RSSD recovery procedure if you have created new subscriptions or other DDL since the last transaction dump, and you have not created new routes. DDL commands in RCL include those for creating, altering, or deleting routes, replication definitions, subscriptions, function strings, functions, function-string classes, or error classes.

 Warning! Do not execute any DDL commands until you have completed the subscription re-creation recovery procedure.

As with the subscription-comparison RSSD recovery procedure, following this procedure makes the failed RSSD consistent with upstream RSSDs or with the most recent database and transaction dumps (if there is no upstream Replication Server). It then makes downstream RSSDs consistent with the recovered RSSD.

In this procedure, however, you also either delete or re-create subscriptions that are in limbo due to the loss of the RSSD.

If DDL commands have been executed at the current Replication Server since the last transaction dump, you may have to reexecute them.

To restore an RSSD that requires that lost subscriptions be re-created, follow these steps:

1   To prepare the failed RSSD for recovery, perform steps 1 through 4 of "Basic RSSD recovery procedure" on page 243.

2   To prepare the RSSDs of all upstream and downstream Replication Servers for recovery, perform step 2 through 3 of "Subscription comparison procedure" on page 246.

3   Shut down all upstream and downstream Replication Servers affected by the previous step. Use the shutdown command.

4   Restart all upstream and downstream Replication Servers in standalone mode, using the -M flag.

   All RepAgents connecting to these Replication Servers shut down automatically when you restart the Replication Servers in standalone mode.

5   To reconcile the failed RSSD with all upstream RSSDs, perform step 4 of "Subscription comparison procedure" on page 246.

   The failed RSSD should now be recovered.

6   To reconcile all downstream RSSDs with the RSSD for the current Replication Server, perform step 5 of "Subscription comparison procedure" on page 246.

7   If the recovering Replication Server is an ID Server, to restore the IDs in its RSSD, perform step 6 of "Subscription comparison procedure" on page 246.

8   If the recovering Replication Server is *not* an ID Server *and* a database connection was created at the recovering Replication Server after the last transaction dump, perform step 7 of "Subscription comparison procedure" on page 246.

9   Query the rs_subscriptions system table of the current Replication Server for the names of subscriptions and replication definitions or publications and their associated databases.

   •   Also query all Replication Servers with subscriptions to primary data managed by the current Replication Server, or with primary data to which the current Replication Server has subscriptions.

   •   You can query the rs_subscriptions system table by using the rs_helpsub stored procedure.

10 For each user subscription in the rs_subscriptions system table, execute the check subscription command using the information obtained in step 9.

- Execute this command at the current Replication Server and at all Replication Servers with subscriptions to primary data managed by the current Replication Server, or with primary data to which the current Replication Server has subscriptions.

- Subscriptions with a status other than VALID must be deleted or re-created, as described below.

11 For each Replication Server that has a non-VALID subscription with the current Replication Server as the primary:

- Note its subid, and delete the appropriate row from the primary rs_subscriptions system table.

- Use the subid from rs_subscriptions to find corresponding rows in the rs_rules system table, and also delete those rows.

For each system table, rs_subscriptions and rs_rules:

- If a subscription is in the primary table and not in the replicate table (because it was dropped), delete the subscription row from the primary table.

- If a subscription is in the replicate table and not in the primary table, delete the subscription row from the replicate table. After completing the rest of this procedure, re-create the subscription, as described in steps 17 through 19.

- If a subscription is in both the primary and replicate tables but is not VALID at one of the sites, delete the rows from both tables. After completing the rest of this procedure, re-create the subscription, as described in steps 17 through 19.

12 For each primary Replication Server for which the current Replication Server has a non-VALID user subscription:

- Note its subid, and delete the appropriate row from the primary rs_subscriptions system table.

- Use the subid from rs_subscriptions to find corresponding rows in the rs_rules system table, and also delete those rows.

For each system table, rs_subscriptions and rs_rules:

- If a subscription is in the primary table and not in the replicate table, delete the subscription row from the primary table. After completing the rest of this procedure, re-create the subscription, as described in steps 17 through 19.

- If a subscription is in the replicate table and not in the primary table (because it was dropped), delete the subscription row from the replicate table.

- If a subscription is in both the primary and replicate tables, but not VALID at one of the sites, delete the rows from both tables. After completing the rest of this procedure, re-create the subscription, as described in steps 17 through 19.

13 At both the primary and replicate Replication Server, execute the sysadmin drop_queue command for all existing materialization queues for subscriptions deleted in steps 17 through 19.

14 Restart in normal mode all Replication Servers, and their RepAgents, that had subscriptions to primary data managed by the current Replication Server or with primary data to which the current Replication Server had subscriptions.

15 Perform steps 5 through 13 of "Basic RSSD recovery procedure" on page 243.

16 Reexecute any DDL commands that executed at the current Replication Server since the last transaction dump.

17 Enable autocorrection for each replication definition.

18 Re-create the missing subscriptions using either the bulk materialization method or no materialization.

Use the define subscription, activate subscription, validate subscription, and check subscription commands for bulk materialization.

19 For each re-created subscription, restore consistency between the primary and replicate data in either of two ways:

- Drop a subscription using the drop subscription command and the with purge option. Then re-create the subscription.

- Use the rs_subcmp program with the -r flag to reconcile replicate and primary subscription data.

Refer to Chapter 7, "Executable Programs," in the *Replication Server Reference Manual* for more information on the rs_subcmp program. Refer to Chapter 8, "Replication Server System Tables," in the *Replication Server Reference Manual* for more information on the RSSD system tables.

Deintegration/reintegration procedure

If you created routes since the last time the RSSD was dumped, you are required to perform the following tasks:

1   Remove the current Replication Server from the replication system.

    See "Removing a Replication Server" on page 105 in the *Replication Server Administration Guide Volume 1* for details.

2   Reinstall the Replication Server.

    Refer to the Replication Server installation and configuration guides for your platform for complete information on re-installing Replication Server.

3   Re-create Replication Server routes and subscriptions.

    See Chapter 6, "Managing Routes" and Chapter 11, "Managing Subscriptions" in the *Replication Server Administration Guide Volume 1* for details.

Recovery support tasks

This section describes standard recovery tasks that are required in performing the recovery procedures described in this chapter. Use recovery tasks only for the procedure to which they apply. These tasks support recovery by letting you manipulate and identify critical data in the replication system.

Refer to this section for background in performing the recovery procedures in this chapter.

Table 7-5 lists the recovery support tasks.

Table 7-5: Overview of recovery support tasks

Recovery support task	See
Rebuild stable queues	"Rebuilding stable queues" on page 257
Check for Replication Server loss detection messages after rebuilding stable queues	"Loss detection after rebuilding stable queues" on page 260
Put Replication Server in log recovery mode	"Setting log recovery for databases" on page 264
Check for Replication Server loss detection messages after setting log recovery for databases	"Loss detection after setting log recovery" on page 265
Determine which dumps and logs to load	"Determining which dumps to load" on page 266
Adjust database generation numbers	"Adjusting database generation numbers" on page 267

Rebuilding stable queues

The rebuild queues command removes all existing queues and rebuilds them. It cannot rebuild individual stable queues.

You can rebuild queues online or off-line, depending on your situation. Generally, you rebuild queues online first to detect if there are lost stable queue messages. If there are lost messages, you can retrieve them by first putting the Replication Server in standalone mode and recovering the data from an off-line log.

Both methods for rebuilding queues are described in more detail in the following sections. Refer to Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about rebuild queues command.

Rebuilding queues online

During the online rebuild process, the Replication Server is in normal mode. All RepAgents and other Replication Servers are automatically disconnected from the Replication Server. Connection attempts are rejected with the following message:

```
Replication Server is Rebuilding
```

Replication Servers and RepAgents retry connections periodically until rebuild queues has completed. At this time, the connections are successful.

When the queues are cleared, the rebuild is complete. The Replication Server then attempts to retrieve the cleared messages from the following sources:

• Other Replication Servers that have direct routes to the rebuilt Replication Server. If you have set a save interval from other Replication Servers, you have a greater likelihood of recovery.

• Database transaction logs for primary databases the Replication Server manages.

If there are loss detection messages, you need to check the status of these messages. Depending on the failure condition, if these messages are no longer available at their source, you may need to rebuild the queues using off-line logs. Or, you can request that Replication Server ignore the lost messages. See "Rebuilding queues from off-line database logs" on page 258 and "Loss detection after rebuilding stable queues" on page 260.

Rebuilding queues from off-line database logs

This task is used to recover data from off-line database logs. You use the rebuild queues command only after you have restarted the Replication Server in standalone mode. For details on standalone mode, see "Using standalone mode" on page 259. Executing rebuild queues in standalone mode puts Replication Server in recovery mode.

In recovery mode, the Replication Server allows only RepAgents in recovery mode to connect. If a RepAgent that is not in recovery mode attempts to connect, Replication Server rejects it with following error message:

```
Rep Agent not in recovery mode
```

If you use a script that automatically restarts RepAgent and connects it to the Replication Server, you must start RepAgent using the for_recovery option. RepAgents are not allowed to connect in normal mode.

Figure 7-2 illustrates the progression from normal mode to standalone mode to recovery mode using the rebuild queues command.

Figure 7-2: Entering recovery mode with the rebuild queues command
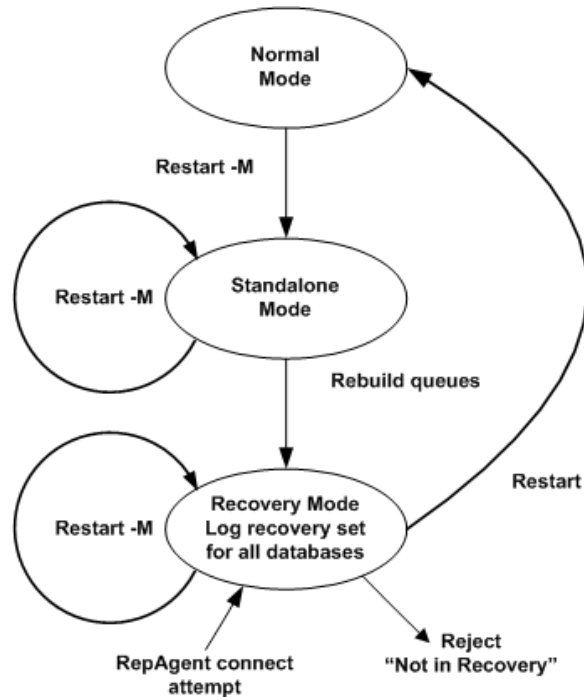
Using standalone mode

To start Replication Server in standalone mode, use the -M flag. Standalone mode is useful for looking at the state of Replication Server because the state is static. Standalone mode allows you to review the contents of the stable queues because no messages are being written to or read from the queues.

Standalone mode differs from the Replication Server normal mode in the following ways:

*   No incoming connections are accepted. If any RepAgent or Replication Server attempts to connect to a Replication Server in standalone mode, the message "Replication Server is in Standalone Mode" is raised.

*   No outgoing connections are started. A Replication Server in standalone mode does not attempt to connect to other Replication Servers.

*   No DSI threads are started, even if there are messages in the DSI queues that have not been applied.

- No Distributor (DIST) threads are started. A DIST thread reads messages from the inbound queues, performs subscription resolution, and writes messages to the outbound queues.

Loss detection after rebuilding stable queues

To determine if any messages could not be recovered after the stable queues were rebuilt, the Replication Server performs loss detection. By checking Replication Server loss-detection messages, you can determine what kind of user intervention, if any, is necessary to restore all data to the system.

Replication Server detects two types of losses after rebuilding stable queues:

- SQM loss, which refers to data lost between two Replication Servers, detected at the next *downstream* site

- DSI loss, which refers to data lost between a Replication Server and a replicate database that the Replication Server manages

Both kinds of loss detection are addressed in the following sections.

If all data is available, no intervention is necessary and the replication system can return to normal operations. For example, if you know that the save interval for the route or connection is set for a longer length of time than the failure, you can recover all messages with no intervention. However, if the save interval is not set or is set too low, some messages may be lost.

Note A Replication Server that has detected a loss does not accept messages from the source. Loss detection prevents the source from truncating its stable queues. For example, if Replication Server RS2 detects that replicate data server DS2.RDB has lost data from primary data server DS1.PDB, Replication Server RS1 cannot truncate its queues until you decide how to handle the loss.

As a result, RS1 may run out of stable storage. Before a loss is detected (that is, after the "Checking Loss" message is reported), you can choose to ignore losses for a source and destination pair.

SQM loss between two Replication Servers

Every time you rebuild stable queues during a recovery procedure, Replication Server requests backlogged messages from sites that send its distributions. If the Replication Server manages primary databases, it instructs their RepAgents to send messages from the beginning of the online transaction logs. The backlogged messages repopulate the emptied stable queues.

Replication Server enables loss detection mode at those sites you are rebuilding that have a direct route from the Replication Server. In Figure 7-3, Replication Server RS3 detects losses if you rebuild the queues of Replication Server RS2. Similarly, RS2 detects losses if you rebuild the queues of Replication Server RS1.

Figure 7-3: Replication system loss detection example
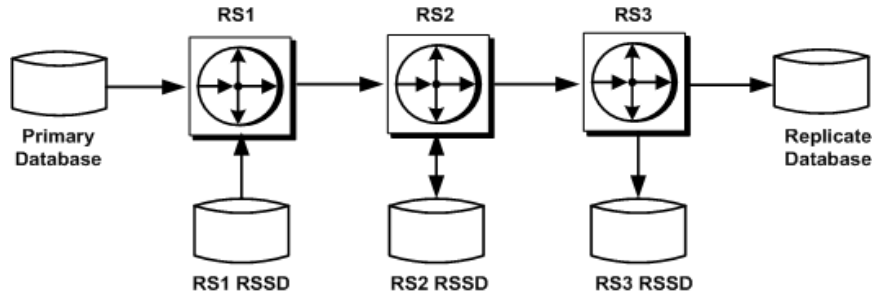
When you execute the rebuild queues command at RS2, RS3 performs loss detection for all primary databases whose updates are routed to RS3 through RS2. RS3 logs messages for each of these databases. If you rebuild queues at RS3, no SQM loss detection is performed, because there are no routes originating from RS3.

Replication Server detects loss by looking for duplicate messages. If RS3 receives a message that it had received before the rebuild queues command, then no messages were lost. If the first message RS3 receives after rebuild queues has not been seen before, then either messages were lost, or no messages were in the stable queue.

Even if there are no messages in the stable queue from a specific source, RS3 identifies them as lost because it has no duplicate messages to use for a comparison. You can prevent this false loss detection by creating a heartbeat with an interval that is less than the save interval. This guarantees that there will always be at least one message in the stable queue.

SQM example

When RS3 performs SQM loss detection for the rebuilt RS2, it logs in to its log file messages similar to the following "Checking Loss" message examples. These messages mark the beginning of the loss detection process. Subsequent messages are logged with the results. Each message contains a source and destination pair.

The first example message indicates that RS3 is checking loss for the RSSD at RS3 from the RSSD at RS2:

```
Checking Loss for DS3.RS3_RSSD from DS2.RS2_RSSD
date=Nov-01-95 10:15 am
qid=0x01234567890123456789
```

The second example message indicates that RS3 is checking loss for the replicate database RDB at RS3, from the primary database PDB at RS1:

```
Checking Loss for DS3.RDB from DS1.PDB
date=Nov-01-95 11:00am
qid=0x01234567890123456789
```

The third example message indicates that RS3 is checking loss for the RSSD at RS3 from the RSSD at RS1:

```
Checking Loss for DS3.RS3_RSSD from DS1.RS1_RSSD
date=Nov-01-95 10:00am
qid=0x01234567890123456789
```

RS3 reports whether it detects a loss. For example, the results of such loss-detection tests might read as follows:

```
No Loss for DS3.RS3_RSSD from DS2.RS2_RSSD
Loss Detected for DS3.RDB from DS1.PDB
No Loss for DS3.RS3_RSSD from DS1.RS1_RSSD
```

DSI loss between a Replication Server and its databases

Some messages in Replication Server queues are destined for databases, rather than for other Replication Servers. The DSI performs loss detection in a way that is similar to stable queue loss detection.

If you rebuild queues at a Replication Server that has no originating routes, no SQM loss detection is performed, but the Replication Server performs DSI loss detection for its messages.

DSI example

The DSI at Replication Server RS2 generates the following message for the RSSD at RS2:

```
DSI: detecting loss for database DS2.RS2_RSSD from
origin DS1.RS1_RSSD
date=Nov-01-95 10:58pm
qid=0x01234567890123456789
```

When retained messages begin arriving from previous sites, the DSI detects a loss, depending on whether the first message from the origin has already been seen by the DSI. If it detects no loss, a message similar to the following one is generated:

```
DSI: no loss for database DS2.RS2_RSSD from origin
DS1.RS1_RSSD
```

If the DSI does detect a loss, a message like the following one is generated:

```
DSI: loss detected for database DS2.RS2_RSSD from origin
DS1.RS1_RSSD
```

Handling losses

When Replication Server detects a loss, no further messages are accepted on the connection to the SQM or the DSI.

For example, when RS3 detects an SQM message loss for the RDB database from the PDB database, it rejects all subsequent messages from the PDB database to the RDB database.

Recovering a loss   To recover the loss, you need to choose one of the following options:

•   Ignore the loss and continue, even though some messages may be lost. You can use the rs_subcmp program with the -r flag to reconcile primary and replicate data.

     To run rs_subcmp, see "Subscription comparison procedure" on page 246. See also Chapter 11, "Managing Subscriptions" in the *Replication Server Administration Guide Volume 1*. Also, refer to Chapter 7, "Executable Programs," in the *Replication Server Reference Manual* for more information about rs_subcmp command.

•   Ignore the loss, then drop and re-create the subscriptions.

•   Recover by replaying transactions from off-line logs (primary Replication Server loss only). In this case, you are not ignoring the loss.

Ignoring a loss   You must execute an ignore loss command in the following situations:

•   If you choose to recover the lost messages by re-creating subscriptions or replaying logs.

•   For an SQM loss, at the Replication Server that reported that loss, to force the Replication Server to begin accepting messages again. For example, to ignore a loss RS3 detected from DS1.PDB, enter the following command at RS3:

     ```
     ignore loss from DS1.PDB to DS3.RDB
     ```

•   For a DSI loss, at the database on the Replication Server where the loss was detected. For example, to ignore a loss reported in DS2.RS2_RSSD from origin DS1.RS1_RSSD, enter the following command at RS2:

     ```
     ignore loss from DS1.RS1_RSSD to DS2.RS2_RSSD
     ```

- For both an SQM and a DSI loss that is detected by a Replication Server at the destination of the route when you rebuild two Replication Servers in succession.

  In this case, you need to execute ignore loss twice, once for SQM losses and once for DSI losses. The ignore loss command that you execute to ignore DSI loss at the destination Replication Server is the same command you use to ignore SQM loss from the previous site.

Setting log recovery for databases

Setting log recovery manually is part of the procedure for recovering from truncated primary database logs off-line or restoring primary and replicate databases from dumps. While the procedure to rebuild queues off-line automatically sets log recovery for all databases, setting log recovery manually allows you to recover each database without reconstructing the stable queue.

The set log recovery command places Replication Server in log recovery mode for a database. You execute this command after placing Replication Server in standalone mode. To connect the RepAgents only to those databases that have been set for log recovery mode, execute the allow connections command. This puts the Replication Server in recovery mode.

Figure 7-4 illustrates the progression from normal mode to standalone mode to recovery mode using the set log recovery and allow connections commands.

For databases specified with the set log recovery command, Replication Server only accepts connections from other Replication Servers and from RepAgents that are in recovery mode. You then recover the transaction dumps into a temporary recovery database.

Figure 7-4: Entering recovery mode with the allow connections command
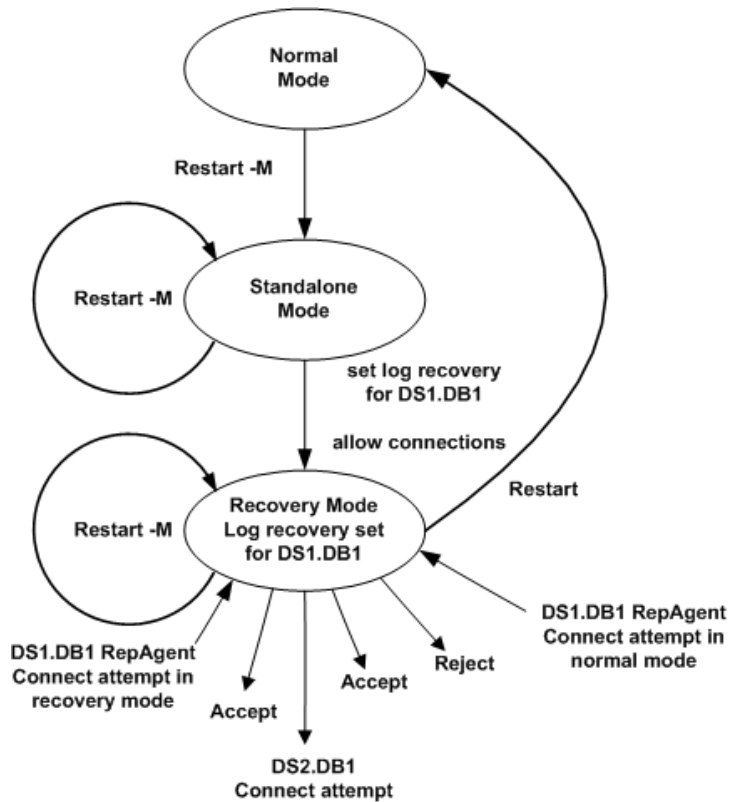
Loss detection after setting log recovery

While you are applying the temporary recovery database to the primary database, Replication Server may detect SQM loss between a primary database and the Replication Server that manages that primary database.

If all data is available, no intervention is necessary and the replication system can return to normal operations. The Replication Server logs a message such as:

```
No Loss Detected for DS1.PDB from DS1.PDB
```

If there were not enough messages, Replication Server logs a loss detection message similar such as:

```
Loss Detected for DS1.PDB from DS1.PDB
```

You must decide whether to ignore the losses by executing the ignore loss command, or repeat the recovery procedure from the beginning. To ignore the loss, enter the following command at the primary Replication Server:

```
ignore loss for DS1.PDB from DS1.PDB
```

If you received loss detection messages, you failed to reload the database to a state old enough to retrieve all of the messages. See "Determining which dumps to load" on page 266.

Determining which dumps to load

When loading transaction log dumps, always examine the "Checking Loss" message that is displayed during loss detection. If there is more than one message, choose the earliest date and time to determine which dumps to load.

For example, if the following message is generated by a Replication Server, you would load the dumps taken just before November 1, 1995 at 10:58 p.m.:

```
Checking Loss for DS3.RDB from DS1.PDB
date=Nov-01-1995 10:58pm
qid=0x01234567890123456789
```

The date in the message is the date and time of the oldest open transaction in the log when the last message received by the Replication Server was generated by the origin queue. Locate the most recent transaction dump with a timestamp before the date and time in the message. Then find the full database dump taken before that transaction dump.

The origin queue ID, or qid, is formed by the RepAgent and identifies a log record in the transaction log. The date is embedded in the qid as a timestamp. Replication Server converts the timestamp to a date for RepAgents for Adaptive Server.

Replication agents for non-Sybase data servers may also embed the timestamp in the qid. Replication Server converts the timestamp for non-Sybase data servers in bytes 20–27. The use of these bytes depends on the Replication Agent.

Note If the data server is not an Adaptive Server, the date in the message may appear nonsensical. You may need to decode the qid in bytes 20–27 to identify the dumps to load.

Adjusting database generation numbers

Each primary database in a replication system includes a database generation number. This number is stored both in the database and in the RSSD of the Replication Server that manages the database.

Any time you load a database for recovery, you may be required to change the database generation number, as instructed in the recovery procedure you are using. This section explains this step.

Determining database generation numbers

RepAgent for a primary database places the database generation number in the high-order 2 bytes of the qid that it constructs for each log record it passes to the Replication Server.

The remainder of the qid is constructed from other information that gives the location of the record in the log and also ensures that the qid increases for each record passed to Replication Server.

The requirement for increasing qid values allows Replication Server to detect duplicate records. For example, when a RepAgent restarts, it may resend some log records that Replication Server has already processed. If Replication Server receives a record with a lower qid than the last record it processed, it treats the record as a duplicate and ignores it.

If you are restoring a primary database to an earlier state, increment the database generation number so that the Replication Server does not ignore log records submitted after the database is reloaded. This step applies only if you are using the procedures described in "Loading a primary database from dumps" on page 240 or in "Loading from coordinated dumps" on page 239.

If you are replaying log records, increment the database generation number only if RepAgent previously sent the reloaded log records with the higher generation number. This situation arises only if you have to restore the database and log to a previous state for the first failure and then later replay the log due to a second failure.

 Warning! Only change the database generation number as part of a recovery procedure. Changing the number at any other time can result in duplicate or missing data at replicate databases.

Dumps and database generation numbers

When you reload a database dump, the database generation number is included in the restored database. Since the database generation number is also stored in the RSSD of the Replication Server that manages the database, you may need to update that number so that it matches the one in the restored database.

However, when you reload a transaction log, the database generation number is not included in the restored log. For example, assume the following operations have occurred in a database:

Table 7-6: Dumps and database generation numbers

Operation	Database generation number
database dump D1	100
transaction dump T1	100
dbcc settrunc('ltm', 'gen_id', 101)	101
transaction dump T2	101
database dump D2	101

If you reload database dump D1, database generation number 100 is restored with it. If you reload transaction dump T1, the generation number remains at 100. After transaction dump T2, the generation number remains at 100, because reloading transaction dumps does not alter the database generation number. In this case, you need to change the database generation number to 101 using the dbcc settrunc command before having RepAgent scan transaction dump T2.

However, if you load database dump D2 before resuming replication, you do not have to alter the database generation number, since the number 101 is restored.

APPENDIX A   **Asynchronous Procedures**

This appendix describes asynchronous stored procedures.

This appendix describes the method for replicating stored procedures that are associated with *table replication definitions*. This method is supported for applications that require it.

See Chapter 10, "Managing Replicated Functions" in the *Replication Server Administration Guide Volume 1* for information about replicated stored procedures that are associated with *function replication definitions*. The method described in that chapter is the recommended method for replicating stored procedures.

Refer to *Replication Server Design Guide* for more information on replication system design issues relating to replicated stored procedures.

Overview

Asynchronous procedure delivery allows you to execute SQL stored procedures that are designated for replication at primary or replicate databases. Because these stored procedures are marked for replication using the sp_setreplicate or sp_setrepproc system procedures, they are called replicated stored procedures.

To satisfy the requirements of distributed applications, Replication Server provides two types of asynchronous stored procedure delivery: applied stored procedures and request stored procedures. Each type is described in this appendix.

Logging replicated stored procedures

Adaptive Server uses the following method to determine in which database a replicated stored procedure execution will be logged:

The procedure gets logged in the database in which the enclosing transaction was started.

- If the user does not begin a transaction explicitly, Adaptive Server will begin one in the user's current database before the stored procedure execution.

- If the user begins the transaction in one database, and then executes a replicated stored procedure in another database, the execution will still be logged in the database where the user began the transaction.

If the execution of a table-style replicated stored procedure (marked for replication by using either sp_setreplicate*proc_name*, 'true' or sp_setrepproc*proc_name*, 'table') is logged in one database and changes replicated tables in another database, the table's changes and the procedure execution are logged in different databases. Therefore, the effects of the stored procedure execution can be replicated twice. The first time the stored procedure execution itself is replicated. The second time table changes that have been logged in the other database are replicated.

Logging replicated stored restrictions

Note that replicated Adaptive Server stored procedures may not contain parameters with the text, unitext, or image datatypes. Refer to the *Adaptive Server Reference Manual* for more information.

Mixed-mode transactions

If a single transaction that invokes one or more request stored procedures is a mixed-mode transaction that also executes applied stored procedures or contains data modification language, Replication Server processes the request stored procedures after all the other operations. All request operations are processed together in a single separate transaction. This situation may arise where a single Replication Server manages both primary and replicate data.

Applied stored procedures

Replicated stored procedures that Replication Server delivers from a primary database to a replicate database are called applied stored procedures.

You use applied stored procedure delivery to replicate transactions first performed on primary data to replicate databases. Data changes are applied at a primary database and then distributed at a later time to replicate databases that subscribe to replication definitions for the data. Replication Server executes the replicated stored procedure in the replicate database as the maintenance user, which is consistent with normal data replication.

You can use applied stored procedures to realize important performance benefits. For example, if your organization has a large amount of row changes, you can create an applied stored procedure which changes many rows, rather than replicating the rows individually. You can also use applied stored procedures to replicate data set changes which are difficult to express using normal subscriptions. Refer to the *Replication Server Design Guide* for more information.

You set up applied stored procedures by making the first statement in the stored procedure update a table. You must also make sure that the destination databases have subscriptions to the before and after images of that updated row. The applied stored procedure must update only one row in a replicated table. Replication Server uses the first row updated by the stored procedure to determine where to send the user-defined function for the procedure.

If the rules in setting up the applied stored procedure are not met, Replication Server fails to distribute the stored procedure to replicate databases. See "Warning conditions" on page 276 for a list of actions that Replication Server takes if it fails to deliver the applied stored procedure.

Request stored procedures

Replicated stored procedures that Replication Server delivers from a replicate database to a primary database are called request stored procedures. You use a request stored procedure to deliver a transaction from a replicate database back to the primary database.

For example, a client application at a remote location may need to make changes to primary data. In this case, the application at the remote location executes a request stored procedure locally to change the primary data. Replication Server delivers this request stored procedure to the primary database by executing, in the replicate database, a stored procedure that has the same name as the stored procedure in the primary database. The stored procedure in the primary database updates the primary data that the transaction changes.

Replication Server executes the replicated stored procedure in the primary database as the user who executed the stored procedure in the replicate database. This ensures that only authorized users may change primary data.

In an application, Replication Server may replicate some or all of the data that is changed in the primary database. The changes are propagated to replicate databases managed by Replication Servers with subscriptions for the related data, either as data rows (insert, delete, or update operation) or as stored procedures. Using this mechanism, the effect of a transaction quickly arrives at both the primary and replicate databases.

Warning! Do not execute a request stored procedure in a primary database. This can lead to looping behavior, in which replicate Replication Servers cause the same procedure to execute in the primary database.

Using request stored procedures ensures that all updates are made at the primary database, preserving the Replication Server basic primary copy data model while keeping the replication system invulnerable to network failures and excess traffic. Even when there is primary database failure, or network failure from the replicate database to the primary database, Replication Server remains fault tolerant. It queues any undelivered request stored procedure invocations until the failed components come back online. When the components are again in service, Replication Server completes delivery.

By using the Replication Server guaranteed request stored procedure delivery feature, you can obtain all the benefits of having a single, definitive copy of your data that includes all the latest changes. At the same time, Replication Server provides the availability and performance benefits of de-coupling applications at replicate databases from the primary database.

Refer to the *Replication Server Design Guide* for more information on replication system design issues relating to asynchronous procedure delivery.

Asynchronous stored procedure prerequisites

Before implementing applied or request stored procedures on your system, be sure you:

*   Understand how you will use asynchronous procedure delivery to meet your application needs. Refer to the *Replication Server Design Guide* for more information.

*   Set up a RepAgent for the stored procedure, even if the database contains no primary data (such as when using request functions). Refer to the Replication Server installation and configuration guides for your platform for details.

*   Create a function string for user-defined functions for function-string classes for which Replication Server does not generate default function strings. You can use the alter function string command to replace a default function string with one that performs the action your application requires.

    See "Function strings and function-string classes" on page 33 for more information.

*   Follow the step-by step instructions provided in this chapter for setting up applied or request stored procedures.

Note For function-string classes for which default generated function strings are provided, Replication Server creates a default function string that executes a stored procedure with the same name as the user-defined function. The procedures in this chapter assume that Replication Server processes applied or request stored procedures for such classes. For all other classes, you must create function strings for the user-defined function string.

Steps for implementing an applied stored procedure

To implement an applied stored procedure, perform the following steps:

1   Review the requirements described in "Asynchronous stored procedure prerequisites" on page 273.

2   Set up replicate databases that contain replicate tables. These tables may or may not match the replication definition for the primary table.

3   As necessary, set up routes from the primary Replication Server to the replicate Replication Servers that have subscriptions to replication definitions for the primary table.

See Chapter 6, "Managing Routes" in the *Replication Server Administration Guide Volume 1* for details on setting up routes.

4   Locate or create a replication definition on the primary Replication Server that identifies the table to be modified.

See Chapter 9, "Managing Replicated Tables" in the *Replication Server Administration Guide Volume 1* for information on creating replication definitions.

5   In the primary database, use the sp_setreplicate system procedure or the sp_setreptable system procedure to mark the table for replication. For example, for a table named employee:

```
sp_setreplicate employee, 'true'
```

or

```
sp_setreptable employee, 'true'
```

For sp_setreptable, the single quotes are optional.

See "Specifying stored procedures and tables for replication" on page 280 for details on using sp_setreplicate. See "Using the sp_setreptable system procedure" on page 270 in the *Replication Server Administration Guide Volume 1* for details on using sp_setreptable.

6   Create the stored procedure on the primary database. The first statement in the stored procedure must contain an update command for the first row of the primary table. For example:

```
create proc upd_emp
 @emp_id int, @salary float
 as
 update employee
 set salary = salary * @salary
```

```
where emp_id = @emp_id
```

> **Warning!** If the first statement in the stored procedure contains an operation other than update, Replication Server cannot distribute the stored procedure to replicate databases. See "Warning conditions" on page 276 for more information.
> Never include dump transaction or dump database commands in the stored procedure. If the stored procedure contains commands with statement level errors, the error may occur at the replicate DSI. Depending on the error actions, the DSI may shut down.

7   In the primary database, use the sp_setreplicate system procedure or the sp_setrepproc system procedure to mark the stored procedure for replication. For example:

```
sp_setreplicate upd_emp, 'true'
```

or

```
sp_setrepproc upd_emp, 'table'
```

See "Specifying stored procedures and tables for replication" on page 280 for details on using sp_setreplicate. See "Marking stored procedures for replication" on page 344 in the *Replication Server Administration Guide Volume 1* for details on using sp_setrepproc.

8   At the replicate Replication Servers, create subscriptions to a replication definition for the table that the stored procedure at the primary database updates.

See Chapter 11, "Managing Subscriptions" in the *Replication Server Administration Guide Volume 1* for details on creating subscriptions.

> **Warning!** Be sure the replicate database subscribes to both the before image and after image of the updated row. If it does not, Replication Server cannot distribute the stored procedure to the replicate database. See "Warning conditions" on page 276 for more information.

9   Create a stored procedure on the replicate database with the same name and parameters as the stored procedure on the primary database, but do not mark the procedure as replicated. For example:

```
create proc upd_emp
 @emp_id int, @salary float
 as
 update employee
```

```
set salary = salary * @salary
where emp_id = @emp_id
```

10   Grant execute permission on the stored procedure to the maintenance user.
      For example:

```
grant execute on upd_emp to maint_user
```

11   Create a user-defined function on the primary Replication Server that
      associates the stored procedure to the name of a replication definition for
      the table it updates. For example:

```
create function employee_rep.upd_emp
  (@emp_id int, @salary float)
```

Only one user-defined function are shared by all replication definitions for
the same table. You can specify the name of any of these replication
definitions.

12   Verify that all Replication Server and database objects in steps 1 through
      11 exist at the appropriate locations.

Refer to Chapter 6, "Adaptive Server Stored Procedures," in the
Replication Server Reference Manual for information about stored
procedures used to query the RSSD for system information.

Warning conditions

If the first statement in the applied stored procedure is an operation other than
update, or the replicate database does not subscribe to the before image and
after image of the updated row, Replication Server fails to deliver the applied
stored procedure to the replicate database. Instead, Replication Server
performs other actions that you can interpret as warnings.

The actions Replication Server takes is based on:

•   The first operation (other than update) contained in the applied stored
    procedure at the primary database

•   Whether the row modification stays in the subscription for the replicate
    database, and whether it matches the subscription's before image or after
    image

Conditions and
actions

This section identifies the warning conditions that prevent Replication Server
from delivering an applied stored procedure at a replicate database.

Condition: The first row operation is an insert operation.

Action: Replication Server distributes the insert operation instead of the applied stored procedure.

Condition: The first row operation is a delete operation.

Action: Replication Server distributes the delete operation instead of the applied stored procedure.

Condition: Replicate Replication Servers have subscriptions that match the before image, but not the after image, of the modified row.

Action: Replication Server distributes a delete operation (rs_delete system function) to replicate databases with subscriptions to the before image but not the after image of the row modification.

Example: Assume there is a table T1 that has a column named C1 with a value of 1. A replicate database has a subscription to a replication definition for table T1 where C1 = 1.

If the associated stored procedure is executed with the parameters= 1 (before image) and = 2 (after image), the replicate database does not subscribe to the after image value of 2. Therefore, Replication Server distributes the delete operation to the replicate database.

Condition: Replicate Replication Servers have subscriptions that match the after image, but not the before image of the modified row.

Action: Replication Server distributes an insert operation (rs_insert system function) to replicate databases with subscriptions to the after image but not the before image of the row modification.

Example: Assume there is a table T1 that has a column named C1 with a value of 1. A replicate database has a subscription to a replication definition for table T1 where C1 = 2.

If the associated stored procedure is executed with the parameters = 1 (before image) and = 2 (after image), the replicate database does not subscribe to the before image value of 1. Therefore, Replication Server distributes the insert operation to the replicate database.

Condition: Replicate Replication Servers have subscriptions that match neither the before image nor the after image of the row modification.

Action: Replication Server does not distribute any operation or stored procedure to the replicate databases.

Example: Assume there is a table T1 that has a column named C1 with a value of 1. A replicate database has a subscription to a replication definition for table T1 where C1 > 2.

If the associated stored procedure is executed with the parameters equal to 1 (before image) and equal to 2 (after image), the replicate Replication Server does not subscribe to either the before image value of 1 or the after image value of 2. Therefore, Replication Server performs no distribution to the replicate database.

Steps for implementing a request stored procedure

To implement a request stored procedure, perform the following steps:

1   Review the requirements described in "Asynchronous stored procedure prerequisites" on page 273.

2   As necessary, set up a route from the replicate Replication Server to the primary Replication Server where the data is updated, and from the primary Replication Server to the replicate Replication Server that sends the update.

See Chapter 6, "Managing Routes" in the *Replication Server Administration Guide Volume 1* for details on setting up routes.

3   Create a login name and password at the primary Replication Server for the user at the replicate Replication Server.

See Chapter 8, "Managing Replication Server Security" in the *Replication Server Administration Guide Volume 1* for details.

4   At the replicate Replication Server, create the necessary permissions for this user to execute the stored procedure at the primary Replication Server.

See Chapter 8, "Managing Replication Server Security" in the *Replication Server Administration Guide Volume 1* for details.

5   At the primary Replication Server, locate or create a replication definition that identifies the table to be modified.

See Chapter 9, "Managing Replicated Tables" in the *Replication Server Administration Guide Volume 1* for information on creating replication definitions.

The replicate Replication Server may have subscriptions on the replication definition.

6   Create the stored procedure, which does not perform any updates, on the replicate database. For example:

```
create proc upd_emp
 @emp_id int, @salary float
 as
 print "Transaction accepted."
```

If you want the stored procedure to have the same name as those in different replicate databases, see "Specifying a nonunique name for a user-defined function" on page 285 for details.

7   In the replicate database, use the sp_setreplicate system procedure or the sp_setrepproc system procedure to mark the stored procedure for replication. For example:

```
sp_setreplicate upd_emp, 'true'
```

or

```
sp_setrepproc upd_emp, 'table'
```

See "Specifying stored procedures and tables for replication" on page 280 for details on using sp_setreplicate. See "Marking stored procedures for replication" on page 344 in the *Replication Server Administration Guide Volume 1* for details on using sp_setrepproc.

8   Create a stored procedure on the primary database with the same name as the stored procedure on the replicate database, but do not mark the procedure as replicated. This stored procedure modifies a primary table. For example:

```
create proc upd_emp
 @emp_id int, @salary float
 as
 update employee
 set salary = salary * @salary
 where emp_id = @emp_id
```

Note The stored procedure names on the primary and replicate databases can differ if you alter the function string for the function to execute a stored procedure with a different name. See "Mapping to a different stored procedure name" on page 284 for more information.

9   Grant permission on the stored procedure to the replicate Replication Server users who will execute this stored procedure. For example:

```
grant all on upd_emp to public
```

10 Create a user-defined function on the primary Replication Server that associates the stored procedure to the name of a replication definition for the table it updates. For example:

```
create function employee_rep.upd_emp
  (@emp_id int, @salary float)
```

11 Verify that all Replication Server and database objects in steps 1 through 10 exist at the appropriate locations.

Refer to Chapter 6, "Adaptive Server Stored Procedures," in the *Replication Server Reference Manual* for information about stored procedures used to query the RSSD for system information.

Specifying stored procedures and tables for replication

You can use the sp_setreplicate system procedure in Adaptive Server to mark database tables and stored procedures for replication.

You can also use the sp_setreptable system procedure to mark tables for replication and the sp_setrepproc system procedure to mark stored procedures for replication. These system procedures extend the capabilities of sp_setreplicate and are intended to replace it.

See "Using the sp_setreptable system procedure" on page 270 and "Marking stored procedures for replication" on page 344 in the *Replication Server Administration Guide Volume 1* for details.

The syntax for the sp_setreplicate system procedure is:

sp_setreplicate [*object_name* [, {' true' | 'false']]

object_name can be either a table name or a stored procedure name.

The "true" and "false" parameters change the replication status of a specified object. (The single quotes are optional.)

• Use sp_setreplicate with no parameters to list all replicated objects in the database.

• Use sp_setreplicate with just the object name to check the replication status of the object. Adaptive Server reports 'true' if replication is enabled for the object, or 'false' if it is not.

- Use sp_setreplicate with the object name and either 'true' or 'false' to enable or disable replication for the object. You must be the Adaptive Server System Administrator or the Database Owner to use sp_setreplicate to change the replication status of an object.

Warning! A replicated stored procedure should only modify data in the database in which it is executed. If it modifies data in another database, Replication Server replicates the updated data and the stored procedure.

Managing user-defined functions

This section describes commands for managing user-defined functions. See Chapter 8, "Managing Replication Server Security" in the *Replication Server Administration Guide Volume 1* for a list of permissions that are required to use the commands. See Chapter 2, "Customizing Database Operations" for details on altering function strings for user-defined functions and displaying function-related information.

Creating a user-defined function

Use the create function command to register a replicated stored procedure with Replication Server. When a stored procedure is executed, Replication Server maps it to a replication definition. The replication definition contains a user-defined function name that matches the name of the stored procedure.

Replication Server delivers the function to the Replication Server that is primary for the replication definition. When the destination Replication Server that owns the replication definition receives the function, it maps the stored procedure parameters into the commands for the user-defined function.

The syntax for the create function command is:

    create function *replication_definition.function*
    ([@*parameter datatype* [, @*parameter datatype*]...])

The *replication_definition* must be an existing replication definition.

Observe these guidelines when using this command:

- Execute this command at the Replication Server where the replication definition was created.

- Do not use the names of system functions. See Chapter 2, "Customizing Database Operations" for the list of reserved system-function names.

- Include the parentheses surrounding the listed parameters, even when you are defining functions with no parameters.

- If you are not using a function-string class for which default generated function strings are provided, after you have created a user-defined function, use the create function string command to add a function string. See Chapter 2, "Customizing Database Operations" for details.

The following example creates a user-defined function named Stock_receipt. The function is associated with the Items_rd replication definition:

```
create function Items_rd.Stock_receipt
  (@Location int, @Recpt_num int,
  @Item_no char(15), @Qty_recd int)
```

When a user executes the replicated stored procedure, Replication Server now delivers it.

Adding parameters to a user-defined function

When you add a parameter to a replicated stored procedure, use the alter function command to tell Replication Server about the new parameters. To add the parameters:

1 Alter the stored procedure at the primary or replicate data server and provide defaults for new parameters.

2 As a precaution, quiesce the system. Altering functions while updates are in process can have unpredictable results.

See "Quiescing Replication Server" on page 104 in the *Replication Server Administration Guide Volume 1* for details on quiescing the system.

3 Alter the function using the alter function command.

4 If you are not using a function-string class for which default generated function strings are provided, alter function strings to use the new parameters. See Chapter 2, "Customizing Database Operations" for details.

The syntax for the alter function command is:

alter function *replication_definition.function*
add parameters @*parameter datatype*
[, @*parameter datatype*]...

The *replication_definition* is the name of the replication definition for the function. A function can have up to 255 parameters.

The following example adds an int parameter named Volume to the New_issue function for the Tokyo_quotes replication definition:

```
alter function Tokyo_quotes.New_issue
 add parameters @Volume int
```

Dropping a user-defined function

Use the drop function command to drop a user-defined function. This command drops a function name and any function strings that have been created for it. You cannot drop system functions.

Before you drop the user-defined function, be sure to:

1   Drop the stored procedure at the primary database using the drop procedure Adaptive Server command, or use the sp_setreplicate or sp_setrepproc system procedure and specify 'false' to disable replication for the stored procedure.

See "Specifying stored procedures and tables for replication" on page 280 for details on using sp_setreplicate. See "Marking stored procedures for replication" on page 344 in the *Replication Server Administration Guide Volume 1* for details on using sp_setrepproc.

2   As a precaution, quiesce the system before executing the drop function command. Dropping functions while updates are in process can have unpredictable results.

See "Quiescing a replication system" on page 104 in the *Replication Server Administration Guide Volume 1* for details on quiescing the system.

The syntax for the drop function command is:

drop function *replication_definition.function*

Execute the command on the Replication Server where the replication definition was created.

The following command drops the Stock_receipt user-defined function created in the previous section:

```
drop function Items_rd.Stock_receipt
```

Mapping to a different stored procedure name

When you create a user-defined function in a database that uses the a function-string class for which default generated function strings are provided, Replication Server generates a default function string. The default generated function string executes a stored procedure with the same name and parameters as the user-defined function.

For example, if you are using a default function string, you can set up a request stored procedure to execute in the replicate database by creating a stored procedure in the primary database with the same name and parameters as the user-defined function.

If you want to map the user-defined function to a different stored procedure name, use the alter function string command to configure Replication Server to deliver the stored procedure by executing a stored procedure with a different name. You can also do so in function-string classes that allow you to customize function strings.

Example

This example illustrates how to map a user-defined function to a different stored procedure name.

1   Assume the stored procedure upd_sales exists on the primary Adaptive Server, and that it performs an update on the Adaptive Server sales table:

```
create proc upd_sales
 @stor_id varchar(10),
 @ord_num varchar(10),
 @date datetime
 as
64 update sales set date = @date
 where stor_id = @stor_id
 and ord_num = @ord_num
```

2   To register the upd_sales stored procedure with the Replication Server, create the following function, whose name includes in its name the sales_def replication definition on the sales table and the upd_sales replicated stored procedure:

```
create function sales_def.upd_sales
  (@stor_id varchar(10), @date datetime)
```

3   On the replicate Adaptive Server, a version of the stored procedure upd_sales that performs no work is created with the same name:

```
create proc upd_sales
 @stor_id varchar(10),
 @ord_num varchar(10),
 @date datetime
```

```
as
print "Attempting to Update Sales Table"
print "Processing Update Asynchronously"
```

4   To execute the upd_sales stored procedure with the name real_update instead of upd_sales:

   •   The default generated function string is altered:

```
alter function string sales_def.upd_sales
 for rs_sqlserver_function_class
 output rpc
 'execute real_update
 @stor_id = ?stor_id!param?,
 @date = ?date!param?'
```

   •   A stored procedure in the primary database is created with the name real_update. It accepts two parameters.

Specifying a nonunique name for a user-defined function

The name of a user-defined function must be globally unique in the replication system so that Replication Server can locate the particular replication definition for which the user-defined function is defined. If you create more than one replication definition for the same primary table, there is only one user-defined function for all of that table's replication definitions.

If the user-defined function name is not unique, the first parameter of the stored procedure must be @*rs_repdef*, and the name of the replication definition must be passed in this parameter when the stored procedure is executed.

Do not define the @*rs_repdef* parameter in the create function command for the user-defined function. The Replication Agent extracts the replication definition name and sends it with the LTL commands. This convention works with RepAgent for Adaptive Server, but may not be supported by Replication Agents for other data servers.

Example   This example assumes that the user-defined function is not unique and the replication definition name is passed to the @*rs_repdef* parameter when the following stored procedure is executed:

```
create proc upd_sales
 @rs_repdef varchar(255),
 @stor_id varchar(10),
 @date datetime
 as
```

```
print "Attempting to Update Sales Table"
print "Processing Update Asynchronously"
```

High Availability on Sun Cluster 2.2

This appendix provides background and procedures for configuring Sybase Replication Server for high availability (HA) on Sun Cluster 2.2.

Topic	Page
Introduction	287
Terminology	288
Technology overview	289
Configuring Replication Server for high availability	290
Administering Replication Server as a data service	295

Introduction

This appendix assumes that:

*   You are familiar with Sybase Replication Server. This chapter does not explain the steps necessary to install Sybase Replication Server.

*   You are familiar with Sun Cluster HA. This document does not explain the steps necessary to install Sun Cluster HA.

*   You have a two-node cluster hardware with Sun Cluster HA 2.2.

Documentation references:

*   *Sun Cluster 2.2 Software Planning and Installation Guide*

*   *Sun Cluster 2.2 System Administration Guide*

*   Configuring Sybase Adaptive Server Enterprise 12.0 Server for High Availability: Sun Cluster HA (see White Papers at http://www.sybase.com/products/databaseservers/ase)

*   Replication Server documentation (see Product Manuals at http://www.sybase.com/support/manuals/)

Terminology

These terms are used in this chapter:

- Cluster – multiple systems, or nodes, that work together as a single entity to provide applications, system resources, and data to users.

- Cluster node – a physical machine that is part of a Sun Cluster. Also called a physical host.

- Data service – an application that provides client service on a network and implements read and write access to disk-based data. Replication Server and Adaptive Server Enterprise are examples of data services.

- Disk group – a well-defined group of multihost disks that move as a unit between two servers in an HA configuration.

- Fault monitor – a daemon that probes data services.

- High availability (HA) – very low downtime. Computer systems that provide HA usually provide 99.999% availability, or roughly five minutes unscheduled downtime per year.

- Logical host – a group of resources including a disk group, logical host name, and logical IP address. A logical host resides on (or is mastered by) a physical host (or node) in a cluster machine. It can move as a unit between physical hosts on a cluster.

- Master – the node with exclusive read and write access to the disk group that has the logical address mapped to its Ethernet address. The current master of the logical host runs the logical host's data services.

- Multihost disk – a disk configured for potential accessibility from multiple nodes.

- Failover – the event triggered by a node or a data service failure, in which logical hosts and the data services on the logical hosts move to another node.

- Failback – a planned event, where a logical host and its data services are moved back to the original hosts.

Technology overview

Sun Cluster HA is a hardware- and software-based high availability solution. It provides high availability support on a cluster machine and automatic data service failover in just a few seconds. It accomplishes this by adding hardware redundancy, software monitoring, and restart capabilities.

Sun Cluster provides cluster management tools for a System Administrator to configure, maintain, and troubleshoot HA installations.

The Sun Cluster configuration tolerates these single-point failures:

*   Server hardware failure

*   Disk media failure

*   Network interface failure

*   Server OS failure

When any of these failures occur, HA software fails over logical hosts onto another node and restarts data services on the logical host in the new node.

Sybase Replication Server is implemented as a data service on a logical host on the cluster machine. The HA fault monitor for Replication Server periodically probes Replication Server. If Replication Server is down or hung, the fault monitor attempts to restart Replication Server locally. If Replication Server fails again within a configurable period of time, the fault monitor fails over to the logical host so the Replication Server will be rebooted on the second node.

To Replication Server clients, it appears as though the original Replication Server has experienced a reboot. The fact that it has moved to another physical machine is transparent to the users. Replication Server is affiliated with a logical host, not the physical machine.

As a data service, the Replication Server includes a set of scripts registered with Sun Cluster as callback methods. Sun Cluster calls these methods at different stages of Failover:

*   FM_STOP – to shut down the fault monitor for the data service to be failed over.

*   STOP_NET – to shut down the data service itself.

*   START_NET – to start the data service on the new node.

*   FM_START – to start the fault monitor on the new node for the data service.

Each Replication Server is registered as a data service using the hareg command. If you have multiple Replication Servers running on the cluster, you must register each of them. Each data service has its own fault monitor as a separate process.

Note For detailed information about the hareg command, see the appropriate Sun Cluster documentation.

Configuring Replication Server for high availability

This section describes the tasks required to configure a Replication Server for HA on Sun Cluster (assuming a two-node cluster machine).

- "Configuring Sun Cluster for HA" on page 290

- "Installing Replication Server for HA" on page 291

- "Installing Replication Server as a data service" on page 292

Configuring Sun Cluster for HA

The system should have following components:

- Two homogenous Sun Enterprise servers with similar configurations in terms of resources like CPU, memory, and so on. The servers should be configured with cluster interconnect, which is used for maintaining cluster availability, synchronization, and integrity.

- The system should be equipped with a set of multihost disks. The multihost disk holds the data (partitions) for a highly available Replication Server. A node can access data on a multihost disk only when it is a current master of the logical host to which the disk belongs.

- The system should have Sun Cluster HA software installed, with automatic failover capability. The multihost disks should have unique path names across the system.

- For disk failure protection, disk mirroring (not provided by Sybase) should be used.

- Logical hosts should be configured. Replication Server runs on a logical host.

- Make sure the logical host for the Replication Server has enough disk space in its multihosted disk groups for the partitions, and that any potential master for the logical host has enough memory for the Replication Server.

Installing Replication Server for HA

During Replication Server installation, you need to perform these tasks in addition to the tasks described in the Replication Server installation guide:

1   As a Sybase user, load Replication Server either on a shared disk or on the local disk. If it is on a shared disk, the release cannot be accessed from both machines concurrently. If it is on a local disk, make sure the release paths are the same for both machines. If they are not the same, use a symbolic link, so they will be the same. For example, if the release is on */node1/repserver* on node1, and */node2/repserver* on node2, link them to */repserver* on both nodes so the *$SYBASE* environment variable is the same across the system.

2   Add entries for Replication Server, RSSD server, and primary/replicate data servers to the interfaces file in the *$SYBASE* directory on both machines. Use the logical host name for Replication Server in the interfaces file.

> **Note** To use LDAP directory services instead of interfaces files, supply multiple entries in the DIRECTORY section of the Replication Server configuration file. If the connection to the first entry fails, the directory control layer (DCL) attempts to connection to the second entry and so on. If a connection cannot be made to any entry in the DIRECTORY section, Open Client/Server *does not* use the default interfaces file to attempt a connection.
>
> See the configuration guide for your platform for information about setting up LDAP directory services.

3   Start the RSSD server.

4   Follow the installation guide for your platform to install Replication Server on the node that is currently the master in the logical host. Make sure that you:

a Set the environment variables SYBASE, SYBASE_REP, and SYBASE_OCS:

```
setenv SYBASE /REPSERVER1210
setenv SYBASE_REP REP-12_1
setenv SYBASE_OCS OCS-12_0
```

/REPSERVER1210 is the release directory.

b Choose a run directory for the Replication Server that will contain the Replication Server run file, configuration file, and log file. The run directory should exist on both nodes and have exactly the same paths on both nodes (the path can be linked if necessary).

c Choose the multihosted disks for the Replication Server partitions.

d Initiate the rs_init command, from the *run* directory:

```
cd RUN_DIRECTORY
$SYBASE/$SYBASE_REP/install/rs_init
```

5 Make sure that Replication Server is started.

6 As a Sybase user, copy the run file and the configuration file to the other node in the same path. Edit the run file on the second node to make sure it contains the correct path of the configuration and log files, especially if links are used.

Note The run file name must be *RUN_repserver_name*, where *repserver_name* is the name of the Replication Server. You can define the configuration and log file names.

Installing Replication Server as a data service

You also need to perform these specialized tasks to install Replication Server as a data service:

1 As root, create the directory */opt/SUNWcluster/ha/repserver_name* on both cluster nodes, where *repserver_name* is the name of your Replication Server. Each Replication Server must have its own directory with the server name in the path. Copy the following scripts from the Replication Server installation directory *$SYBASE/$SYBASE_REP/sample/ha* to:

/opt/SUNWcluster/ha/repserver_name

on both cluster nodes, where *repserver_name* is the name of your Replication Server:

```
repserver_start_net
repserver_stop_net
repserver_fm_start
repserver_fm_stop
repserver_fm
repserver_shutdown
repserver_notify_admin
```

If the scripts already exist on the local machine as part of another Replication Server data service, you can create the following as a link to the script directory instead:

/opt/SUNWcluster/ha/repserver_name

2   As root, create the directory */var/opt/repserver* on both nodes if it does not exist.

3   As root, create a file */var/opt/repserver/repserver_name* on both nodes for each Replication Server you want to install as a data service on Sun Cluster, where *repserver_name* is the name of your Replication Server. This file should contain only two lines in the following form with no blank space, and should be readable only by root:

> *repserver*:*logicalHost*:*RunFile*:*releaseDir*:*SYBASE_OCS* :*SYBASE_REP*

> *probeCycle*:*probeTimeout*:*restartDelay*:*login/password*

where:

•   *repserver* – the Replication Server name.

•   *logicalHost* – the logical host on which Replication Server runs.

•   *RunFile* – the complete path of the runfile.

•   *releaseDir* – the $SYBASE installation directory.

•   *SYBASE_OCS* – the $SYBASE subdirectory where the connectivity library is located.

•   *SYBASE_REP* – the $SYBASE subdirectory where the Replication Server is located.

•   *probeCycle* – the number of seconds between the start of two probes by the fault monitor.

- *probeTimeout* – time, in seconds, after which a running Replication Server probe is aborted by the fault monitor, and a timeout condition is set.

- *restartDelay* – minimum time, in seconds, between two Replication Server restarts. If, in less than restartDelay seconds after a Replication Server restart, the fault monitor again detects a condition that requires a restart, it triggers a switch over to the other host instead. This resolves situations where a database restart does not solve the problem.

- *login/password* – the login/password the fault monitor uses to ping Replication Server.

To change probeCycle, probeTimeout, restartDelay, or login/password for the probe after Replication Server is installed as data service, send SIGINT(2) to the monitor process (repserver_fm) to refresh its memory.

```
kill -2 monitor_process_id
```

4   As root, create a file */var/opt/repserver/repserver_name.mail* on both nodes, where *repserver_name* is the name of your Replication Server. This file lists the UNIX login names of the Replication Server administrators. The login names should be all in one line, separated by one space.

If the fault monitor encounters any problems that need intervention, this is the list to which it sends mail.

5   Register the Replication Server as a data service on Sun Cluster:

```
hareg -r repserver_name \
-b "/opt/SUNWcluster/ha/repserver_name" \
-m START_NET="/opt/SUNWcluster/ha/repserver_name/
repserver_start_net" \
-t START_NET=60 \
-m STOP_NET="/opt/SUNWcluster/ha/repserver_name/
repserver_stop_net" \
-t STOP_NET=60 \
-m FM_START="/opt/SUNWcluster/ha/repserver_name/
repserver_fm_start" \
-t FM_START=60 \
-m
FM_STOP="/opt/SUNWcluster/ha/repserver_name/repserv
er_fm_stop" \
-t FM_STOP=60 \
[-d sybase] -h logical_host
```

where *-d sybase* is required if the RSSD is under HA on the same cluster, and *repserver_name* is the name of your Replication Server and must be in the path of the scripts.

6   Turn on the data service using `hareg -y` **`repserver_name.`**

Administering Replication Server as a data service

This section describes how to start and shut down Replication Server as a data service, and useful logs for monitoring and troubleshooting.

Data service start/shutdown

Once a Replication Server is registered as data service, use the following to start Replication Server as a data service:

```
hareg -y repserver_name
```

This starts Replication Server if it is not already running, and also starts the fault monitor for Replication Server.

To shut down Replication Server, use:

```
hareg -n repserver_name
```

The fault monitor restarts or fails over this Replication Server if it is shut down or stopped (killed) any other way.

Logs

There are several logs you can use for debugging:

- Replication Server log – the Replication Server logs its messages here. Use the log to find informational and error messages from Replication Server. The log is located in the Replication Server *Run* directory.

- Script log – the data service START and STOP scripts log messages here. Use the log to find informational and error messages that result from running the scripts. The log is located in */var/opt/repserver/harep.log*.

- Console log – the operating system logs messages here. Use this log to find informational and error messages from the hardware. The log is located in */var/adm/messages.*

- CCD log – the Cluster Configurations Database, which is part of the Sun Cluster configuration, logs messages here. Use this log to find informational and error messages about the Sun Cluster configuration and health. The log is located in */var/opt/SUNWcluster/ccd/ccd.log*.

Pre-15.1 Request Function Replication

This appendix describes how you can replicate request functions in Replication Server versions earlier than 15.1. This request function replication lets you deliver a transaction from a replicate database to the primary database without subscriptions.

Topic	Page
Prerequisites and restrictions	297
Using the pre-15.1 request function replication	300
Modifying or dropping replicated functions	306
Model variations and strategies	308

See Chapter 10, "Managing Replicated Functions" in the *Replication Server Administration Guide Volume 1* for more information about the recommended method for replicating stored procedures.

See the *Replication Server Design Guide* for more information on replication system design issues relating to replicated stored procedures.

Prerequisites and restrictions

Before you implement a no-subscription request function:

*   Read this appendix to fully understand how to use the no-subscription request function replication with versions earlier than 15.1 to meet your application needs.

*   Set up a RepAgent for the replicate database from which replicated functions are delivered. See Chapter 4, "Managing a Replication System"and the *Replication Server Configuration Guide* for details.

*   Set up the route from a replicate Replication Server to the primary Replication Server. See Chapter 6, "Managing Routes" in the *Replication Server Administration Guide Volume 1*.

These restrictions apply:

- Only the function replication definition created with create function replication definition command can be used as the no-subscription request function replication with versions earlier than 15.1.

- If you create a new applied or request function replication definition with create applied function replication definition or create request function replication definition command for a primary function, the no-subscription request function replication for the same primary function will be disabled.

- The names of all replication definitions, including function replication definitions, must be unique in the replication system. The function replication definition, created for no-subscription request function must have the same name as the primary function.

  To replicate a version of a request stored procedure earlier than 15.1 with a non-unique name, invoke it with a stored procedure that has a unique name and is executed in the replicate database.

  For example, the non-unique stored procedure upd_sales may invoke the unique stored procedure upd_salesA or vice versa. Mark upd_salesA for replication using the sp_setrepproc system procedure, and leave upd_sales, the stored procedure that invokes it, unmarked.

  Alternatively, you can declare the first parameter of the stored procedure with a non-unique name as @rs_repdef and pass the unique name of the replication definition in this parameter when you run the stored procedure. Do not include the @rs_repdef parameter in the create function replication definition command. This method works only with the RepAgent for Adaptive Server.

- Replication Server does not support nested transactions—those containing begin or commit statements—within replicated stored procedures. If the stored procedures with nested stored procedures are marked for replication:

  - The RepAgent forwards only the outer stored procedure call to the Replication Server.

  - The RepAgent shuts down.

  - An error message appears in the Adaptive Server error log.

If the replicated stored procedure contains nested transaction commands, the Replication Server replicates several transactions together, as one group, using only the first begin transaction and the last commit transaction of all the grouped transactions. For more information, see dsi_max_xacts_in_group, in "Connection parameters that affect performance" on page 137.

If the replicated stored procedure contains commands like begin transaction, commit transaction, or rollback transaction, errors may result when you run the procedure. For example, a rollback transaction command may roll back to the start of the transaction group, rather than to the nested begin transaction command that was the intended rollback point.

- Replicated functions, like Adaptive Server stored procedures, cannot contain parameters with text and image datatypes. See the *Adaptive Server Enterprise Reference Manual*.

- Adaptive Server logs a replicated stored procedure invocation in the database in which the enclosing transaction was started:

  - If you do not begin a transaction explicitly, Adaptive Server begins one in the user's current database before the stored procedure is invoked.

  - If you do not begin the transaction in one database and then run a replicated stored procedure in another database, the execution is still logged in the database where the transaction began.

- If a single transaction invokes one or more request functions and executes applied functions or contains data modification language, or a mixed-mode transaction, Replication Server processes the request functions after all the other operations have completed, together in a separate transaction. This can occur if a single Replication Server manages both the primary and replicate data.

- Replication Server does not perform translations on parameter values for request functions. However, during function-string mapping, the delimiters defined for the parameter values of their declared datatype are used to generate the SQL statements.

- Do not put a commit statement inside a replicated function as this can cause a duplicate key and make Replication Server recovery fail.

Using the pre-15.1 request function replication

A replicated stored procedure is an Adaptive Server stored procedure that you have marked for replication using either sp_setrepproc or sp_setreplicate. A function replication definition describes a replicated stored procedure, its parameters, and its location. When you create a function replication definition, Replication Server creates a function, which contains the information in the function replication definition.

When a replicated stored procedure that has its own function replication definition is invoked, its function is transferred from the source to a destination Replication Server. The function passes parameters to a corresponding stored procedure that is invoked in the destination database. A function string translates the function into a syntax that the subscribing database can interpret.

In versions earlier than 15.1, the no-subscription request function delivers a replicated stored procedure from a replicate database to the primary database. The function replication definition's primary database is defined as primary database, while the database from which the function is originally invoked is defined as a replicate database.

For example, a client application at a remote location needs to make changes to the primary data. The client application first executes a stored procedure, which may or may not make changes at the replicate database, at the remote site. When you run the stored procedure, the replicate Replication Server passes a request function to the primary database, where a corresponding stored procedure is invoked and updates the primary data. No subscription is needed to deliver this request function.

In a primary copy model, a single primary database contains the most recent updates. A client application at a remote replicate site can update the primary database using the request functions. As updates occur at the primary table, the Replication Server captures the updates and sends them to the replicate data servers.

If communication between the primary and replicate databases fails, operations executed in the primary database are stored in the Replication Server stable queues until they can be delivered to the replicate sites. Likewise, operations executed remotely at the replicate database are held in stable queues until they can be delivered to the primary database.

To use a no-subscription request function with versions earlier than 15.1, create a stored procedure in the primary database and a corresponding stored procedure in the replicate database. Then, at the primary Replication Server, create a function replication definition with the same name as the name of the stored procedure created in the replicate database. You do not need to create any subscriptions. When the stored procedure in the replicate database is invoked, it invokes the stored procedure in the primary database.

Replication Server runs the stored procedure in the primary database with the same user that invokes the stored procedure in the replicate database. This guarantees that only authorized users can change the primary data.

In an application, Replication Server may replicate some or all of the data that changes in the primary database. The changes are distributed to the replicate databases, which are managed by Replication Servers that have subscriptions to table replication definitions, or as separate applied functions. Either way, a transaction arrives at the primary database first, and then arrives at the replicate databases.

When you use a no-subscription request function with versions earlier than 15.1, all updates are made at the primary database. This preserves the Replication Server primary copy data model and protects the replication system from network failure and excess traffic.

Commands for managing function replication definition

Table C-1 lists the Replication Server commands that are used to work with the no-subscription request function replication definitions with versions earlier than 15.1.

Table C-1: Commands for managing function replication definitions

Command	Task
create function replication definition	Creates a function replication definition that describes the stored procedure and its parameters for replication. It also describes the location of the primary data. See "Implementing a pre-15.1 request function" on page 302.
alter function replication definition	Modifies a function replication definition. For example, it: • Specifies a different name for the stored procedure invoked at the destination database. • Adds parameters or searchable parameters. • Changes how the replication definition is used in replicating to a standby database. See "Modifying or dropping replicated functions" on page 306".
drop function replication definition	Removes a function replication definition from the replication system. See "Modifying or dropping replicated functions" on page 306.

Implementing a pre-15.1 request function

To implement a request function with versions earlier than 15.1:

1   Review the requirements described in "Prerequisites and restrictions" on page 297.

2   In the primary Adaptive Server, create a login name and password for the user who executes the stored procedure at the replicate Adaptive Server.

See Chapter 8, "Managing Replication Server Security" in the *Replication Server Administration Guide Volume 1* for details.

3   In the primary database, create a stored procedure that updates the primary data. For example:

```
create proc update_pubs
@pub_id char(4), @pub_name varchar(40)
as update publishers
set pub_name = @pub_name
```

```
where pub_id = @pub_id
```

> **Warning!** A stored procedure invoked in a primary database in request
> function delivery is invoked inside a user-defined transaction. See the
> *Adaptive Server Enterprise Transact-SQL User's Guide* for information
> about operations that are not allowed inside user-defined transactions (for
> example, the dump transaction and dump database commands).

Do not mark this stored procedure as replicated. In request function
delivery with versions earlier than 15.1, only the stored procedure in the
replicate database is marked as replicated.

However, if the primary database is also part of a warm standby
application, then mark the stored procedure in the active and standby
primary databases as replicated if you want to replicate the stored
procedures to the standby database.

4   In the primary database, grant execute permission on the stored procedure
to the same user for whom you created a login name and password in step
2. For example:

```
grant execute on update_pubs to pubs_user
```

5   In the replicate database, create a stored procedure with the same
parameters and datatypes as the stored procedure in the primary database.
The new stored procedure must either do nothing or display a message to
indicate a pending update. For example:

```
create proc update_pubs_request
@pub_id char(4), @pub_name varchar(40)
as print "Transaction accepted."
```

> **Note** Use a different name for the stored procedure you create in the
> replicate database from the one created in the primary database. You must
> use a different name if the function will replicate back to the replicate
> database as an applied function. When you create the function replication
> definition in step 8, you must specify the name of the stored procedure in
> the destination (primary) database.

6   In the replicate database, mark the stored procedure for replicated function
delivery using the sp_setrepproc system procedure. For example:

```
sp_setrepproc update_pubs_request, 'function'
```

See "Marking stored procedures for replication" on page 305 for details.

7   In the replicate database, grant execute permission on the stored procedure to the replicate Replication Server user who invokes it. For example:

```
grant execute on update_pubs_request to pubs_user
```

8   In the primary Replication Server that manages the primary data, create a function replication definition for the stored procedure in the replicate database. For example:

```
create function replication definition
update_pubs_request
with primary at TOKYO_DS.pubs2
deliver as 'update_pubs'
(@pub_id char(4), @pub_name varchar(40))
```

The function replication definition must use the same name, parameters and datatypes as the stored procedure in the replicate database. You have the option to include only the parameters you want to replicate.

See Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about create function replication definition command.

In the preceding example, the optional deliver as clause specifies that the stored procedure to run at the primary data server is named update_pubs, not update_pubs_request.

9   When you create a function replication definition, Replication Server automatically creates a corresponding user-defined function.

If you are not using a default function string or you want to customize the function's invocation, you need to create a function string for the user-defined function. See "Creating or modifying a function string for a replicated function" on page 307 for more information.

10   Verify that all the Replication Server and the database objects in steps 1 through 9 exist at their appropriate locations. You must now be able to run the request function.

See Chapter 6, "Adaptive Server Stored Procedures," in the *Replication Server Reference Manual* for information about stored procedures, such as rs_helpfunc, that you can use to query the RSSD for information about the replication system.

Marking stored procedures for replication

The system procedure sp_setrepproc is used to mark stored procedures for replication. The syntax is:

> sp_setrepproc [*proc_name* [, {'function' | 'table' | 'false'}, [{log_current |log_sproc}]]]

where:

proc_name – the name of a stored procedure in the current database.

'function' – enables the replication for a stored procedure associated with a function replication definition.

'table' – enables replication for a stored procedure associated with a table replication definition. For information on replicating stored procedures associated with table replication definitions, see "Asynchronous Procedures" on page 269.

'false' - disables replication for the stored procedure.

'log_current' - logs the execution of the stored procedure you are replicating in the current database, not in the database where the stored procedure resides.

'log_sproc' - logs the execution of the stored procedure you are replicating in the database where the stored procedure resides, not in the current database. 'log_sproc' is the default parameter.

Use sp_setrepproc according to these guidelines:

- To list all replicated objects in the database, enter sp_setrepproc with no parameters.

- To determine the replication status of the stored procedure, enter sp_setrepproc with the stored procedure name only.

- Enter sp_setrepproc with the stored procedure name and 'function', 'table', or 'false' to enable each type of replication or to disable replication for the stored procedure. You must be the System Administrator or the Database Owner to use sp_setrepproc to change the replication status of a stored procedure.

- To log the execution of a replicated stored procedure in the database you selected, enter sp_setrepproc with 'log_current', to log execution in the current database, or 'log_sproc', to log execution in the database where the stored procedure resides.

- Mark the stored procedure in the replicate database as replicated and specify 'function' to indicate the type of replication definition associated with the stored procedure.

  For more information on sp_setrepproc, see the *Replication Server Reference Manual*.

Subscribing to replicated functions

You do not need to create a subscription for a request function. The destination database is always the primary database; you create the function replication definition in the primary Replication Server.

Modifying or dropping replicated functions

This section explains how to modify or drop replicated functions.

Before modifying a function replication definition

1   Change the stored procedure at the primary or replicate data server and provide defaults for the new parameters, if necessary.

2   As a precaution, quiesce the system. Altering functions while updates are in process can have unpredictable results.

    See Chapter 4, "Managing a Replication System" in the *Replication Server Administration Guide Volume 1* for information on how to quiesce the system.

Modifying a function replication definition

To add new parameters, add new searchable parameters, or change the name of the destination stored procedure, use alter function replication definition to alter the function replication definition.

The syntax for this command is:

```
alter function replication definition function_rep_def
```

```
{deliver as proc_name |
add @parameter datatype[, @parameter datatype]... |
add searchable parameters @parameter [, @parameter]...|
send standby {all | replication definition} parameters}
```

The options for alter function replication definition are similar to those for create function replication definition. See Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual* for more information about alter function replication definition command.

See "Creating or modifying a function string for a replicated function" on page 307 for information about function strings for function replication definitions.

Dropping a function replication definition

To change or remove parameters, or to rename a function replication definition, use the drop function replication definition command to drop it, then recreate it.

The syntax for this command is:

    drop function replication definition function_rep_def

When you drop a function replication definition, the associated user-defined function and function string are also dropped.

Creating or modifying a function string for a replicated function

When you create or alter a function replication definition, Replication Server automatically creates or alters the corresponding user-defined function. You must, however, create a function string for the user-defined function if you are not using a class that inherits function strings from rs_default_function_class, either directly or indirectly.

See "User-defined functions" on page 14 for more information.

Create a function string for a user-defined function in the function-string class assigned to the destination database for the replicated function. Use create function string at the primary Replication Server to create a function string for a user-defined function.

See "Function strings and function-string classes" on page 33 for more information.

When you drop a function replication definition, Replication Server always drops the user-defined function and function strings.

You can customize function strings in function-string classes that allow it. In a typical application, the replicated user-defined function passes stored procedure parameter values to the destination Replication Server, and the function string runs the stored procedure with these values in the destination database.

To change the default function string to perform some other action, such as inserting data into an audit log, use the alter function string command at the primary Replication Server for the replicated function. The function-string class assigned to the destination database for the replicated function must allow you to customize function strings.

See Chapter 2, "Customizing Database Operations" for information on creating and altering function strings. See also Chapter 3, "Replication Server Commands," in the *Replication Server Reference Manual*, for more information about the create function string command.

Model variations and strategies

This section describes some model variations and other strategies you can use to implement your replication system design with a version of no-subscription request function earlier than 15.1.

- A basic primary copy model that allows remote sites to make changes to primary data without accessing the primary database directly.

- Pending tables – a strategy used with no-subscription request functions in versions earlier than 15.1 that allows users to see the results of updates to primary data before the update has been returned to the replicate site.

Request functions

You can use no-subscription request functions with versions earlier than 15.1 to invoke stored procedures on primary data. Request functions allow clients at remote replicate sites to update the primary data without accessing the primary database directly. Replication Server invokes a stored procedure on the primary database, which carries out the requested transactions. This method preserves the basic primary copy model while letting you distribute transactions. You do not have to fragment the primary data among replicate sites.

These two examples illustrate:

- A system that uses request functions with versions earlier than 15.1.

- A system that uses request functions with versions earlier than 15.1, applied functions, and a local pending table.

Example 1: A basic example using pre15.1 no-subscription request functions

In this example, a client application at the replicate site (Sydney) executes the stored procedure upd_publishers_pubs2_req, which makes no changes to the replicate database but causes an associated stored procedure, upd_publishers_pubs2, to execute and change data on the primary site (Tokyo).

upd_publishers_pubs2_req is an empty-body stored procedure. Perform these tasks to create a request function. No subscriptions are necessary at the primary database; function executions at a site other than the primary for the replication definition automatically flow to the primary site.

At the replicate site:
- Create an empty-body stored procedure at the replicate data server, and:

    - Mark the procedure for the replicated function delivery using sp_setrepproc.

    - Grant appropriate permissions to the procedure for the appropriate user.

The empty-body procedure either does nothing or displays a message indicating a pending update. It may have a different name than the associated stored procedure in the primary database.

At the primary site:
- Create a stored procedure that updates primary data. Grant permissions to the appropriate user.

- Create a function replication definition at the primary Replication Server for the empty-body stored procedure in the replicate database. The function replication definition and the empty-body stored procedure have the same name.

Once the primary data has been updated via the replication of the request stored procedure, data can be delivered to replicate sites (including the requestor) either via table replication or applied function replication.

Note It is not necessary to create a subscription for a request function.

Figure C-1: Request functions
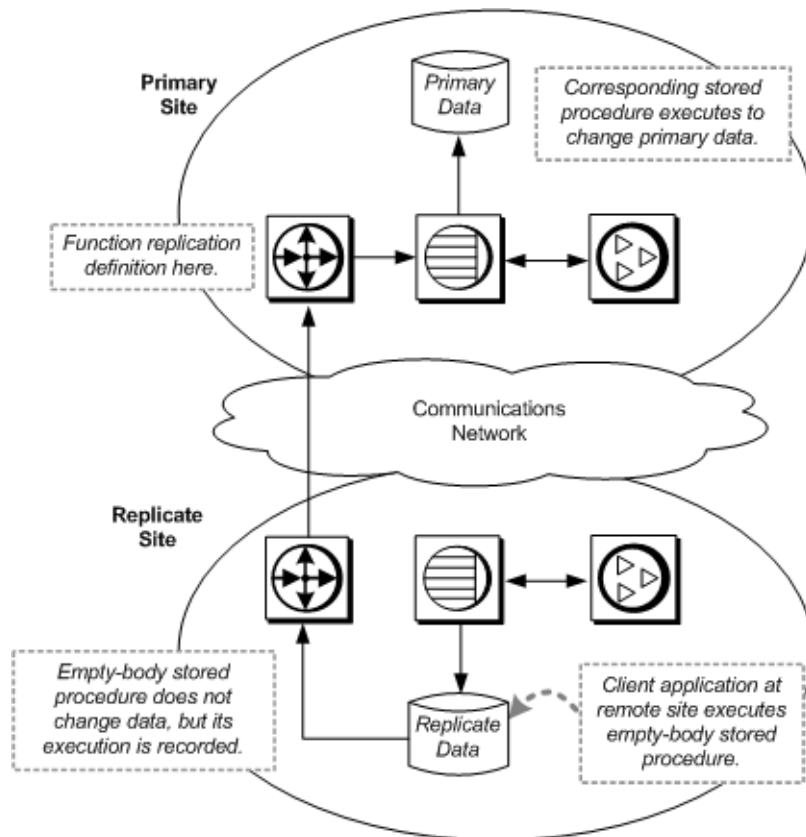
Stored procedures

At the primary site: This script creates the user stored procedure upd_publisher_pubs2 at the primary site.

```
-- Execute this script at Sydney data server
-- Creates stored procedure upd_publishers_pub2
```

```
create procedure upd_publishers_pubs2
 (@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
 as
 insert into publishers
 values
 (@pub_id, @pub_name, @city, @state)
 go
 /* end of script */
```

At the replicate site: This script creates the empty-body stored procedure
upd_publishers_pubs2_req.

```
-- Execute this script at Sydney Replication Server
-- Creates stored procedure
        upd_publishers_pubs2_req
create procedure upd_publishers_pubs2_req
(@pub_id char(4),
@pub_name varchar(40),
@city varchar(20),
@state char(2))
as
begin
if (select 1) > 1
   print "Submitting request."
end
go
/* end of script */
```

Function replication definition

This script creates the function replication definition at the primary Replication
Server. It has the same name as the empty-body stored procedure. The deliver
as clause tells the primary data server to run *upd_publisher_pubs2* at the
primary site.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
create function replication definition
upd_publishers_pubs2_req
 with primary at TOKYO_DS.pubs2
 deliver as 'upd_publisher_pubs2'
    (@pub_id char(4),
    @pub_name varchar(40),
    @city varchar(20),
```

```
    @state char(2))
 go
 /* end of script */
```

Example 2: An example using a local pending table

The pending table is a design enhancement of applied and request functions that allows clients at a remote site to update primary data and see the updates at the replicate site before they are returned from the primary site. Use this model to implement local update applications.

In this strategy, a client application at a remote replicate site executes a user stored procedure that updates data at the primary site using a request function. Changes to the primary data are replicated to the remote replicate site via an applied function. A local pending table lets clients at the remote replicate site to see updates that are pending at the replicate site before the replication system returns the updates.

When a client application executes the user stored procedure at the replicate data server, it:

- Causes an associated stored procedure to execute and update data at the primary site.

- Enters those updates in the local pending table.

When the update succeeds at the primary database, it is distributed to the remote replicate sites, including the site where the transaction originated. At the replicate site, a stored procedure updates the replicated table and deletes the corresponding updates from the pending table.

To use applied functions, request functions, and a local pending table, you must:

At the replicate site:

- Create a pending table in the replicate database and grant appropriate permissions.

- Create a user stored procedure in the replicate database that initiates the request function and inserts data updates into the pending table.

- Mark the user stored procedure for replicated function delivery using sp_setrepproc.

- Grant procedure permissions to the appropriate user.

- Create a user stored procedure in the replicate database that updates the replicated table and deletes the corresponding update from the pending table. Grant appropriate permissions to the maintenance user.

- Create a subscription to the function replication definition.

At the primary site:

- Create the stored procedure that modifies the primary data.

- Create the function replication definition for the applied function.

- Create the function replication definition for the request function.

In this example, a client application at the replicate site (Sydney) executes a stored procedure upd_publishers_pubs2_req, which inserts values in the publishers_pend table and causes an associated stored procedure, upd_publishers_pubs2, to execute at the primary site (Tokyo). Execution of upd_publishers_pubs2 at the primary site causes the stored procedure upd_publishers_pubs to execute at the replicate site, which updates the publishers table and deletes the corresponding information from the publishers_pend table.

Figure C-2 illustrates the data flow when you use applied functions, request functions, and a local pending table. The gray arrows show the flow of the request function delivery. The black arrows show the flow of the applied function delivery.

Figure C-2: Request functions and a local pending table
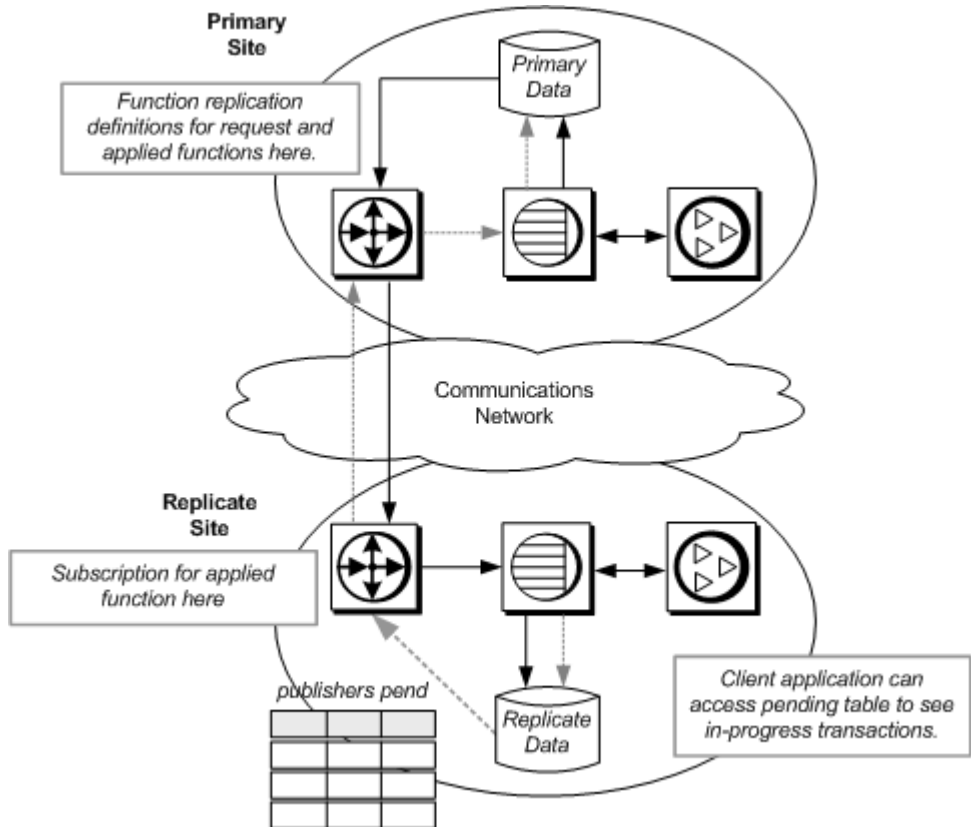
Pending table

This script creates a pending table in the replicate database.

```
-- Execute this script at Sydney data server
-- Creates local pending table
create table publishers_pend
(pub_id char(4) not null,
pub_name varchar(40) null,
city varchar(20) null,
statechar(2) null)
go
/* end of script */
```

Stored procedures

The script creates the stored procedure upd_publisher_pubs2 at the primary site.

The following script modifies the upd_publishers_pub2_req stored procedure at the replicate site. The insert into clause tells the replicate Replication Server to insert values into the publishers_pend table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure
create procedure upd_publishers_pubs2_req
(@pub_id char(4),
@pub_name varchar(40),
@city varchar(20),
@state char(2))
as
insert into publishers_pend
values
   (@pub_id, @pub_name, @city, @state)
go
/* end of script */
```

This script creates the upd_publishers_pubs2 procedure for the replicate site. It updates the publishers table and deletes the corresponding information from the publishers_pend table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure upd_publishers_pubs2
create procedure upd_publishers_pubs2
(@pub_id char(4),
@pub_name varchar(40),
@city varchar(20),
@state char(2))
as
update publishers
set
   pub_name = @pub_name,
   city = @city,
   state = @state
where
   pub_id = @pub_id
delete from publishers_pend
where
   pub_id = @pub_id
go
/* end of script */
```

Function replication definitions

The script "Function replication definition" on page 311 creates the function replication definition upd_publishers_pubs2_req—for the request function with versions earlier than 15.1 —at the primary site.

This script creates a function replication definition for the publishers table at the primary Replication Server. The replication definition uses the same parameters and datatypes as the stored procedure in the primary database.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
upd_publishers_pubs2
create function replication definition
upd_publishers_pubs2
with primary at TOKYO_DS.pubs2
(@pub_id char(4),
@pub_name varchar(40),
@city varchar(20),
@state char(2))
go
/* end of script */
```

Subscription

This script creates a subscription at the replicate Replication Server using the no-materialization method for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription using no-materialization
for upd_publishers_pubs2

create subscription upd_publishers_pubs2_sub
for upd_publishers_pubs2
with replicate at SYDNEY_DS.pubs2
without materialization
go
/* end of script */
```

Glossary

active database In a warm standby application, a database that is replicated to a standby database. See also **warm standby application**.

Adaptive Server The Sybase version 11.5 and later relational database server. If you choose the RSSD option when configuring Replication Server, Adaptive Server maintains Replication Server system tables in the RSSD database.

application programming interface (API) A predefined interface through which users or programs communicate with each other. Open Client and Open Server are examples of APIs that communicate in a client/server architecture. RCL, the Replication Command Language, is the Replication Server API.

applied function A replicated function, associated with a function replication definition, that Replication Server delivers from a primary database to a subscribing replicate database. The function passes parameter values to a stored procedure that is executed at the replicate database. The stored procedure executed at the replicate database by the maintenance user. See also **replicated function delivery**, **request function**, and **function replication definition**.

article A replication definition extension for tables or stored procedures that can be an element of a publication. Articles may or may not contain where clauses, which specify a subset of rows that the replicate database receives.

asynchronous procedure delivery A method of replicating, from a source to a destination database, a stored procedure that is associated with a table replication definition.

asynchronous command A command that a client submits where the client is not prevented from proceeding with other operations before the completion status is received. Many Replication Server commands function as asynchronous commands within the replication system.

atomic materialization	A materialization method that copies subscription data from a primary to a replicate database through the network in a single atomic operation, using a select operation with a holdlock. No changes to primary data are allowed until data transfer is complete. Replicate data may be applied either as a single transaction or in increments of ten rows per transaction, which ensures that the replicate database transaction log does not fill. Atomic materialization is the default method for the create subscription command. See also **nonatomic materialization**, **bulk materialization** and **no materialization**.
autocorrection	Autocorrection is a setting applied to replication definitions, using the set autocorrection command, to prevent failures caused by missing or duplicate rows in a copy of a replicated table. When autocorrection is enabled, Replication Server converts each update or insert operation into a delete followed by an insert. Autocorrection should *only* be enabled for replication definitions whose subscriptions use nonatomic materialization.
base class	A function-string class that does not inherit function strings from a parent class. See also **function-string class**.
bitmap subscription	A type of subscription that replicates rows based on bitmap comparisons. Create columns using the int datatype, and identify them as the rs_address datatype when you create a replication definition. When you create a subscription, compare each rs_address column to a bitmask using a bitmap comparison operator (&) in the where clause. Rows matching the subscription's bitmap are replicated.
bulk materialization	A materialization method whereby subscription data in a replicate database is initialized outside of the replication system. For example, data may be transferred from a primary database using media such as magnetic tape, diskette, CD-ROM, or optical storage disk. Bulk materialization involves a series of commands, starting with define subscription. You can use bulk materialization for subscriptions to table replication definitions or function replication definitions. See also **atomic materialization**, **nonatomic materialization**, and **no materialization**.
centralized database system	A database system where data is managed by a single database management system at a centralized location.
class	See **error class** and **function-string class**.
class tree	A set of function-string classes, consisting of two or more levels of derived and parent classes, that derive from the same base class. See also **function-string class**.

client	A program connected to a server in a client/server architecture. It may be a front-end application program executed by a user or a utility program that executes as an extension of the system.
Client/Server Interfaces (C/SI)	The Sybase interface standard for programs executing in a client/server architecture.
concurrency	The ability of multiple clients to share data or resources. Concurrency in a database management system depends upon the system protecting clients from conflicts that arise when data in use by one client is modified by another client.
connection	A connection from a Replication Server to a database. See also **Data Server Interface (DSI)** and **logical connection**.
coordinated dump	A set of database dumps or transaction dumps that is synchronized across multiple sites by distributing an rs_dumpdb or rs_dumptran function through the replication system.
database	A set of related data tables and other objects that is organized and presented to serve a specific purpose.
database generation number	Stored in both the database and the RSSD of the Replication Server that manages the database, the database generation number is the first part of the origin queue ID (*qid*) of each log record. The origin queue ID ensures that the Replication Server does not process duplicate records. During recovery operations, you may need to increment the database generation number so that Replication Server does not ignore records submitted after the database is reloaded.
database replication definition	A description of a set of database objects—tables, transactions, functions, system stored procedures, and DDL—for which a subscription can be created.
	You can also create table replication definitions and function replication definitions. See also **table replication definition** and **function replication definition**.
database server	A server program, such as Sybase Adaptive Server, that provides database management services to clients.
data definition language (DDL)	The set of commands in a query language, such as Transact-SQL, that describes data and their relationships in a database. DDL commands in Transact-SQL include those using the create, drop, and alter keywords.
data manipulation language (DML)	The set of commands in a query language, such as Transact-SQL, that operates on data. DML commands in Transact-SQL include select, insert, update, and delete.

data server	A server whose client interface conforms to the Sybase Client/Server Interfaces and provides the functionality necessary to maintain the physical representation of a replicated table in a database. Data servers are usually database servers, but they can also be any data repository with the interface and functionality Replication Server requires.
Data Server Interface (DSI)	Replication Server threads corresponding to a connection between a Replication Server and a database. DSI threads submit transactions from the DSI outbound queue to a replicate data server. They consist of a scheduler thread and one or more executor threads. The scheduler thread groups the transactions by commit order and dispatches them to the executor threads. The executor threads map functions to function strings and execute the transactions in the replicate database. DSI threads use an Open Client connection to a database. See also **outbound queue** and **connection**.
data source	A specific combination of a database management system (DBMS) product such as a relational or non-relational data server, a database residing in that DBMS, and the communications method used to access that DBMS from other parts of a replication system. See also **database** and **data server**.
decision support application	A database client application characterized by ad hoc queries, reports, and calculations and few data update transactions.
declared datatype	The datatype of the value delivered to the Replication Server from the Replication Agent:
	• If the Replication Agent delivers a base Replication Server datatype, such as datetime, to the Replication Server, the declared datatype is the base datatype.
	• Otherwise, the declared datatype must be the UDD for the original datatype at the primary database.
default function string	The function string that is provided by default for the system-provided classes rs_sqlserver_function_class and rs_default_function_class and classes that inherit function strings from these classes, either directly or indirectly. See also **function string**.
dematerialization	The optional process, when a subscription is dropped, whereby specific rows that are not used by other subscriptions are removed from the replicate database.
derived class	A function-string class that inherits function strings from a parent class. See also **function-string class** and **parent class**.

direct route	A route used to send messages directly from a source to a destination Replication Server, with no intermediate Replication Servers. See also **indirect route** and **route**.
disk partition	See **partition**.
distributed database system	A database system where data is stored in multiple databases on a network. The databases may be managed by data servers of the same type (for example, Adaptive Server) or by heterogeneous data servers.
Distributor	A Replication Server thread (DIST) that helps to determine the destination of each transaction in the inbound queue.
dump marker	A message written by Adaptive Server in a database transaction log when a dump is performed. In a warm standby application, when you are initializing the standby database with data from the active database, you can specify that Replication Server use the dump marker to determine where in the transaction stream to begin applying transactions in the standby database. See also **warm standby application**.
Embedded Replication Server System Database (ERSSD)	The Adaptive Server Anywhere (ASA) database that stores Replication Server system tables. You can choose whether to store Replication Server system tables on the ERSSD or the Adaptive Server RSSD. See also **Replication Server System Database (RSSD)**.
error action	A Replication Server response to a data server error. Possible Replication Server error actions are ignore, warn, retry_log, log, retry_stop, and stop_replication. Error actions are assigned to specific data server errors.
error class	A name for a collection of data server error actions that are used with a specified database.
exceptions log	A set of three Replication Server system tables that holds information about transactions that failed on a data server. The transactions in the log must be resolved by a user or by an intelligent application. You can use the rs_helpexception stored procedure to query the exceptions log.
Failover	Sybase Failover allows you to configure two version 12.0 and later Adaptive Servers as companions. If the primary companion fails, that server's devices, databases, and connections can be taken over by the secondary companion.
	For more detailed information about how Sybase Failover works in Adaptive Server, refer to *Using Sybase Failover in a High Availability System*, which is part of the Adaptive Server Enterprise documentation set.

For instructions on how to enable Failover support for non-RSSD Replication Server connections to Adaptive Server, see "Configuring the replication system tosupport Sybase Failover" in Chapter 7, "Replication System Recovery".

fault tolerance
The ability of a system to continue to operate correctly even though one or more of its component parts is malfunctioning.

function
A Replication Server object that represents a data server operation such as insert, delete, select, or begin transaction. Replication Server distributes such operations to other Replication Servers as functions. Each function consists of a function name and a set of data parameters. In order to execute the function in a destination database, Replication Server uses function strings to convert a function to a command or set of commands for a type of database. See also **user-defined function**, and **replicated function delivery**.

function replication definition
A description of a replicated function used in replicated function delivery. The function replication definition, maintained by Replication Server, includes information about the parameters to be replicated and the location of the primary version of the affected data. There are two types of function replication definition, applied and request. See also **replicated function delivery**.

function scope
The range of a function's effect. Functions have replication definition scope or function-string class scope. A function with replication definition scope is defined for a specific replication definition, and cannot be applied to other replication definitions. A function with function-string class scope is defined once for a function-string class and is available only within that class.

function string
A string that Replication Server uses to map a database command to a data server API. For the rs_select and rs_select_with_lock functions only, the string contains an input template, used to match function strings with the database command. For all functions, the string also contains an output template, used to format the database command for the destination data server.

function-string class
A named collection of function strings used with a specified database connection. Function-string classes include those provided with Replication Server and those you have created. Function-string classes can share function string definitions through function-string inheritance. The three system-provided function-string classes are rs_sqlserver_function_class, rs_default_function_class, and rs_db2_function_class. See also **base class**, **class tree**, **derived class**, **function-string inheritance**, and **parent class**.

function-string inheritance
The ability to share function string definitions between classes, whereby a derived class inherits function strings from a parent class. See also **derived class**, **function-string class**, and **parent class**.

function-string variable	An identifier used in a function string to represent a value that is to be substituted at run time. Variables in function strings are enclosed in question marks (?). They represent column values, function parameters, system-defined variables, or user-defined variables.
function subscription	A subscription to a function replication definition (used in both applied and request function delivery).
generation number	See **database generation number**.
heterogeneous data servers	Multi-vendor data servers used together in a distributed database system.
high availability (HA)	Very low downtime. Computer systems that provide HA usually provide 99.999% availability, or roughly five minutes unscheduled downtime per year.
hibernation mode	A Replication Server state in which all DDL commands, except admin and sysadmin commands, are rejected; all routes and connections are suspended; most service threads, such as DSI and RSI, are suspended; and RSI and RepAgent users are logged off and not allowed to log on. Used during route upgrades, and may be turned on for a Replication Server to debug problems.
hot standby application	A database application in which the standby database can be placed into service without interrupting client applications and without losing any transactions. See also **warm standby application**.
ID Server	One Replication Server in a replication system is the ID Server. In addition to performing the usual Replication Server tasks, the ID Server assigns unique ID numbers to every Replication Server and database in the replication system, and maintains version information for the replication system.
inbound queue	A stable queue used to spool messages from a Replication Agent to a Replication Server.
indirect route	A route used to send messages from a source to a destination Replication Server, through one or more intermediate Replication Servers. See also **direct route** and **route**.
interfaces file	A file containing entries that define network access information for server programs in a Sybase client/server architecture. Server programs may include Adaptive Servers, gateways, Replication Servers, and Replication Agents. The interfaces file entries enable clients and servers to connect to each other in a network.

latency	The measure of the time it takes to distribute to a replicate database a data modification operation first applied in a primary database. The time includes Replication Agent processing, Replication Server processing, and network overhead.
local-area network (LAN)	A system of computers and devices, such as printers and terminals, connected by cabling for the purpose of sharing data and devices.
locator value	The value stored in the rs_locater table of the Replication Server's RSSD that identifies the latest log transaction record received and acknowledged by the Replication Server from each previous site during replication.
logical connection	A database connection that Replication Server maps to the connections for the active and standby databases in a warm standby application. See also **connection** and **warm standby application**.
login name	The name that a user or a system component such as Replication Server uses to log in to a data server, Replication Server, or Replication Agent.
Log Transfer Language (LTL)	A subset of the Replication Command Language (RCL). A Replication Agent such as RepAgent uses LTL commands to submit to Replication Server the information it retrieves from primary database transaction logs.
maintenance user	A data server login name that Replication Server uses to maintain replicate data. In most applications, maintenance user transactions are not replicated.
materialization	The process of copying data specified by a subscription from a primary database to a replicate database, thereby initializing the replicate table. Replicate data can be transferred over a network, or, for subscriptions involving large amounts of data, loaded initially from media. See also **atomic materialization**, **bulk materialization**, **no materialization**, and **nonatomic materialization**.
materialization queue	A stable queue used to spool messages related to a subscription being materialized or dematerialized.
missing row	A row missing from a replicated copy of a table but present in the primary table.
mixed-version system	A replication system containing Replication Servers of different software versions that have different capabilities based on their different software versions and site versions. Mixed-version support is available only if the system version is 11.0.2 or greater.

For example, a replication system containing Replication Servers version 11.5 or later and version 11.0.2 is a mixed-version system. A replication system containing Replication Servers of releases earlier than release 11.0.2 is not a mixed-version system, because any newer Replication Servers are restricted by the system version from using certain new features. See also **site version** and **system version**.

more columns Columns in a replication definition exceeding 250, but limited to 1024. More columns are supported by Replication Server version 12.5 and later.

multi-site availability (MSA) Methodology for replicating database objects—tables, functions, transactions, system stored procedures, and DDL from the primary to the replicate database. See also **database replication definition**.

name space The scope within which an object name must be unique.

nonatomic materialization A materialization method that copies subscription data from a primary to a replicate database through the network in a single operation, without a holdlock. Changes to the primary table are allowed during data transfer, which may cause temporary inconsistencies between replicate and primary databases. Data is applied in increments of ten rows per transaction, which ensures that the replicate database transaction log does not fill. Nonatomic materialization is an optional method for the create subscription command. See also **autocorrection**, **atomic materialization**, **no materialization**, and **bulk materialization**.

network-based security Secure transmission of data across a network. Replication Server supports third-party security mechanisms that provide user authentication, unified login, and secure message transmission between Replication Servers.

no materialization A materialization method that lets you create a subscription when the subscription data already exists at the replicate site. Use the create subscription command with the without materialization clause. You can use this method to create subscriptions to table replication definitions and function replication definitions. See also **atomic materialization** and **bulk materialization**.

online transaction processing (OLTP) application A database client application characterized by frequent transactions involving data modification (inserts, deletes, and updates).

Origin Queue ID (qid) Formed by the RepAgent, the qid uniquely identifies each log record passed to the Replication Server. It includes the date and timestamp and the database generation number. See also **database generation number**.

orphaned row A row in a replicated copy of a table that does not match an active subscription.

outbound queue	A stable queue used to spool messages. The DSI outbound queue spools messages to a replicate database. The RSI outbound queue spools messages to a replicate Replication Server.
parallel DSI	Configuring a database connection so that transactions are applied to a replicate data server using multiple DSI threads operating in parallel, rather than a single DSI thread. See also **connection** and **Data Server Interface (DSI)**.
parameter	An identifier representing a value that is provided when a procedure executes. Parameter names are prefixed with an @ character in function strings. When a procedure is called from a function string, Replication Server passes the parameter values, unaltered, to the data server. See also **searchable parameter**.
parent class	A function-string class from which a derived class inherits function strings. See also **function-string class** and **derived class**.
partition	A raw disk partition or operating system file that Replication Server uses for stable queue storage. Only use operating system files in a test environment.
physical connection	See **connection**.
primary data	The definitive version of a set of data in a replication system. The primary data is maintained on a data server that is known to all of the Replication Servers with subscriptions for the data.
primary database	Any database that contains data that is replicated to another database via the replication system.
primary fragment	A horizontal segment of a table that holds the primary version of a set of rows.
primary key	A set of table columns that uniquely identifies each row.
primary site	A Replication Server where a function-string class or error class is defined. See **error class** and **function-string class**.
principal user	The user who starts an application. When using network-based security, Replication Server logs in to remote servers as the principal user.
projection	A vertical slice of a table, representing a subset of the table's columns.
publication	A group of articles from the same primary database. A publication lets you collect replication definitions for related tables and/or stored procedures and then subscribe to them as a group. You collect replication definitions as articles in a publication at the source Replication Server and subscribe to them with a publication subscription at the destination Replication Server. See also **article** and **publication subscription**.

publication subscription	A subscription to a publication. See also **article** and **publication**.
published datatype	The datatype of the column after the column-level translation (and before a class-level translation, if any) at the replicate data server. The published datatype must be either a Replication Server base datatype or a UDD for the datatype in the target data server. If the published datatype is omitted from the replication definition, it defaults to the declared datatype.
query	In a database management system, a query is a request to retrieve data that meets a given set of criteria. The SQL database language includes the select command for queries.
quiescent	A quiescent replication system is one in which all updates have been propagated to their destinations. Some Replication Server commands or procedures require that you first quiesce the replication system.
remote procedure call (RPC)	A request to execute a procedure that resides in a remote server. The server that executes the procedure could be a Adaptive Server, a Replication Server, or a server created using Open Server. The request can originate from any of these servers or from a client application. The RPC request format is a part of the Sybase Client/Server Interfaces.
RepAgent thread	The Replication Agent for Adaptive Server databases. RepAgent is an Adaptive Server thread; it transfers transaction log information from the primary database to a Replication Server for distribution to other databases.
replicate database	Any database that contains data that is replicated from another database via the replication system.
replicated function delivery	A method of replicating, from a source to a destination database, a stored procedure that is associated with a function replication definition. See also **applied function**, **request function**, and **function replication definition**.
replicated stored procedure	An Adaptive Server stored procedure that is marked as replicated using the sp_setrepproc or the sp_setreplicate system procedure. Replicated stored procedures can be associated with function replication definitions or table replication definitions. See also **replicated function delivery** and **asynchronous procedure delivery**.
replicated table	A table that is maintained by Replication Server, in part or in whole, in databases at multiple locations. There is one primary version of the table, which is marked as replicated using the sp_setreptable or the sp_setreplicate system procedure; all other versions are replicated copies.

Replication Agent	A program or module that transfers transaction log information representing modifications made to primary data from a database server to a Replication Server for distribution to other databases. RepAgent is the Replication Agent for Adaptive Server databases.
Replication Command Language (RCL)	The commands used to manage information in Replication Server.
replication definition	Usually, a description of a table for which subscriptions can be created. The replication definition, maintained by Replication Server, includes information about the columns to be replicated and the location of the primary version of the table.
	You can also create function replication definitions; sometimes the term "table replication definition" is used to distinguish between table and function replication definitions. See also **function replication definition**.
Replication Server	The Sybase server program that maintains replicated data, typically on a LAN, and processes data transactions received from other Replication Servers on the same LAN or on a WAN.
Replication Server Interface (RSI)	A thread that logs in to a destination Replication Server and transfers commands from the RSI outbound stable queue to the destination Replication Server. There is one RSI thread for each destination Replication Server that is a recipient of commands from a primary or intermediate Replication Server. See also **outbound queue** and **route**.
Replication Monitoring Services (RMS)	A small Java application built using the Sybase Unified Agent Framework (UAF) that monitors and troubleshoot a replication environment.
replication system administrator	The system administrator that manages routine operations in the Replication Server.
Replication Server System Database (RSSD)	The Adaptive Server database containing a Replication Server system tables. You can choose whether to store Replication Server system tables on the RSSD or the Adaptive Server Anywhere (ASA) ERSSD. See also **Embedded Replication Server System Database (ERSSD)**.
Replication Server system Adaptive Server	The Adaptive Server with the database containing a Replication Server's system tables (the RSSD).
replication system	A data processing system where data is replicated in multiple databases to provide remote users with the benefits of local data access. Specifically, a replication system that is based upon Replication Server and includes other components such as Replication Agents and data servers.

replication system domain	All replication system components that use the same ID Server.
request function	A replicated function, associated with a function replication definition, that Replication Server delivers from a primary database to a replicate database. The function passes parameter values to a stored procedure that is executed at the replicate database. The stored procedure is executed at the replicate site by the same user as it is at the primary site.See also **replicated function delivery**, **request function**, and **function replication definition**.
route	A one-way message stream from a source Replication Server to a destination Replication Server. Routes carry data modification commands (including those for RSSDs) and replicated functions or stored procedures between Replication Servers. See also **direct route** and **indirect route**.
route version	The lower of the site version numbers of the route's source and destination Replication Servers. Replication Server version 11.5 and later use the route version number to determine which data to send to the replicate site. See also **site version**.
row migration	The process whereby column value changes in rows in a primary version of a table cause corresponding rows in a replicate version of the table to be inserted or deleted, based on comparison with values in a subscription's where clause.
SQL Server	The Sybase relational database pre-11.5 server.
schema	The structure of the database. DDL commands and system procedures change system tables stored in the database. Supported DDL commands and system procedures can be replicated to standby databases when you use Replication Server version 11.5 or later and Adaptive Server version 11.5 or later.
searchable column	A column in a replicated table that can be specified in the where clause of a subscription or article to restrict the rows replicated at a site.
searchable parameter	A parameter in a replicated stored procedure that can be specified in the where clause of a subscription to help determine whether or not the stored procedure should be replicated. See also **parameter**.
secondary truncation point	See **truncation point**.
site	An installation consisting of, at minimum, a Replication Server, data server, and database, and possibly a Replication Agent, usually at a discrete geographic location. The components at each site are connected over a WAN to those at other sites in a replication system. See also **primary site**.

site version	The version number for an individual Replication Server. Once the site version has been set to a particular level, the Replication Server enables features specific to that level, and downgrades are not allowed. See also **software version**, **route version**, and **system version**.
software version	The version number of the software release for an individual Replication Server. See also **site version** and **system version**.
Stable Queue Manager (SQM)	A thread that manages the stable queues. There is one Stable Queue Manager (SQM) thread for each stable queue accessed by the Replication Server, whether inbound or outbound.
Stable Queue Transaction (SQT) interface	A thread that reassembles transaction commands in commit order. A Stable Queue Transaction (SQT) interface thread reads from inbound stable queues, puts transactions in commit order, then sends them to the Distributor (DIST) thread or a DSI thread, depending on which thread required the SQT ordering of the transaction.
stable queues	Store-and-forward queues where Replication Server stores messages destined for a route or database connection. Messages written into a stable queue remain there until they can be delivered to the destination Replication Server or database. Replication Server builds stable queues using its disk partitions. See also **inbound queue**, **outbound queue**, and **materialization queue**.
standalone mode	A special Replication Server mode used for initiating recovery operations.
standby database	In a warm standby application, a database that receives data modifications from the active database and serves as a backup of that database. See also **warm standby application**.
stored procedure	A collection of SQL statements and optional control-of-flow statements stored under a name in a Adaptive Server database. Stored procedures supplied with Adaptive Server are called system procedures. Some stored procedures for querying the RSSD are included with the Replication Server software.
subscription	A request for Replication Server to maintain a replicated copy of a table, or a set of rows from a table, in a replicate database at a specified location. You can also subscribe to a function replication definition, for replicating stored procedures.
subscription dematerialization	See **dematerialization**.
subscription materialization	See **materialization**.

subscription migration	See **row migration**.
Sybase Central	A graphical tool that provides a common interface for managing Sybase and Powersoft products. Replication Server uses Replication Manager as a Sybase Central plug-in. See also **Replication Monitoring Services (RMS)**.
symmetric multiprocessing (SMP)	On a multiprocessor platform, the ability of an application's threads to run in parallel. Replication Server supports SMP, which can improve server performance and efficiency.
synchronous command	A command that a client considers complete only after the completion status is received.
system function	A function that is predefined and part of the Replication Server product. Different system functions coordinate replication activities, such as rs_begin, or perform data manipulation operations, such as rs_insert, rs_delete, and rs_update.
system-provided classes	Replication Server provides the error class rs_sqlserver_error_class and the function-string classes rs_sqlserver_function_class, rs_default_function_class, and rs_db2_function_class. Function strings are generated automatically for the system-provided function-string classes and for any derived classes that inherit from these classes, directly or indirectly. See also **error class** and **function-string class**.
system version	The version number for a replication system that represents the version for which new features are enabled, for Replication Servers of release 11.0.2 or earlier, and below which no Replication Server can be downgraded or installed. For a Replication Server version 11.5, your use of certain new features requires a site version of 1150 and a system version of at least 1102. See also **mixed-version system**, **site version**, and **software version**.
table replication definition	See **replication definition**.
table subscription	A subscription to a table replication definition.
thread	A process running within Replication Server. Built upon Sybase Open Server, Replication Server has a multi-threaded architecture. Each thread performs a certain function such as managing a user session, receiving messages from a Replication Agent or another Replication Server, or applying messages to a database. See also **Data Server Interface (DSI)**, **Distributor**, and **Replication Server Interface (RSI)**.

transaction	A mechanism for grouping statements so that they are treated as a unit: either all statements in the group are executed or no statements in the group are executed.
Transact-SQL	The relational database language used with Adaptive Server. It is based on standard SQL (Structured Query Language), with Sybase extensions.
truncation point	An Adaptive Server database that holds primary data has an active truncation point, marking the transaction log location where Adaptive Server has completed processing. This is the primary truncation point.
	The RepAgent for an Adaptive Server database maintains a secondary truncation point, marking the transaction log location separating the portion of the log successfully submitted to the Replication Server from the portion not yet submitted. The secondary truncation point ensures that each operation enters the replication system before its portion of the log is truncated.
user-defined function	A function that allows you to create custom applications that use Replication Server to distribute replicated functions or asynchronous stored procedures between sites in a replication system. In replicated function delivery, a user-defined function is automatically created by Replication Server when you create a function replication definition.
variable	See **function-string variable**.
version	See **mixed-version system**, **site version**, **software version**, and **system version**.
warm standby application	An application that employs Replication Server to maintain a standby database for a database known as the active database. If the active database fails, Replication Server and client applications can switch to the standby database.
wide-area network (WAN)	A system of local-area networks (LANs) connected together with data communication lines.
wide columns	Columns in a replication definition containing char, varchar, binary, varbinary, unichar, univarchar, or Java inrow data that are wider that 255 bytes. Wide columns are supported by Replication Server version 12.5 and later.
wide data	Wide data rows, limited to the size of the data page on the data server. Adaptive Server supports page sizes of 2K, 4K, 8K, and 16K. Wide data is supported by Replication Server version 12.5 and later.
wide messages	Messages larger that 16K that span blocks. Wide messages are supported by Replication Server version 12.5 and later.

Index

Numerics

1024 columns, limit to expressions in where clause
116

A

abort switch command   92
abstract plans, replication of   60
activate subscription command
    with suspension at replicate only clause   120
    with suspension clause   120
active database   56
    managing old active after switching   93
    restarting clients   93
Adaptive Server
    error handling   212
admin commands   91
    described   7
alarm daemon (dAlarm)   129
allow connections command   264
alter connection command
    assigning databases to function-string classes   31
alter function command   282
alter function string command   41
    mapping user-defined functions   284
    replacing default function string   273
alter logical connection command   101
alter table command support for warm standby   111
applied stored procedures
    prerequisites for implementing   273
    setting up   274
assign action command   212
asynchronous I/O daemon (dAIO)   129
asynchronous stored procedures
    adding parameters to   282
    and non-unique user-defined function name   285
    applied   271
    executing   269

    request stored procedures   272
    user-defined functions   281
atomic materialization
    in warm standby applications   118

B

batch commands in function strings   46
batch configuration parameter   137
bcp utility program   78, 119
bulk materialization
    in warm standby applications   119

C

caching
    stable queue   143
case, in RCL commands   xviii
certifications
    component   xv
    product   xv
changing
    function strings   16
check subscription command
    after executing switch active command   119, 120
client application
    restarting after active switch   93
clusters
    Sun   287
    terminology   288
commands
    for managing function replication definition   301
    hareg   295
configuration parameters
    affecting performance   131
    dynamic_sql   179
    dynamic_sql_cache_management   180
    dynamic_sql_cache_size   179

E