SYBASE®
An SAP® Company

Administration Guide Volume 2

Replication Server® 15.7

DOCUMENT ID: DC32518-01-1570-01
LAST REVISED: November 2011

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

Conventions

These style and syntax conventions are used in Sybase® documentation.

Style conventions

Key	Definition
`monospaced(fixed-width)`	• SQL and program code • Commands to be entered exactly as shown • File names • Directory names
`italic monospaced`	In SQL or program code snippets, placeholders for user-specified values (see example below).
italic	• File and variable names • Cross-references to other topics or documents • In text, placeholders for user-specified values (see example below) • Glossary terms in text
bold san serif	• Command, function, stored procedure, utility, class, and method names • Glossary entries (in the Glossary) • Menu option paths • In numbered task or procedure steps, user-interface (UI) elements that you click, such as buttons, check boxes, icons, and so on

If necessary, an explanation for a placeholder (system- or setup-specific values) follows in text. For example:

Run:

```
installation directory\start.bat
```

where *installation directory* is where the application is installed.

Syntax conventions

Key	Definition
{ }	Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces when you enter the command.
[ ]	Brackets mean that choosing one or more of the enclosed options is optional. Do not type the brackets when you enter the command.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you can select only one of the options shown.
,	The comma means you can choose as many of the options shown as you like, separating your choices with commas that you type 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.

Case-sensitivity

- All command syntax and command examples are shown in lowercase. However, replication command names are not case-sensitive. For example, **RA_CONFIG**, **Ra_Config**, and **ra_config** are equivalent.
- Names of configuration parameters are case-sensitive. For example, **Scan_Sleep_Max** is not the same as **scan_sleep_max**, and the former would be interpreted as an invalid parameter name.
- Database object names are not case-sensitive in replication commands. However, to use a mixed-case object name in a replication command (to match a mixed-case object name in the primary database), delimit the object name with quote characters. For example: **pdb_get_tables** "*TableName*"
- Identifiers and character data may be case-sensitive, depending on the sort order that is in effect.
  - If you are using a case-sensitive sort order, such as “binary,” you must enter identifiers and character data with the correct combination of uppercase and lowercase letters.
  - If you are using a sort order that is not case-sensitive, such as “nocase,” you can enter identifiers and character data with any combination of uppercase or lowercase letters.

Terminology

Replication Agent™ is a generic term used to describe the Replication Agents for Adaptive Server® Enterprise, Oracle, IBM DB2 UDB, and Microsoft SQL Server. The specific names are:

- RepAgent – Replication Agent thread for Adaptive Server Enterprise
- Replication Agent for Oracle

- Replication Agent for Microsoft SQL Server
- Replication Agent for UDB – for IBM DB2 on Linux, Unix, and Windows

Verify and Monitor Replication Server

Verifying and monitoring Replication Server® includes checking error logs, verifying that the components of a replication system are running, and monitoring the status of system components and processes.

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, see the 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. Basic troubleshooting tasks you can perform on the replication system include:

- Checking error logs for status and error messages
- 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
- Monitoring Replication Server and its threads and checking partition threshold levels

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

Check Replication System Log Files for Errors

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.

See *Replication Server Reference Manual > Replication Server Commands* 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

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.

Prerequisites

Ensure that no threads are down before you confirm that the replication system is working.

Task

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 threads that are not running. Alternatively, execute **admin who** to display information about all threads.

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 **admin who** 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

See also

- *Display Replication System Thread Status* on page 9

Monitor Replication Server

While the replication system is in operation, you may need to monitor its components and processes.

You may need 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.

Verify Server Status

There are several methods you can use to verify the status of your servers.

- Use **isql** to log in to each server. If the login succeeds, 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 cannot 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

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.

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 appears 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.

See *Replication Server Administration Guide Volume 1 > Managing Replication Server with Sybase Central*.

Display Replication System Thread Status

Display general information on the different types of current Replication Server threads with the relevant **admin who** command or a system procedure.

Table 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**

See:

- *Replication Server Troubleshooting Guide* to interpret the command output for troubleshooting purposes.
- *Replication Server Reference Manual > Replication Server Commands >* ***admin who***
- *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***sp_help_rep_agent***
- *Adaptive Server Enterprise > Reference Manual: Procedures > System Procedures >* ***sp_who***

Use System Information Commands

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

See *Replication Server Reference Manual > Replication Server Commands* for details on each command.

Table 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_versions	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.

Command	Description
admin sqm_readers	Displays information about threads that are reading the inbound queue.
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.

Set and Use Threshold Levels

You can configure Replication Server to warn when partitions become too full.

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.

Use **configure replication server** with **sqm_warning_thr1**, **sqm_warning_thr2**, and **sqm_warning_thr_ind** to configure Replication Server to warn when partitions become too full. See *Replication Server Reference Manual > Replication Server Commands > **configure replication server***.

Monitor Partition Percentages

Use the messages in the log file to monitor changes in Replication Server partition percentages.

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

- 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 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 is written 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 a 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:

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

Customize Database Operations

Create and alter functions, function strings, and function-string classes to allow replication definitions to work with database servers other than Adaptive Server.

Functions, Function Strings, and Function-string Classes

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, IBM DB2 UDB, and other 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.

Work with Functions, Function Strings, and Classes

There are several ways you can work with functions and function strings to customize database operations.

You can:

- Create a new function-string class for use with a specific type of database, and customize some or all of the function strings.
- 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**.
- 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-ASE data servers: **rs_iq_function_class**, **rs_db2_function_class**, **rs_mss_function_class**, or **rs_oracle_function_class**. See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Translate Datatypes Using HDS* for detailed information on datatype translations using the heterogeneous datatype support (HDS) feature.

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

See *Replication Server Reference Manual > Replication Server System Functions* for more information about the system functions.

See also

- *Manage Function-String Classes* on page 26
- *Manage Function Strings* on page 31

Functions

Replication Server uses two major types of functions.

The major Replication Server types of functions are:

- System functions
- User-defined functions

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

See also

- *Manage Function Strings* on page 31

System Functions

System functions represent data server operations that use function strings 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.
- 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 also

- *Function Scope* on page 16
- *Summary of System Functions* on page 17

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. See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions*.
- Functions that are used in replicating stored procedures associated with table-replication definitions. You create and maintain user-defined functions of this type yourself.

  You can use asynchronous procedures to replicate stored procedures that are associated with table-replication definitions

User-defined functions have replication-definition scope as the type of function scope.

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 *Replication Server Administration Guide Volume 1 > Manage Replicated Functions > Use Replicated Functions*.

See also

- *Asynchronous Procedures* on page 363
- *Function Scope* on page 16

Function Scope

The scope of a function defines the object to which the function applies: a replication definition or a function-string class.

You must know the scope of a function to determine where to customize a function string at the primary or replicate Replication Server.

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.

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).
- 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.

See *Replication Server Reference Manual > Replication Server System Functions* for complete documentation of all of the system functions.

See also

- *Primary Site for a Function-String Class* on page 29
- *System Functions with Function-String-Class Scope* on page 17
- *System Functions with Replication-Definition Scope* on page 19

Summary of System Functions

Replication Server provides system functions with function-string-class scope and replication-definition scope.

See *Replication Server Reference Manual > Replication Server System Functions* for complete documentation of all of the system functions.

System Functions with Function-String-Class Scope

Replication Server provides several 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. You may also need to assign or change the primary site from one Replication Server to another for a function-string class.

Table 3. 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.
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.
rs_get_lastcommit	Retrieve rows from the `rs_lastcommit` system table.
rs_get_sortorder	Return the sort order used by a data server.

Function Name	Description
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_nohold-lock	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_non_blocking_commit	Coordinates Replication Server non-blocking commit with the corresponding function in the replicate data server. Maps to the **set delayed_commit on** function string in Adaptive Server 15.0 and later, and with the **alter session set commit_write = nowait;** function string in Oracle 10g v2. For all other non-Sybase databases, **rs_non_blocking_commit** maps to null. Executes every time DSI connects to the replicate data server and if the **dsi_non_blocking_commit** value is from 1 to 60. If the value of **dsi_non_blocking_commit** is zero, **rs_non_blocking_commit** does not execute.
rs_non_blocking_com-mit_flush	Ensures that database transactions are flushed to disk when **dsi_non_blocking_commit** is enabled. Maps to the corresponding function string in Adaptive Server 15.0 and later, and Oracle 10g v2 and later. For all other non-Sybase databases, **rs_non_blocking_commit_flush** maps to null. **rs_non_blocking_commit_flush** executes at intervals equal to any number of minutes from 1 to 60 that you specify with **dsi_non_blocking_commit**. **rs_non_blocking_commit_flush** does not execute if the value of **dsi_non_blocking_commit** is zero.
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.

Function Name	Description
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_set_quoted_identifiers	Sets the DSI connection to the data server to allow quoted identifiers to be sent through the connection. Pre-requisites: **dsi_quoted_identifier** must be set to "on" and **rs_set_quoted_identifier** must contain the necessary commands to enable the use of quoted identifiers for the data server. For Adaptive Server and Microsoft SQL Server the command is: **set quoted_identifiers on**.
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.
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.

See also

- *Change the Primary Site for a Function-String Class* on page 29

System Functions with Replication-Definition Scope

Replication Server provides several 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 4. System Functions with Replication Definition Scope

Function name	Description
rs_datarow_for_write-text	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 `raw-object` 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 `raw-object` 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 (remote procedure call) 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 can be replaced with data values. The format used by a function string is determined by the type of data server and how you want Replication Server to interact with it. You can alter output templates to customize function strings.

Function strings are grouped into function-string classes. Each database connection must be assigned a function-string class according to the type of replicate database. Replication Server provides function-string classes that generate default function strings for all actively supported data servers.

When you set up a replication system or add databases to the system, anticipate your function-string requirements and decide how you will use function-string classes and whether you need to customize function strings.

See also

- *Output Templates* on page 33
- *Function-String Classes* on page 22
- *Manage Function Strings* on page 31

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 also

- *Function-string Input and Output Templates* on page 32

Applications for Customized Function Strings

There are several applications for customized function strings.

- 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.
- Alter output templates for existing system function strings to:
  - 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 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 *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Create Replication Definitions > Create Multiple Replication Definitions Per Table*.

System Functions with Multiple Function Strings

You can create multiple function-string instances for the same replication definition for other system functions with replication-definition scope

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 non-ASE data servers, provided you use ECDA to access the various data servers.

System-Provided Classes

Replication Server provides several function-string classes called system-provided classes which contain default function strings for data servers supported by Replication Server.

- **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 *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Translate Datatypes Using HDS > Create Class-Level Translations*.
  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_iq_function_class** – Sybase® IQ function strings are provided for this class. See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Translate Datatypes Using HDS > Create Class-Level Translations*.
- **rs_msss_function_class** – Microsoft SQL Server function strings are provided for this class. See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Translate Datatypes Using HDS > Create Class-Level Translations*.
- **rs_oracle_function_class** – Oracle function strings are provided for this class. See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Translate Datatypes Using HDS > Create Class-Level Translations*.

See also

- *Guidelines for Creating Function Strings* on page 38

- *System Functions with Function-String-Class Scope* on page 17

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_iq_function_class**, **rs_msss_function_class**, and **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.

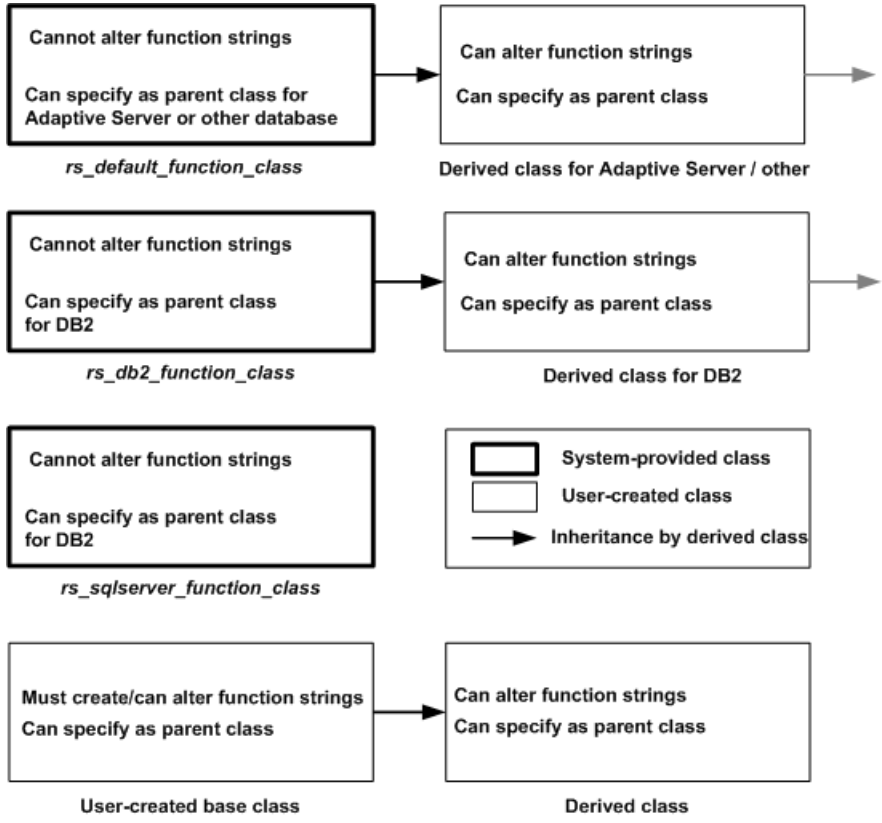
A base class that you have created can be modified to become a derived class, or it can be designated as the parent class for a derived class. A derived class can be modified to inherit function strings from a different parent class, or it can be detached from a parent class and become a base class.

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 this figure.

Figure 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.

Manage Function-String Classes

Managing function-string classes includes creating, assigning, and dropping 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**.

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.

See also

- *Manage Function Strings* on page 31

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.

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.

1. Create the function-string class with **create function string class**.

   Use the appropriate syntax to either:
   - Create a derived class, or
   - Create a base class.

   The name of the new class must conform to the rules for identifiers. See *Replication Server Reference Manual > Topics > Identifiers*.
2. Create function strings for the new class with **create function string**.
   - If you are creating a derived class, you need to 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**, which is a system-provided 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.
   - 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 *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections > Suspend Database Connections*.
4. Create or alter the database connection to assign the new class.
5. If you altered an existing database connection to use the new class, resume the connection.

   See *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections > Suspend Database Connections*.

See also

- *Function-String Inheritance* on page 24
- *Create a Derived Class* on page 28
- *Create a Base Class* on page 28
- *Create Function Strings* on page 38
- *System-Provided Classes* on page 23

- *Assign a Function-String Class to a Database* on page 30

Create a Derived Class

Use the **create function string class** command and specify a parent class to create a derived function-string class that inherits function strings from the parent class.

For example, at the primary site of the parent, enter:

```
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.

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.

See also

- *Function-String Inheritance* on page 24

Create a Base Class

Use the **create function string class** command without specifying a parent class to create a base function-string class, which is one that does not inherit function strings from a parent class.

For example, enter:

```
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.

Change 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 **move primary** to assign one. You must use **create function string class** to first designate a primary site for that class.

See also

- *Primary Site for a Function-String Class* on page 29

Assign 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 be created and be available to the Replication Server.
- All necessary function strings must be created in the class.

Note: When you create a connection using a connection profile, the function string class is assigned by the connection profile.

See *Replication Server Administration Guide Volume 1 > Manage Database Connections > Create Database Connections* and *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections* for more information about using the **create connection** and **alter connection** commands, and connection profiles. Also refer to reference pages for these commands in the *Replication Server Reference Manual > Replication Server Commands*.

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
```

See also

- *Create Function Strings* on page 38
- *Creating a Function-String Class* on page 27

Drop a Function-String Class

Use the **drop function string class** command to drop a function-string class that you created, from the replication system, if you are sure that you will not need the function-string class again.

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.

For example, to drop the **tokyo_func_class** function-string class and all of its function strings, enter at the **isql** command line:

```
drop function string class tokyo_func_class
```

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

See *Replication Server Reference Manual > Replication Server Commands > **drop function string class***.

Manage 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.

See *Replication Server Administration Guide Volume 1 > Replication Server Technical Overview* for more information about DSI threads, the components that perform this conversion at the replicate Replication Server.

See *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.

- 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.

See also

- *Function-String Classes* on page 22

Function-string Input and Output Templates

To customize function strings, you alter their input and 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

There are several requirements for altering templates to customize function strings.

Requirements include:

- 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 also

- *Output Templates* on page 33

Output Templates

Replication Server uses output templates to determine the format of the command sent to a data server. You can alter output templates to customize function strings.

Most output templates use one of three formats: language, RPC or **none**, corresponding to the format of the function string itself.

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 also

- *Function Strings* on page 20
- *Use Function Strings with text, unitext, image, and rawobject Datatypes* on page 49

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 commands to the data server to process.

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 also

- *Create Function Strings* on page 38
- *Function-string Variables* on page 36
- *Function-string Variable Formatting* on page 37

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 That Use the none Parameter

You can increase function-string efficiency when you create or alter function strings by using the **none** parameter to identify class-level and table-level function strings that do not have output commands. Replication Server does not execute these function strings on replicate databases.

Output Templates for rs_writetext Function Strings

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

See also

- *Use Function Strings with text, unitext, image, and rawobject Datatypes* on page 49

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 *Replication Server Administration Guide Volume 1 > Manage Subscriptions > Subscription Materialization Methods* 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 *Replication Server Administration Guide Volume 1 > Manage*

Subscriptions > Subscription Commands > drop subscription Command 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.

Determining Where to Create Function Strings

Determine the 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 this 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*?.

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 *Replication Server Reference Manual* for complete syntax.

See also

- *Function-string Variables* on page 36
- *Create Function Strings* on page 38

Function-string Variables

You can use variables embedded in function-string input or output templates as 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?
```

See also

- *Default System Variable* on page 47

Function-string Variable Modifiers

Replication Server recognizes several function-string variable modifiers.

Table 5. 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.

See *Replication Server Reference Manual > Replication Server Commands > create function string* for a list of system-defined variables that you can use in function-string input or output templates.

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*.

Create Function Strings

Use the **create function string** command to add a function string to a function-string class.

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.
- 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.

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.

See also

- *Primary Site for a Function-String Class* on page 29

Guidelines for Creating Function Strings

There are several 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**.
  For **rs_db2_function_class**, **rs_iq_function_class**, **rs_msss_function_class**, and **rs_oracle_function_class**, you:
  - Cannot use function-string class scope system functions, such as **rs_begin** to create customized class-level function strings
  - Can use replication definition scope system functions such as **rs_insert** to create customized table-level function strings
- 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 multiple commands in a language output template, separating them with semicolons.
- You can batch commands for non-ASE servers.

  Make sure that the database connection **batch** parameter has been set to allow command batching. See *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections > Set and Change Parameters Affecting Physical Connections > Change Parameters Affecting a Single Connection*.
- You can use Adaptive Server syntax to specify a null value for a *constant* in a function string.
- You can increase function string efficiency when you create or alter function strings by using the **none** parameter to identify class-level and table-level function strings that do not have output commands. Replication Server does not execute these function strings on replicate databases.

See *Replication Server Reference Manual > Replication Server Commands >* ***create function string*** for the complete syntax.

See also

- *Define Multiple Commands in a Function String* on page 43
- *Command Batching for Non-ASE Servers* on page 44

Create Function Strings Examples

Learn from the examples showing how to create function strings.

- Example for rs_begin Function String

  This example shows you how to create 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.

  This example shows you how to create 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?'
```

Alter 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.
- 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.

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 *Replication Server Reference Manual > Replication Server Commands > **alter function string*** for the complete syntax.

See also

- *Primary Site for a Function-String Class* on page 29
- *Create Function Strings* on page 38

Drop 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.

See *Replication Server Reference Manual* > *Replication Server Commands* > ***drop function string***.

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 also

- *Restore Default Function Strings* on page 42

Drop Function Strings Examples

Learn from the examples showing how to drop function strings.

Drop Function String for the Replication Definition

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
```

Drop Instance of a Function String for the Replication Definition

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
```

Drop Function String from Function-string 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
```

Restore 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.

See *Replication Server Reference Manual > Replication Server Commands*, 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.

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
```

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 also

- *Drop Function Strings* on page 41

- *Alter Function Strings* on page 40
- *Create Function Strings* on page 38

Create 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 none
```

See also

- *Alter Function Strings* on page 40

Define Multiple Commands in a Function String

You can use function strings to batch commands for database servers.

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.

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'
```

Command Batching for Non-ASE Servers

Replication Server lets you batch commands for non-ASE database servers, which may improve performance.

Support for command batching requires:

- Using the two function strings, **rs_batch_start** and **rs_batch_end**.
- Using the DSI connection parameters to control the processing of the two function strings.

Function Strings to Support Command Batching

Support for command batching to non-ASE servers is achieved through the use of two function strings, **rs_batch_start** and **rs_batch_end**.

These function strings store the SQL translation needed for marking the beginning and end of command batches. Use of these function strings is not necessary for ASE or any other data server where the function strings **rs_begin** and **rs_commit** already support the needed functionality

Connection Settings to Support Command Batching

The **use_batch_markers** DSI connection parameter is used to control the processing of the **rs_batch_start** and **rs_batch_end** function strings.

Set **use_batch_markers** with **alter connection** and **configure connection**. If **use_batch_markers** is set to on, the **rs_batch_start** and **rs_batch_end** function strings are executed. The default is off.

Note: Set **use_batch_markers** on only for replicate data servers that require additional SQL to be sent at the beginning and end of a batch of commands that are not contained in the **rs_begin** function string.

Order of Processing

When you configure a connection to use the batch marker function strings, Replication Server sends statements to the data server in a certain order.

1. The **rs_begin** command is sent to the replicate data server first, either separately or grouped with the batch of commands, based on the configuration parameter **batch_begin** as it is with current functionality.
2. The **rs_batch_start** command is processed and sent only when **use_batch_markers** is configured to true.

The **rs_batch_start** marker is grouped with the commands being sent as a batch. Valid **rs_begin** and **rs_batch_start** function strings allows processing of both single and batched transactions to the data servers.

3. A batch of commands is sent to the replicate data server.
   The size of the batch varies, and sending of the batch follows the existing rules for terminating the grouping and flushing of the commands to the replicate data server. These commands contain a command separator between each individual command.
4. The **rs_batch_end** command is the last command in the batch of commands. The **rs_batch_end** marker is sent only when the configuration parameter **use_batch_markers** is set to true.
   The **rs_batch_start**, a batch of commands, and **rs_batch_end** may be repeated if more than one batch is required when commands have been flushed by limits such as **dsi_cmd_batch_size**.
5. After the final **rs_batch_end** command has been sent, the **rs_commit** command is sent to the replicate data server. The **rs_commit** is processed according to the present rules.

DSI Configuration

There are several DSI configuration parameters that you need to consider for each connection that will be batching commands

- **batch**
- **batch_begin**
- **use_batch_markers**

See the *Replication Server Heterogeneous Replication Guide* to determine whether command batching is allowed for your non-ASE replicate data server.

See the *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections > Set and Change Parameters Affecting Physical Connections > Configuration Parameters Affecting Physical Database Connections* and *Replication Server Reference Manual > Replication Server Commands >* ***alter connection*** to use the configuration parameters.

Use 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 for Adaptive Server, 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.

See *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections > Set and Change Parameters Affecting Physical Connections > Configuration Parameters Affecting Physical Database Connections* for more information about **batch**.

Display Function-Related Information

Use the Replication Server **admin** command or Adaptive Server stored procedures to obtain information about existing function strings and classes in your replication system.

See *Replication Server Reference Manual > Replication Server Commands* for more information on **admin** command.

Obtain Information Using the admin Command

You can display the names of the function-string classes used in your Replication Server system using one of the Replication Server **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
...
```

See *Replication Server Reference Manual > Replication Server Commands > **admin show_function_classes***.

Obtain 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.

See *Replication Server Reference Manual > RSSD Stored Procedures* for more information.

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]
```

Default System Variable

Use the default system variable, *rs_default_fs*, to extend and customize function strings.

- 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.

See *Replication Server Reference Manual* > *Replication Server Commands* > ***create function string*** for a complete list of function string system variables.

Extend 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, an error is raised.

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.

Use the replicate minimal columns Clause

Customize **rs_update** and **rs_delete** function strings and continue to use the **replicate minimal columns** clause.

If you have specified the **replicate minimal columns** clause 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 *Replication Server Reference Manual > Replication Server Commands >* ***create replication definition*** for more information about the **replicate minimal columns** clause.

Use 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**.

See *Replication Server Reference Manual > Replication Server System Functions >* ***rs_writetext***.

For Replication Server version 11.5 or later, you can use multiple replication definitions instead of function strings. See *Replication Server Administration Volume 1 > Manage Replicated Tables > Create Replication Definitions > Create Multiple Replication Definitions Per Table*.

Use the writetext Output Template Option 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 these 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.

Use the none Output Tempate 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, providing 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 non-ASE 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.

Manage Warm Standby Applications

Set up, configure, and monitor a warm standby application between two 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.

Replication Server supports setting up and managing warm standby applications for Adaptive Server and Oracle databases. See *Replication Server Heterogeneous Guide > Heterogeneous Warm Standby for Oracle* for detailed information on how to set up and configure a warm standby application between two Oracle databases.

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 *Replication Server Administration Guide Volume 1 > Manage Replicated Objects Using Multisite Availability*.

Warm Standby Applications

A warm standby application is a pair of 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.

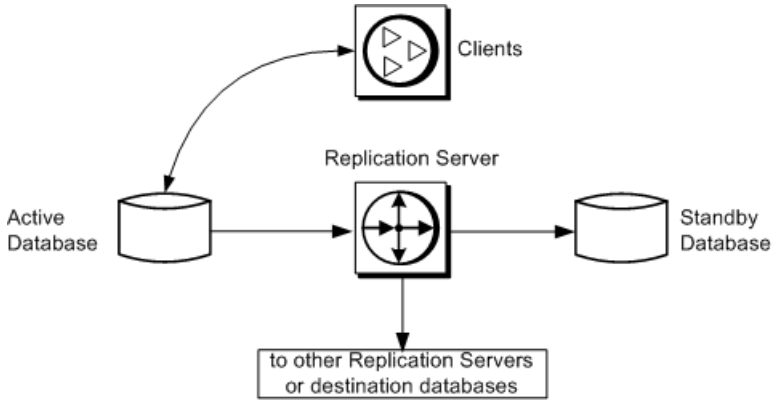
To keep the standby database consistent with the active database, Replication Server reproduces transaction information retrieved from the active database's transaction log. Although replication definitions facilitate replication into the standby database, they are not required. Subscriptions are not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

How a Warm Standby Works

Learn how a warm standby works.

This figure illustrates the normal operation of an example warm standby application.

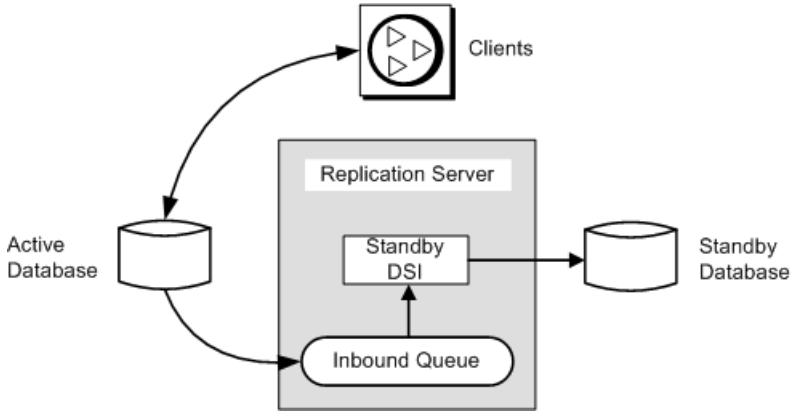
Figure 2: Warm Standby Application—Normal Operation



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.

Figure 3: Warm Standby Application Example—Before Switching



This figure shows details about the components and processes in a warm standby application.

See also

- *Before Switching Active and Standby Databases* on page 85

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 these 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 *Replication Server Administration Guide Volume 1 > Manage Database Connections*.

To improve replication performance, you can create alternate connections and alternate logical connections in warm standby environments.

See also

- *Multiple Replication Paths for Warm Standby Environments* on page 261
- *Setting Up ASE Warm Standby Databases* on page 70

Primary and Replicate Databases and Warm Standby Applications

A logical database may also function as a primary or a replicate database.

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 replicate database receives data from the primary 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 database that does not participate in replication and exists solely as a warm standby backup, or the logical database may also function as a primary or a replicate database.

Comparison of Database Relationships

Usually, databases are defined as “primary” or “replicate.” In discussing warm standby applications, however, databases are also defined as “active” or “standby.”

Table 6. 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.
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.

See also

- *Warm Standby Applications Using Replication* on page 100

Warm Standby Requirements and Restrictions

There are several requirements and restrictions that apply to all Replication Server warm standby applications.

- You must use a data server such as Adaptive Server, that supports warm standby applications.
- One Replication Server manages both the active and standby databases. Both the active and standby databases must be Adaptive Server databases. See *Replication Server Heterogeneous Guide > Heterogeneous Warm Standby for Oracle* for detailed information on how to set up and configure a warm standby application between two Oracle databases.
- You cannot create a standby database for the RSSD. You can create a standby database for the `master` database only if the Adaptive Server supports master database replication, such as Adaptive Server 15.0 ESD #2 and later.
- Replication Server does not switch client applications to the standby database.
- 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.
- The commands and procedures for abstract plans are replicated, except for:
  - 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.
- Failover support is not a substitute for warm standby. While warm standby keeps a copy of a database, Sybase Failover accesses the same database from a different machine. Failover support works the same for connections from Replication Server to warm standby databases.

  See *Using Sybase Failover in a High Availability System* in the Adaptive Server Enterprise documentation set.

- You cannot use the dump and enable marker on the active database and then use cross-platform **dump** and **load** to rebuild the standby database. The Replication Agent must send the dump marker to the standby database you are rebuilding. During the cross-platform dump and load, the active database must be in single-user mode when you obtain the dump from the active database.

See also

- *Set up Clients to Work with the Active Data Server* on page 93
- *Configure the Replication System to Support Sybase Failover* on page 310
- *Cross-Platform Dump and Load* on page 76

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**.

Replicated Information for Warm Standby

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:

- You can copy the execution of user stored procedures to the standby database by marking the stored procedures with the **sp_setrepproc** system procedure. Normally, only stored procedures associated with function replication definitions are replicated to standby databases.

For detailed information on what information is replicated for Oracle warm standby, see *Replication Server Heterogeneous Guide > Heterogeneous Warm Standby for Oracle*.

See also

- *Use sp_setrepproc to Copy User Stored Procedures* on page 66

Comparison of Replication Methods

Compare **sp_reptostandby** and **sp_setreptable**, to learn how each copies information to the standby database.

Table 7. 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.	Allows replication of DML commands executed on marked tables. **Note:** You can force replication of supported DDL operations for an **isql** session.
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.	If you use Adaptive Server databases, copies execution of **truncate table** to standby databases. No subscription is needed.

sp_reptostandby	sp_setreptable
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.
By default, `text`, `unitext`, `image`, and `rawobject` columns are copied to the standby database only if changed. 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.	By default, `text`, `unitext`, and `image` columns are always copied to the standby database. 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.	

See also

- *Force Replication of DDL Commands to the Standby Database* on page 69
- *Replicate Truncate Table To Standby Databases* on page 97
- *Replication of text, unitext, image, and rawobject Data in Warm Standby Applications* on page 67
- *Supported DDL Commands and System Procedures* on page 61

Use sp_reptostandby to Enable Replication

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

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

```
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.

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

See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***sp_reptostandby***.

Restrictions and Requirements when Using sp_reptostandby

Consider these restrictions and requirements 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. See the *Adaptive Server Enterprise System Administration Guide*.
- The active database name must exist in the standby server. Otherwise, replication of commands or procedures containing the name of that database fails.
- 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. Make the server user's IDs match, and add login information to the destination Replication Server.
- Some commands are not copied to the standby database:
  - **select into**
  - **update statistics**
  - Database or configuration options such as **sp_dboption** and **sp_configure**
- Replication Server does not support the replication of DDL commands after **set proxy** is executed on the primary Adaptive Server; Replication Server returns error 5517:

  ```
  A REQUEST transaction to database '...' failed because the
  transaction    owner's password is missing. This prevents the
  preservation of transaction ownership.
  ```

See also

- *Making the Server User's IDs Match* on page 79

Supported DDL Commands and System Procedures

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 for which replication is supported for Adaptive Server 12.5 and later.

The supported DDL commands are:

- **alter encryption key**
- **alter key**
- **alter table**
- **create default**
- **create encryption key**
- **create function**
- **create index**
- **create key**
- **create plan***
- **create procedure**
- **create rule**
- **create schema***
- **create table**
- **create trigger**
- **create view**
- **drop default**
- **drop encryption key**
- **drop function**
- **drop index**
- **drop procedure**
- **drop rule**
- **drop table**
- **drop trigger**
- **drop view**
- **grant**
- **installjava*** – replication of **installjava** is not supported for MSA environments.
- **remove java***
- **revoke**

The supported system procedures are:

- **sp_add_qpgroup***
- **sp_addalias**
- **sp_addgroup**
- **sp_addmessage**
- **sp_addtype**
- **sp_adduser**
- **sp_bindefault**
- **sp_bindmsg**
- **sp_bindrule**

- **sp_cachestrategy**
- **sp_changegroup**
- **sp_chgattribute**
- **sp_commonkey**
- **sp_config_rep_agent**
- **sp_drop_all_qplans***
- **sp_drop_qpgroup***
- **sp_dropalias**
- **sp_dropgroup**
- **sp_dropkey**
- **sp_dropmessage**
- **sp_droptype**
- **sp_dropuser**
- **sp_encryption**
- **sp_export_qpgroup***
- **sp_foreignkey**
- **sp_hidetext**
- **sp_import_qpgroup***
- **sp_primarykey**
- **sp_procxmode**
- **sp_recompile**
- **sp_rename**
- **sp_rename_qpgroup***
- **sp_replication_path**
- **sp_setrepcol**
- **sp_setrepdefmode**
- **sp_setrepproc**
- **sp_setreplicate**
- **sp_setreptable**
- **sp_unbindefault**
- **sp_unbindmsg**
- **sp_unbindrule**

The set of DDL commands and system procedures that are supported for replication in the master database is different than the set supported from replication in a user database.

If the database is the master database, the supported DDL commands are:

- **alter role**
- **create role**
- **drop role**

- **grant role**
- **revoke role**

If the database is the master database, the supported system procedures are:

- **sp_addlogin**
- **sp_defaultdb**
- **sp_defaultlanguage**
- **sp_displaylevel**
- **sp_droplogin**
- **sp_locklogin**
- **sp_modifylogin**
- **sp_password**
- **sp_passwordpolicy** – replicated for all options except **allow password downgrade**.
- **sp_role**

Replication of 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, execute either of these statements at the primary database:

- ```
  alter table table_name
          ...
          replace column_name default null
  ```
- ```
  alter table table_name
          ...
          drop constraint constraint_name
  ```

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

Replication of the Master Database: Limitations

User tables and user stored procedures are not replicated from the master database.

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

- **sp_addlogin**
- **sp_defaultdb**

- **sp_defaultlanguage**
- **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 Adaptive Server and the target Adative 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).

Disable Replication

Use **sp_reptostandby** with the **none** option to turn off data and schema replication.

Log in to Adaptive Server and at the **isql** prompt, enter:

```
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: To turn off replication only for the current **isql** session, use the **set replication** command.

Also, if the database is marked for replication to use indexes on `text`, `unitext`, `image`, and `rawobject` columns, **sp_reptostandby** *dbname,* **'none'** also drops indexes for replication on tables not explicitly marked for replication.

See also

- *Change Replication for the Current isql Session* on page 69

Use 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.

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

See also

- *Change Replication for the Current isql Session* on page 69

Use 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 Server copies DML changes effected by the procedure to the standby database, if the affected tables have been marked for replication.

See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions* for more information about the **sp_setrepproc** system procedure.

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

Replication of 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.

See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Mark Tables for Replication > Use the sp_setreptable System Procedure > Enable Replication with* ***owner_on*** *Status*.

Note: If you mark a table with a nonunique name for replication, then create a replication definition for it, you must include owner information in the replication definition. Otherwise, Replication Server cannot find the correct table in the replicate or standby database.

Replication of text, unitext, image, and rawobject Data in Warm Standby Applications

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. Make sure the replication status is the same at the Adaptive Server and the Replication Server.

Use the use_index Option in a Replicate Database

Use the **use_index** option 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.

Configure Warm Standby Database for SQL Statement Replication

By default, warm standby applications do not replicate the DML commands that support SQL statement replication. However, there are several ways you can use SQL statement replication.

- Create table replication definitions using **replicate SQLDML** and **send standby** clauses.
- Set the **ws_sqldml_replication** parameter on. The default value is **UDIS**. However, **ws_sqldml_replication** has a lower precedence than the table replication definition for SQL replication. If your table replication definition contains **send standby** clause for a table, the clause determines whether or not to replicate the DML statements, regardless of the **ws_sqldml_replication** parameter setting.

Replication of Encrypted Columns

Considerations when working with encrypted columns in warm standby applications are similar to non-warm standby environments.

See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Replicate Encrypted Columns*.

Replication of Quoted Identifiers

When replicating to a warm standby database and to replication definition subscribers, and the primary table name is marked as quoted but the replicate table name is not, or vice-versa, Replication Server sends both the primary table name and the replicate table name as quoted.

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 the `text`, `unitext`, `image`, and `rawobject` columns of the table 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.

To copy only `text`, `unitext`, `image`, and `rawobject` columns that have changed, use **sp_setrepcol** to set the replication status to **replicate_if_changed**.

Change Replication for the Current isql Session

You can use **set replication** to control replication of DML and 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 8. 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.**

See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***set replication***.

Force Replication of DDL Commands to the Standby Database

Use **set replication force** to force replication of supported DDL commands and system procedures.

For example, 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
```

Turn off All Replication to the Standby Database

Use **set replication force off** to turn 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 ASE Warm Standby Databases

Setting up databases for a warm standby application involves several high-level 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 need not 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

There are several prerequisites for setting up ASE warm standby databases.

- 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 also

- *Warm Standby Requirements and Restrictions* on page 57

Client Application Issues

There are several client application issues to consider before you set up warm standby databases.

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, implement a mechanism for switching client applications to the new active database.

See also

- *Set up Clients to Work with the Active Data Server* on page 93

Task One: Creating the Logical Connection

Create the logical connection and reconfigure RepAgent for the active database if the active database is already part of the replication system.

See also

- *Database Connections in a Warm Standby Application* on page 55

Name the Logical Connection

The name you assign to the logical connection depends on whether the active database has been added to the replication system.

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

- 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 also

- *Warm Standby Applications Using Replication* on page 100

Procedure for Creating the Logical Connection

Use the **create logical connection** command to create the connection from Replication Server.

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

Reconfigure and restart RepAgent after you create the logical connection.

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.

   See *Replication Server Administration Guide Volume 1 > Manage RepAgent and Support Adaptive Server > Set up RepAgent* and *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***sp_config_rep_agent***.
2. Restart RepAgent.

Task Two: Add the Active Database

Use **rs_init** to add a database to the replication system as the active database for a warm standby application.

Perform the steps for adding a database to the replication system as described in the Replication Server installation and configuration guides for your platform.

Task Three: Enabling Replication for Objects in the Active Database

Use **sp_reptostandby** to enable replication of stored procedures, and **sp_reptostandby** or **sp_setreptable** to enable replication for tables in the active database.

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

- **sp_reptostandby** to mark the database for replication, enabling replication of data and supported schema changes.
- **sp_setreptable** to mark individual tables for replication of data changes.

1. Log in to the Adaptive Server as the system administrator or as the database owner, and execute:

   ```
   use active_database
   ```
2. Mark database tables for replication, using one of three 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 for replication of data changes 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. Execute **sp_setrepproc** with the relevant parameter for every stored procedure which has executions you want to replicate into the standby database.

   - If you are using the replicated functions feature described in *Replication Server Administration Guide Volume 1 > Manage Replicated Functions*, execute **sp_setrepproc** with the **'function'** parameter:

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

   - If you are using asynchronous procedures such as replicated stored procedures associated with table replication definitions, execute **sp_setrepproc** with the **'table'** parameter:

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

See also

- *Replicated Information for Warm Standby* on page 58
- *Asynchronous Procedures* on page 363

Enable Replication for Objects Added Later

Mark and add new tables and user stored procedures for replication 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 the standby database with data from the active database.

After you add the standby database to the replication system, you must prepare it for operation.

You can then enable replication for objects in the standby database and grant 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.

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. Bring the new database online using the **online database** clause before replicating.

Create 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.

Determine How to Initialize the Standby Database

Initialize the standby database with data from the active database.

Use these Adaptive Server commands and utilities to initialize the standby database:

- **dump** and **load**, or
- **bcp**, or
- **quiesce database ... to** *manifest_file* to generate the manifest file and **mount** to copy the data into the standby database.

See the *Adaptive Server Enterprise Reference Manual: Commands*.

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 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.

Summary of Database Initialization Methods

Consider the issues for each of the initialization methods and the role of these markers.

Table 9. 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.)

Issue	Use **dump** and **load** with "dump marker"	Use **dump** and **load** without "dump marker"	Use **bcp**	Use **mount**
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.
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.

Cross-Platform Dump and Load

You can use cross-platform dump and load to initialize a standby database with a RepAgent.

1. On the active database:
   a) Stop the RepAgent with **sp_stop_rep_agent** *database*.
   b) Remove the secondary truncation point with **dbcc settrunc('ltm', 'ignore')**.
   c) Set the database in single-user mode in Adaptive Server.

      Enter:

      ```
      sp_dboption database_name, 'single user', true
      ```

   d) Checkpoint the database.

      Enter:

      ```
      checkpoint
      ```

   e) Dump the database transaction log by executing in Adaptive Server:

      ```
      dump tran database_name with truncate_only
      go
      ```

   f) Obtain a dump of the database.
2. On the standby database:
   a) Load the dump you obtained from the standby database.

Sybase recommends that you run **sp_post_xpload** to check and rebuild indexes even if the endian types of the platforms are the same.

b) Dump the transaction log to delete the log records that **sp_post_xpload** creates:

```
dump tran database_name with truncate_only
go
```

c) Execute the Adaptive Server **sp_indsuspect** system procedure to check user tables for indexes marked as suspect.

d) Rebuild suspect indexes if required. If there is a change in characterset or sort order, you must execute **sp_indsuspect** and rebuild indexes again until sp_indsuspect does not show any tables with suspect indexes.

e) Execute **dbcc settrunc ('ltm', 'valid')** to restore the secondary truncation point in the database log followed by **rs_zeroltm** to reset the database locater value to zero.

Executing these commands allows RepAgent to start at the secondary truncation point.

f) Start RepAgent with **sp_start_rep_agent** *database*.

See "Dumping and loading databases across platforms," in Chapter 11 "Developing a Backup and Recovery Plan" in the *Adaptive Server Enterprise System Administration Guide Volume 2*.

If You Do Not Suspend Transaction Processing

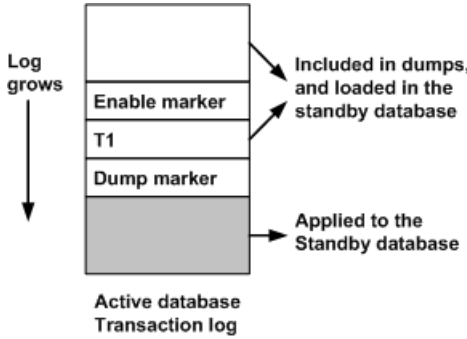
If you do not suspend transaction processing for the active database while initializing the standby database, choose the "dump marker" option when you add the standby database.

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 this figure, 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 4: 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.

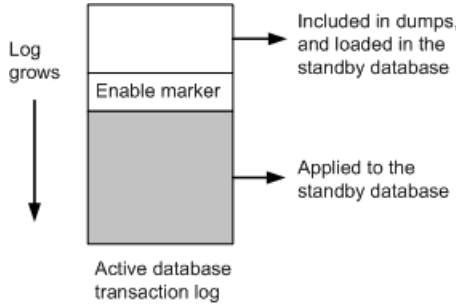
If You Suspend Transaction Processing

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 5: Using dump and load Without Dump Marker, or Using bcp



As shown in the figure, 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.

Add 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 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.

Make the server user's IDs match using one of three methods:

- 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

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.

   See 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

Initialize the standby database, bring it online, and resume the connection to it, to add it to the replication sysem.

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. You must resume the connection to the active database.

   In the Replication Server enter:

   ```
   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.

   In the Replication Server enter:

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

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

Use a Blocking Command for Standby Creation

Use the **wait for create standby** Replication Server blocking command to instruct 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
```

Enable Replication for Objects in the Standby Database

To be ready to switch to the standby database, you must enable replication 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 enable replication for objects in the standby database, adapt the procedure for enabling replication for objects in the active database.

See also

- *Task Three: Enabling Replication for Objects in the Active Database* on page 72

Enable 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.

1. Log in to the Adaptive Server as the System Administrator or as the Database Owner, and specify the database with which you want to work.

   Enter:

   ```
   use standby_database
   ```

2. Grant **replication_role** to the maintenance user.

   Enter:

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

   replication_role ensures that the maintenance user can execute **truncate table** at the standby database.

3. Execute the **grant all** command for each table.

   Enter:

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

Replication of the Master Database in a Warm Standby Environment for ASE

There are several requirements and restrictions for replicating the master database in an Adaptive Server 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.

There are several restrictions and requirements when you set up your warm standby application and enable replication with **sp_reptostandby**, and there are several supported DDL and system procedures that apply to the master database.

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 *Replication Server Administration Guide Volume 1 > Manage Replicated Objects Using Multisite Avaiability > Replicating the Master Database in an MSA Environment* for information about master database replication in an MSA environment.

See also

- *Restrictions and Requirements when Using sp_reptostandby* on page 61
- *Supported DDL Commands and System Procedures* on page 61

Setting Up Master Database Replication in a Warm Standby Environment

Set 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 use “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.
2. Mark the master database on both the active and the standby database to send system procedures.

   Enter:

   ```
   sp_reptostandby master, ‘all’
   ```
3. Stop the RepAgent on the active master database.

   Enter:

   ```
   sp_stop_rep_agent master
   ```
4. Configure the Replication Agents on both the active and the standby databases to send warm standby transactions.

   Enter:

```
sp_config_rep_agent master, ‘send warm standby
xacts’, ‘true’
```

5. Restart the RepAgent on the active master database.

   Enter:

   ```
   sp_start_rep_agent master
   ```

6. Resume the DSI connections to both the active and the standby master databases on the Replication Server.

   Enter:

   ```
   resume connection to active_ds.master
   go
   resume connection to standby_ds.master
   go
   ```

7. Verify the status of warm standby.

   Enter:

   ```
   admin logical_status
   ```

See also

- *Setting Up ASE Warm Standby Databases* on page 70

Switch the Active and Standby ASE Databases

You can switch to the standby database when the active database fails or when you want to perform maintenance on the active database.

Determine 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.

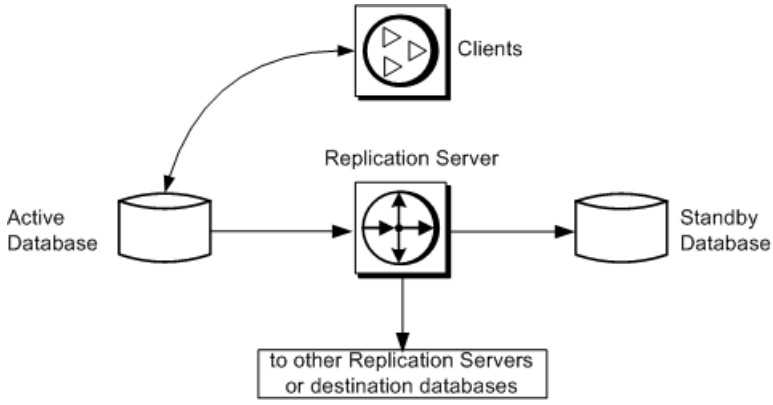
You may also want to switch the roles of the active and standby databases to perform planned maintenance on the active database or its data server.

Before Switching Active and Standby Databases

Learn the processes involved and the status of the components in a warm standby enviroment before you switch from the active to the standby database.

This figure illustrates the normal operation of an example warm standby application.

Figure 6: Warm Standby Application



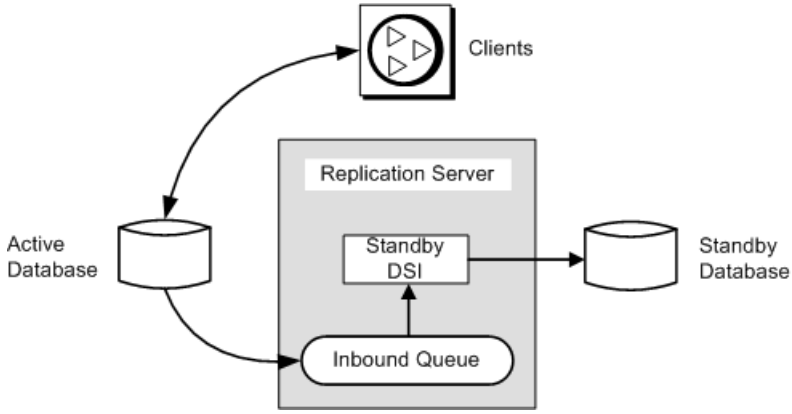
The "Warm Standby Application Example—before Switching" figure:

- 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.
- Represents the warm standby application in normal operation, before you switch the active and standby databases.
- Adds internal detail to show that:
  - Replication Server writes transactions received from the active database into an inbound message queue.

    See *Replication Server Administration Guide Volume 1 > Replication Server Technical Overview > Transaction Handling with Replication Server > Distributed Concurrency Control* 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.

Figure 7: Warm Standby Application Example—before Switching



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 also

- *Warm Standby Applications Using Replication* on page 100

Internal Switching Steps

When you switch active and standby databases, Replication Server performs several tasks.

Replication Server:

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.

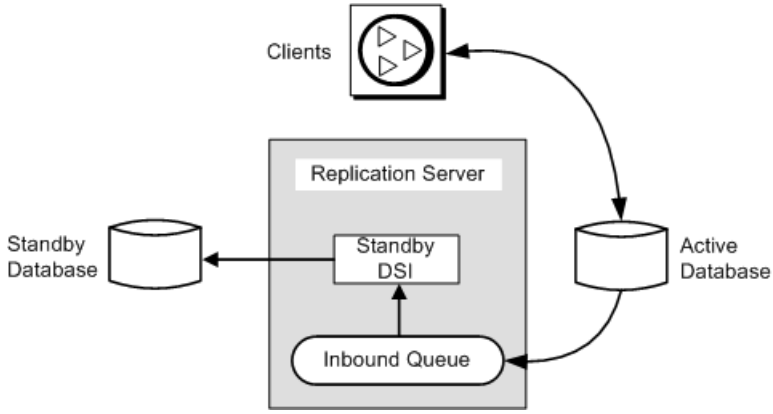
   All committed transactions in the inbound queue must be processed before the switch can complete.
3. Suspends the standby DSI.
4. Enables the secondary truncation point in the new active database.
5. Places a marker in the transaction log of the new active database. Replication Server uses this marker 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

Learn the processes involved and the status of the components in a warm standby environment after you switch from the active to the standby database.

After you have switched the roles of the active and standby databases, the replication system will have changed, as shown in this figure:

Figure 8: 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: After switching, the Replication Agent for the previous active database has shut down, and the Replication Agent for the new active database has started.

Making the Switch

Making the switch from the active to the standby database consists of several tasks.

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

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 also

- *Set up Clients to Work with the Active Data Server* on page 93

Switching the Active and Standby Databases

Learn the procedure to switch the active and standby databases for a logical connection.

Prerequisites

Before switching, you must set up clients to work with the active data server.

Task

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. Execute the switch active command at the Replication Server.

   Enter:

   ```
   switch active for logical_ds.logical_db
   to data_server.database
   ```

   data_server.database is the new active database.
3. Use **admin logical_status** to monitor the progress of a switch.

   Enter:

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

   See the `Operation in Progress` and `State of Operation in Progress` output columns for the switch status.
4. When the active database switch is complete, you must restart RepAgent for the new active database.

   Enter:

   ```
   sp_start_rep_agent dbname
   ```

Next

Note: If Replication Server stops in the middle of switching, the switch resumes after you restart Replication Server.

See also

- *Set up Clients to Work with the Active Data Server* on page 93
- *Internal Switching Steps* on page 86

Use a Blocking Command for Switch Active

Use the **wait for switch** Replication Server blocking command to instruct 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
```

Monitor 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 **admin logical_status** 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 also

- *Commands for Monitoring Warm Standby Applications* on page 92

Abort 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.

The syntax is:

```
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 also

- *Set up Clients to Work with the Active Data Server* on page 93

Resolve 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 **drop connection**, and add it again later as the new standby database. If you drop the database, any queued messages for the database are

deleted. See *Replication Server Reference Manual > Replication Server Commands > **drop connection***.

Bring 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.

Enter:

```
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.

Monitor a Warm Standby Application

You can use the Replication Server log file or several commands to monitor a warm standby application between two Adaptive Server databases or Oracle databases.

Replication Server Log File

You can read the Replication Server log file for messages pertaining to warm standby operations such as messages you 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 Replication Agent, 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.

See *Replication Server Reference Manual > Replication Server Commands* 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
```

Set up Clients to Work with the Active Data Server

You must devise a method to switch client applications when you switch the active and standby databases in Replication Server using the **switch active** command, as Replication Server does not switch client applications to the new active data server and database automatically.

There are 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 for data server name, host name, and port number.

Table 10. 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.

Alter Warm Standby Database Connections

Learn the 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.

Alter Logical Connections

Use the **alter logical connection** command to modify parameters for warm standby logical database connections.

Use **alter logical connection** 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

Change Parameters Affecting Logical Connections

Use the **alter logical connection** command 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.

Configuration Parameters Affecting Logical Connections

There are several parameters you can use to configure logical connections.

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 11. Configuration Parameters Affecting Logical Connections

logical_database_param	*value*
deferred_name_resolution	Enable deferred name resolution in Replication Server to support deferred name resolution in Adaptive Server. You must ensure that deferred name resolution is supported in the replicate Adaptive Server before you enable deferred name resolution support in Replication Server. Default: off **Note:** This parameter is only applicable to Adaptive Server.
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, or if there is no replication definition at all. Otherwise, Replication Server uses the value of the “replicate minimal columns” or “replicate all columns” parameter in the replication definition. Default: on When you set **dsi_compile_enable** to ‘on’, Replication Server ignores what you set for **replicate_minimal_columns**.
save_interval	Specifies the save interval which is the number of minutes that the Replication Server saves messages after they have been successfully passed to the destination data server. Default: 0 minutes

logical_database_param	*value*
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**

See also

- *Save Interval for Recovery* on page 313

Disable the Distributor Thread

Use the **alter logical connection** command to disable 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.

Replicate Truncate Table To Standby Databases

Use the **alter logical connection** command to enable or disable replication of the **truncate table** command.

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."

Alter Physical Connections

Use the **alter connection** command at the source Replication Server to modify parameters that affect physical connections for warm standby applications.

The syntax is:

```
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 *Replication Server Administration Guide Volume 1 > Manage Database Connections > Altering Database Connections*.

Configure Triggers in the Standby Database

You can use the **alter connection** command to configure a connection to fire or not fire triggers.

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 **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."

Configure Replication in the Standby Database

Set the **dsi_replication** configuration parameter to specify 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.

Change Configuration Parameters in the Standby Database

When you create the standby database, several configuration parameters, if they are set for the active database, are copied from the active database to the standby database. You can change the setting of any of these configuration parameters.

Table 12. Configuration Parameters Copied to Standby Database

batch	**batch_begin**	**command_retry**
db_packet_size	**dsi_cmd_separator**	**dsi_charset_convert**
dsi_cmd_batch_size	**dsi_keep_triggers**	**dsi_fadeout_time**
dsi_isolation_level	**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_xact_group_size**	**dsi_serialization_method**
dsi_sqt_max_cache_size	**dsi_xact_in_group**	**dump_load**
parallel_dsi	**use_batch_markers**	

See *Replication Server Administration Guide Volume 1 > Manage Database Connections*.

Drop 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 *Replication Server Administration Guide Volume 1 > Manage Database Connections > Drop Database Connections* 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
```

Drop 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

Learn about warm standby applications that involve replication, where the logical database serves as a primary or replicate database in the replication system.

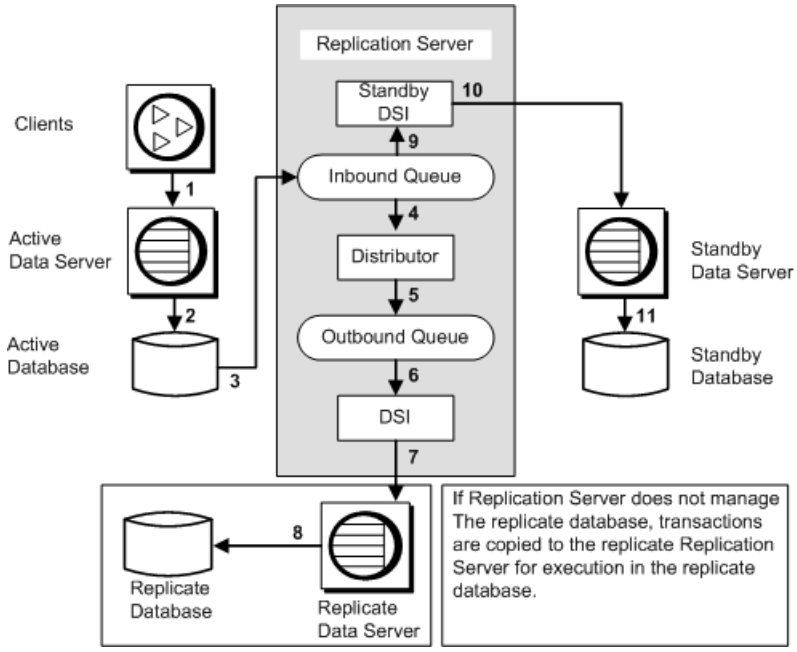
Warm Standby Application for a Primary Database

Learn about warm standby applications for a primary database.

This figure illustrates an example of a warm standby application for a primary database. In the 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.

Figure 9: Warm Standby Application for a 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.

The numbers in the figure 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 the figure, numbers 1 through 3 represent 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 Replication Agent 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 the figure, 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.

  Note: For Oracle, transactions executed by the maintenance user, are always replicated to the replicate database because the **filter_maint_userid** configuration parameter is invalid for Replication Agent for Oracle irrespective of whether the parameter is set to "true" or "false".
- 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 the figure, 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

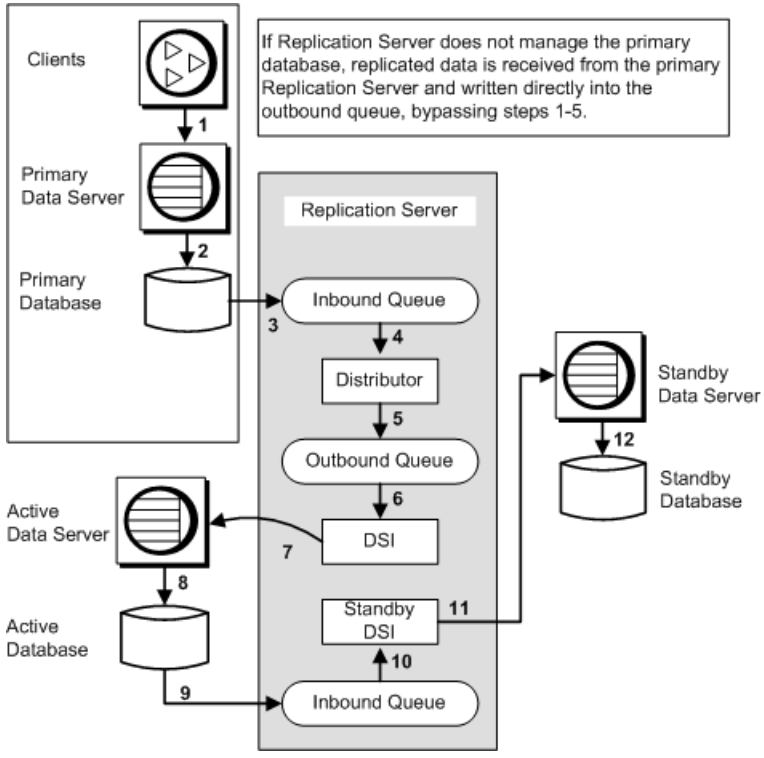
Learn about warm standby applications for a replicate database.

This figure illustrates an example of 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.

Figure 10: Warm Standby Application for a 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.

The numbers in the figure 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 the figure, 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.
- Replication Agent 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 the figure, numbers 9 through 12 trace transactions from the active database for the logical replicate database to its standby database:

- Replication Agent 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.

Configure Logical Connection Save Intervals

You can use the DSI queue save interval or the materialization queue save interval to reconfigure 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 **configure logical connection** to configure the DSI queue save interval and the materialization queue save interval for the logical connection.

See *Replication Server Reference Manual* > *Replication Server Commands* > ***configure logical connection***.

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'
```

Replication Definitions and Subscriptions for Warm Standby Databases

You can create replication definitions and subscriptions for warm standby applications.

In a replication system containing only Adaptive Server databases, you can reduce the need for replication definitions for tables in a warm standby environment or multisite availability (MSA) environment if a replication definition exists solely to specify primary key or quoted identifier information. See *Replication Server Administration Guide Volume 1 > Manage Replicated Objects Using Multisite Availability > Reduce the Use of Replication Definitions and Subscriptions*.

However, 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 normal replication into or out of the logical database.

See also

- *Warm Standby Applications Using Replication* on page 100

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.

For Oracle warm standby applications, you need replication definitions to enable replication of user defined datatypes. Replication Agent for Oracle automatically creates replication definition at the time of initialization. In such a scenario, you need to manually create new replication definition or alter existing replication definition to explicitly specify in the replication definition which user defined datatype is being replicated to the standby database.

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 *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Modify Replication Definitions* 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 the **alter table** command 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 *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***alter table*** for syntax and information.

alter table Add Column with Default

Learn how to avoid a DSI error when you issue the **alter table** command in the active database to add a column with a default value.

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 Replication Agent. 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 one with no replication defintion. 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.

See also

- *No Replication Definition* on page 107

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 Replication Agent 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 one with no replication definition. 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.

See also

- *No Replication Definition* on page 107

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.

Use Replication Definitions to Optimize Performance

Use replication definitions to improve the performance of the replication system in a warm standby environment.

When you specify that you want to use a replication definition for replicating into a standby database:

- 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.

Create 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.

See *Replication Server Reference Manual > Replication Server Commands >* ***create replication definition***.

Specify a Primary Key

The presence or absence of a table replication definition determines how primary key columns are packed in the **where** clause for a database. See *Primary Key Columns and* ***where*** *Clause Packing in Warm Standy and Multisite Availability Environments*, in *Replication Server Administration Guide Volume 1 > Manage Replicated Objects Using Multisite Availability > Reduce the Use of Replication Definitions and Subscriptions*.

Update 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.

Specify 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.

See *Replication Server Reference Manual > Replication Server Commands >* ***create replication definition*** for more information about using the **send standby** clause with the command.

Specify 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 *Replication Server Reference Manual > Replication Server Commands* for more information about using the **send standby** clause with the **create applied function replication definition** and **create request function replication definition**commands.

Use Replication Definitions to Copy Redundant Updates

If you want to replicate redundant updates, create a replication definition for the column that includes the **send standby replication definition** parameter option

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.

Include the **send standby replication definition** parameter option, if you want to replicate redundant updates.

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.

Use Subscriptions with Warm Standby Applications

You can use subscriptions with warm standby applications.

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 *Replication Server Administration Guide Volume 1 > Manage Subscriptions* for more information about subscriptions and subscription materialization.

See also

- *Warm Standby Applications Using Replication* on page 100

Restrictions on Using Subscriptions

Several restrictions apply to the creation of subscriptions that replicate data from or into warm standby databases.

Replication Server supports all forms of subscription materialization and dematerialization in warm standby applications. Restrictions that apply to the creation of subscriptions include:

- 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

Learn about subscription materialization issues for a logical primary database, and 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 Replication Agent 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

Learn about subscription materialization issues for a logical replicate database, and what happens if you execute the **switch active** command for a logical replicate database during subscription materialization.

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.

Note: Nonatomic materialization is not supported in heterogenous warm standby applications. See *Replication Server Heterogeneous Guide > Materialization*.

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.

Check 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.

Drop 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 a certain combination of circumstances.

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.

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 13. 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

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.

See also

- *Replication System Recovery* on page 309

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. These internal operations support various processes within the primary and replicate Replication Servers.

Threads, Modules, and Daemons

Learn how threads, modules, and daemons work in Replication Server.

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.

When you troubleshoot the replication system, verify the status of Replication Server threads, modules, and daemons.

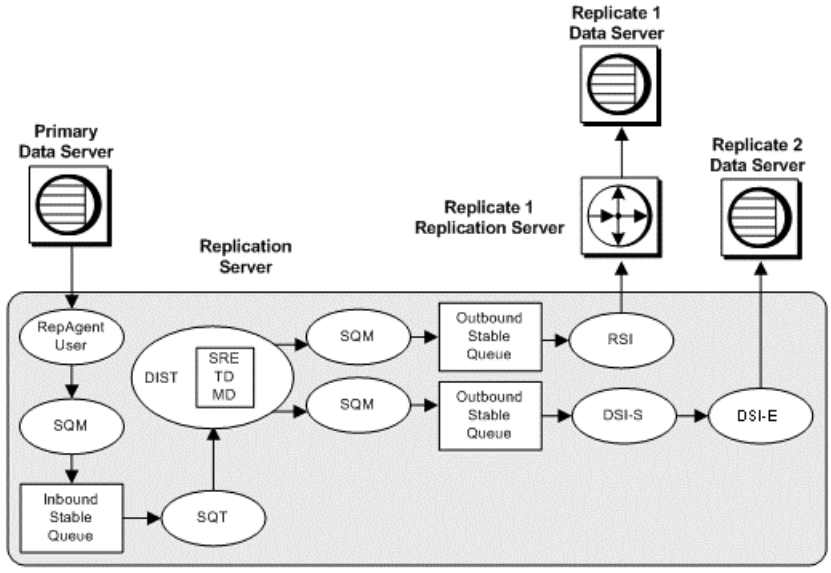
See also

- *Processes in the Primary Replication Server* on page 118
- *Verify and Monitor Replication Server* on page 5

Processes in the Primary Replication Server

Learn 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.

Figure 11: Threads Used for Processing in the Primary Replication Server



Replication Agent User Thread

Learn how RepAgent and other Replication Agents work with Replication Server to distribute transaction information to subscribing replicate databases

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

The Stable Queue Transaction (SQT) thread reassembles transactions and places the transactions in commit order.

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. 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 also

- *Parallel DSI Threads* on page 162

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.

Subscription Resolution Engine

The Subscription Resolution Engine (SRE) matches transaction rows with subscriptions.

When the SRE 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

Replication Server records the DIST status of a distributor thread in the Replication Server `rs_databases` system table in the RSSD

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.

The record in `rs_databases` allows the DIST thread to retain its status even after the Replication Server is shut down.

See *Replication Server Reference Manual* > *Replication Server System Tables* > `rs_databases`.

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, that is by using parallel DSI threads.

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.

See also

- *Parallel DSI Threads* on page 162

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. See *Replication Server Administration Guide Volume 1 > Manage Routes*.

Miscellaneous Daemon Threads

There are several Replication Server daemon threads that perform miscellaneous tasks in the replication system.

Table 14. Additional Replication Server Daemon Threads

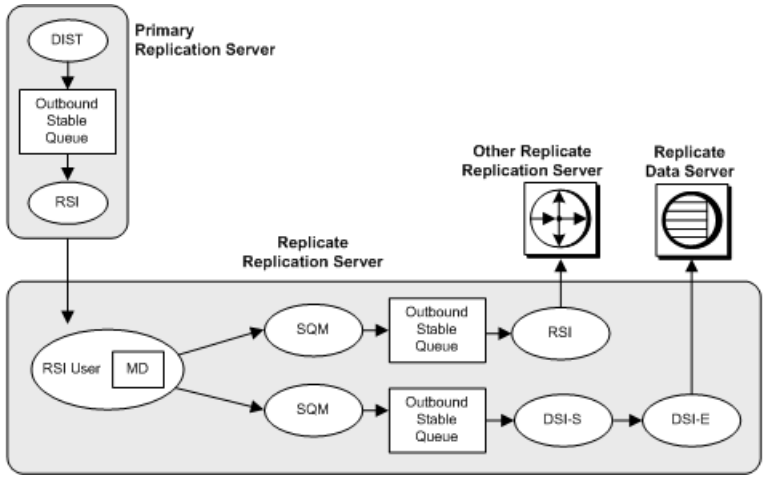
Thread or Daemon Name	Description
Alarm daemon (dALARM)	The alarm daemon keeps track of alarms set by other threads, such as the fade-out time for connections and the interval for the subscription retry daemon.
Asynchronous I/O daemon (dAIO)	The asynchronous I/O daemon manages asynchronous I/O to Replication 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.
RS user thread	The RS user thread manages connections from replicate Replication Servers during the process of creating or dropping subscriptions. See *Replication Server Administration Guide Volume 1 > Manage Subscriptions > Subscription Materialization Methods* 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.

Processes in the Replicate Replication Server

Learn the processes involved when a replicate Replication Server receives incoming messages from a primary Replication Server.

Some of the same threads—SQM, RSI, DSI—also involved in processes in the primary Replication Server are shown in the figure.

Figure 12: Transaction Processing in the Replicate Replication Server



See also

- *Processes in the Primary Replication Server* on page 118

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. 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.

The RSI user thread writes commands destined for other Replication Servers or databases into outbound queues. The threads involved in processes in the primary Replication Server process messages after they are stored in the outbound queues.

See also

- *Data Server Interface Threads* on page 121
- *Replication Server Interface Thread* on page 121
- *Processes in the Primary Replication Server* on page 118

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

You can change the values of the configuration parameters to improve Replication Server performance.

rs_init sets default configuration parameters after you install your Replication Server.

See *Replication Server Administration Guide Volume 1 > Manage a Replication System > Set Replication Server Configuration Parameters > Change Replication Server Parameters* for information on how to modify these parameters using **configure replication server**.

Table 15. Replication Server Parameters that Affect Performance

Configuration parameter	Description
block_size to '*value*' **with shutdown**	Specifies the queue block size which is the number of bytes in a contiguous block of memory used by stable queue structures. Valid values: 16KB, 32KB, 64KB, 128KB, or 256KB Default: 16KB **Note:** When you execute the command to change the block size, Replication Server shuts down. You must include the **with shutdown** clause after specifying the block size in versions prior to Replication Server 15.6. In version 15.6 and later the **with shutdown** clause is optional; you need not restart Replication Server for the change in queue block size to take effect. You should change this parameter only with the **configure replication server** command. Doing otherwise corrupts the queues. License: Separately licensed under the Advanced Services Option.
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. You may change this value if you have Adaptive Server that has been reconfigured. Maximum: 16,384 bytes Default: 512-byte network packet for all Adaptive Server databases

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: 2,048 on Linux and HPIA32, 1024 on other platforms
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
dist_direct_cache_read	Enables the distributor (DIST) thread to read SQL statements directly from the Stable Queue Thread (SQT) cache. This reduces the workload from SQT and the dependency between the two, and improves the efficiency of both SQT and DIST. Default: off
dsi_bulk_copy	Turns the bulk copy-in feature on or off for a connection. If **dynamic_sql** and **dsi_bulk_copy** are both on, DSI applies bulk copy-in. Dynamic SQL is used if bulk copy-in is not used. Sybase recommends that you turn **dsi_bulk_copy** on to improve performance if you have large batches of inserts. Default: off.

Configuration parameter	Description
dsi_bulk_threshold	The number of consecutive **insert** commands in a transaction that, when reached, triggers Replication Server to use bulk copy-in. When Stable Queue Transaction (SQT) encounters a large batch of **insert** commands, it retains in memory the number of **insert** commands specified to decide whether to apply bulk copy-in. Because these commands are held in memory, Sybase suggests that you do not configure this value much higher than the configuration value for **dsi_large_xact_size**. Replication Server uses **dsi_bulk_threshold** for Real-time loading (RTL) replication to Sybase IQ and High volume adaptive replication (HVAR) to Adaptive Server. If the number of commands for an insert, delete, or update operation on one table is less than the number you specify after compilation, RTL and HVAR use language instead of bulk interface. Minimum: 1 Default: 20
dsi_cmd_batch_size	The maximum number of bytes that Replication Server places into a command batch. Default: 8192 bytes
dsi_cmd_prefetch	Allows DSI to build the next batch of commands while waiting for the response from data server, and therefore improves DSI efficiency. If you also tune your data server to enhance performance, you are likely to gain an additional performance increase when you use this feature. Default: off When you set **dsi_compile_enable** to 'on', Replication Server ignores what you set for **dsi_cmd_prefetch**. License: Separately licensed under the Advanced Services Option.
dsi_max_xacts_in_group	Specifies the maximum number of transactions in a group. Larger numbers may improve data latency at the replicate database. Range of values: 1 – 1000. Default: 20 This parameter is ignored when **dsi_compile_enable** is turned on.

Configuration parameter	Description
dsi_non_blocking_commit	Specifies the number of minutes to extend the period of time Replication Server saves messages after a commit. Range of values: 0– 60 minutes. Default: 0 – non-blocking commit is disabled. Enable this parameter to improve replication performance when you use the the **delayed_commit** option of Adaptive Server 15.0 and later, or the equivalent feature in Oracle 10g v2 and later.
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. A value of –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. **Note:** Obsolete for Replication Server version 15.0 and later. Retained for compatibility with older Replication Servers. Maximum: 2,147,483,647 Default: 65,536 bytes This parameter is ignored when **dsi_compile_enable** is turned on.
dynamic_sql	Turns dynamic SQL feature on or off. Other dynamic SQL related configuration parameters take effect onlyif this parameter is on. Default: off
dynamic_sql_cache_size	Specifies to Replication Server how many database objects may use the dynamic SQL for a connection. Default: 100 Minimum: 1 Maximum: 65,536
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.

Configuration parameter	Description
exec_cmds_per_timeslice	Control the number of commands the RepAgent executor can process by using **exec_cmds_per_timeslice** to specify 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**. Default: 2,147,483,647 Minimum: 1 Maximum: 2,147,483,647
exec_nrm_request_limit	Specifies the amount of memory available for messages from a primary database waiting to be normalized. Set **nrm_thread** to 'on' with **configure replication server** before you use **exec_nrm_request_limit**. Default: 1,048,576 bytes (1MB) Minimum: 16,384 bytes (16KB) Maximum: 2,147,483,647 bytes (2GB) License: Separately licensed under the Advanced Services Option.
exec_sqm_write_request_limit	Specifies the amount of memory available for messages waiting to be written to an inbound queue. Default: 1MB Minimum: 16KB 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. Default: 100 milliseconds

Configuration parameter	Description
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. Default: 1,000 milliseconds
mem_reduce_malloc	Enable to allocate memory in larger chunks, which reduces the number of memory allocations and leads to improved Replication Server performance. Default: off License: Separately licensed under the Advanced Services Option.
mem_thr_dsi	Specifies the percentage of the total memory used to force the DSI thread to stop populating the SQT cache. Default: 80% of **memory_limit** value. Range: 1 – 100
mem_thr_exec	Specifies the percentage of the total memory used to force the EXEC thread to stop receiving commands from RepAgent. Default: 90% of **memory_limit** value. Range: 1 – 100
mem_thr_sqt	Specifies the percentage of the total memory used to force the SQT thread to flush the largest transaction from its cache if possible. Default: 85% of **memory_limit** value. Range: 1 – 100
mem_warning_thr1	Specifies the threshold percentage of the total memory used before the first warning message is generated. See **memory_limit**. Default: 80% of **memory_limit** value. Range: 1 – 100
mem_warning_thr2	Specifies the threshold percentage of the total memory used before the second warning message is generated. See **memory_limit**. Default: 90% of **memory_limit** value. Range: 1 – 100

Configuration parameter	Description
memory_control	Manages the memory control behavior of threads that require significant amount of memory. See **memory_limit**. Values are: • on – enables memory control • off – disables memory control Default: on

Configuration parameter	Description
memory_limit	The maximum total memory the Replication Server can use, in megabytes. 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_nrm_request_limit**, **exec_sqm_write_request_limit**, **md_sqm_write_request_limit**, **queue_dump_buffer_size**, **sqt_max_cache_size**, **sre_reserve**, and **sts_cachesize**. Default: 2,047 For 32-bit Replication Server: • Minimum – 0 • Maximum – 2,047 For 64-bit Replication Server: • Minimum – 0 • Maximum – 2,147,483,647 When **memory_control** is: • on – Replication Server does not shut down when memory consumption exceeds **memory_limit** • off – Replication Server automatically shuts down when memory consumption exceeds **memory_limit** Monitor memory usage and increase **memory_limit** if required. In Replication Server, the threads that require significant amount of memory are: • EXEC • SQT • DST These threads execute memory control by performing a memory usage check before receiving or processing new data. During memory control, if the memory usage is found to be high, thread functioning is adjusted by: • Stopping the thread from grouping new data, and cleaning and processing existing data; or, • Making the thread go into a sleep mode such that it does not receive new data until memory is available.

Configuration parameter	Description
	If the value you set is larger than 2,047, downgrading resets the value to 2,047 to protect against overflow.
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
nrm_thread	Enables the NRM thread which Replication Server can use to normalize and pack Log Transfer Language (LTL) commands in parallel with parsing by the RepAgent Executor thread. Parallel processing by the NRM thread reduces the response time of the RepAgent executor thread. The NRM thread is a thread split from RepAgent executor thread. Use the **configure replication server** command to set **nrm_thread** to on before you use **exec_nrm_request_limit**. Default: off License: Separately licensed under the Advanced Services Option.
rec_daemon_sleep_time	Set wake up intervals by specifying the sleep time for the recovery daemon, which handles “strict” save interval messages in warm standby applications and certain other operations. Default: 2 minutes

Configuration parameter	Description
smp_enable	Enables symmetric multiprocessing (SMP). Specifies whether Replication Server threads should be scheduled internally by Replication Server or externally by the operating 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.” Default: on Upgrading or downgrading does not change the value you set.
sqm_async_seg_delete	Set **sqm_async_seg_delete** to on to enable a dedicated daemon for deleting segments and improve performance for inbound and outbound queue processing. Default: on You must restart Replication Server for any change to the parameter setting to take effect. If **sqm_async_seg_delete** is on, Replication Server may require a larger partition. Use **alter partition** to expand a partition. See: • *Replication Server Configuration Guide > Preparation for Installing and Configuring Replication Server > Plan the Replication System > Initial Disk Partition for Each Replication Server*. • *Replication Server Administration Guide Volume 1 > Replication Server Technical Overview > Transaction Handling with Replication Server > Stable Queues > Partitions for Stable Queues*.
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**. Range is 1 to 4096. Default: 16

Configuration parameter	Description
sqm_page_size	Indicates the number of blocks in a page. Sets server-wide stable queue page size in blocks per page. Enclose page sizes in single quotes or double quotes. For example, setting page size to 4 instructs Replication Server to write to the stable queue in 64K chunks. Configuring the page size also sets the I/O size of Replication Server. The range is 1 to 64. Default: 4
sqm_recover_segs	Specifies the number of stable queue segments Replication Server allocates before updating the RSSD with recovery QID information. Sybase recommends that you increase the value of **sqm_recover_segs** to improve performance. Default: 1 Minimum: 1 Maximum: 2,147,483,648
sqm_write_flush	For stable device considerations, **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”. 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	Use **sqt_max_cache_size** for sizing the SQT cache to the maximum SQT cache memory, in bytes. For 32-bit Replication Server: • Default – 1,048,576 • Minimum – 0 • Maximum – 2,147,483,647 For 64-bit Replication Server: • Default – 20,971,520 • Minimum – 0 • Maximum – 2,251,799,813,685,247 If the value you set is larger than 2,147,483,647 bytes, downgrading resets the value to 2,147,483,647 bytes to protect against overflow.
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	For caching system tables, use **sts_cachesize** to specify 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 examining the Replication Server log for warnings that rows have been removed from the STS cache. Default: 1000

Configuration parameter	Description
sts_full_cache_*system_table_name*	For caching system tables, use **sts_full_cache**_*system_table_name* to specify an RSSD system table that is to be fully cached. Fully cached tables do not require access to the RSSD for simple **select** statements. Only some RSSD tables can be fully cached. Default: `rs_asyncfuncs`, `rs_clsfunctionsrs_columns`, `rs_objects`, `rs_objfunctions`, `rs_repobjs`, and `rs_users` are fully cached. Sybase recommends that you cache these tables to improve performance.
sub_daemon_sleep_time	For setting wake up intervals, use**sub_daemon_sleep_time** to set the number of seconds the subscription daemon sleeps before waking up to recover subscriptions. The range is 1 to 31,536,000. 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

See also

- *Increase Queue Block Size* on page 232
- *Advanced Services Option* on page 216
- *Set the Amount of Time SQM Writer Waits* on page 147
- *Set Wake up Intervals* on page 154
- *Stable Devices: Considerations* on page 136
- *Size the SQT Cache* on page 154
- *Cache System Tables* on page 148
- *Control the Number of Commands the RepAgent Executor Can Process* on page 156
- *Make SMP More Effective* on page 158
- *Specify the Number of Stable Queue Segments Allocated* on page 157

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 and later.

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 and later supports caching for stable device.

See also

- *Stable Queue Cache* on page 151

Connection Parameters that Affect Performance

Replication Server provides several database connection parameters that can affect performance.

See *Replication Server Administration Guide Volume 1 > Manage Database Connections* for a complete list of connection parameters.

Table 16. Connection Parameters that Affect Performance

Configuration Parameter	Description
batch	The default, "on," allows command batches to a replicate database. Default: on
cmd_direct_replicate	Set **cmd_direct_replicate** on for the Executor thread to send parsed data directly to the Distributor thread along with binary data. When required, the Distributor thread can retrieve and process data directly from parsed data, and improve replication performance by saving time otherwise spent parsing data again. Default: off
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
dist_sqt_max_cache_size	The maximum Stable Queue Transaction (SQT) cache size for the inbound queue 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 For 32-bit Replication Server: • Minimum – 0 • Maximum – 2,147,483,647 For 64-bit Replication Server: • Minimum – 0 • Maximum – 2,251,799,813,685,247
dsi_cmd_batch_size	The maximum number of bytes that Replication Server places into a command batch. Default: 8192 bytes
dsi_cmd_prefetch	Allows DSI to pre-build the next batch of commands while waiting for the response from data server, and therefore improves DSI efficiency. If you also tune your data server to enhance performance, it is likely that you will gain an additional performance increase when you use this feature. Default: off When you set **dsi_compile_enable** to 'on', Replication Server ignores what you set for **dsi_cmd_prefetch**. License: Separately licensed under the Advanced Services Option.
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. Default: 1000 ms (1 second) Minimum: 0 Maximum: 86,400,000 ms (24 hours)

Configuration Parameter	Description
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. 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. 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: 2,147,483,647 This parameter is ignored when **dsi_compile_enable** is turned on.

Configuration Parameter	Description
dsi_max_cmds_in_batch	Defines the maximum number of source commands whose output commands can be batched. You must suspend and resume the connection for any change in the parameter to take effect. Range: 1 – 1000 Default: 100
dsi_max_xacts_in_group	Specifies the maximum number of transactions in a group. Larger numbers may improve data latency at the replicate database. Range of values: 1 – 1000. Default: 20 This parameter is ignored when **dsi_compile_enable** is turned on.
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**, **time**, **user**, **name**, and **none**. Default: none This parameter is ignored when **dsi_compile_enable** is turned on.

Configuration Parameter	Description
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 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_partitioning_rule** option. • **no_wait** – specifies that a transaction can start as soon as it is ready —without regard to the state of other transactions. **Note:** You can only set **dsi_serialization_method** to **no_wait** if **dsi_commit_control** is set to "on". • **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. • **wait_after_commit** – specifies that a transaction cannot start until the transaction scheduled to commit immediately preceding it has committed completely. 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**. **Note:** The **isolation_level_3** value is no longer supported as a serialization method but it is the same as setting **dsi_serialization_method** to **wait_for_start** and **dsi_isolation_level** to 3. Default: **wait_for_commit**

Configuration Parameter	Description
dsi_sqt_max_cache_size	Maximum SQT (Stable Queue Transaction) interface cache size for the outbound queue 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 For 32-bit Replication Server: • Minimum – 0 • Maximum – 2,147,483,647 For 64-bit Replication Server: • Minimum – 0 • Maximum – 2,251,799,813,685,247
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. A value of –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. **Note:** Obsolete for Replication Server version 15.0 and later. Retained for compatibility with older Replication Servers. Maximum: 2,147,483,647 Default: 65,536 bytes This parameter is ignored when **dsi_compile_enable** is turned on.
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**. Default: 2,147,483,647 Minimum: 1 Maximum: 2,147,483,647

Configuration Parameter	Description
exec_max_cache_size	Specifies the amount of memory to allocate for the Executor command cache. Default: 1,048,576 bytes For 32-bit Replication Server: • Minimum – 0 • Maximum – 2,147,483,647 For 64-bit Replication Server: • Minimum – 0 • Maximum – 2,251,799,813,685,247
exec_nrm_request_limit	Specifies the amount of memory available for messages from a primary database waiting to be normalized. Set **nrm_thread** to 'on' with **configure replication server** before you use **exec_nrm_request_limit**. Minimum: 16,384 bytes Maximum: 2,147,483,647 bytes Default for: • 32-bit – 1,048,576 bytes (1MB) • 64-bit – 8,388,608 bytes (8MB) After you change the configuration for **exec_nrm_request_limit**, suspend and resume the Replication Agent. License: Separately licensed under the Advanced Services Option.
exec_sqm_write_request_limit	Specifies the amount of memory available for messages waiting to be written to an inbound queue. Default: 1MB Minimum: 16KB 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

Configuration Parameter	Description
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
sqm_async_seg_delete	Set **sqm_async_seg_delete** to on to enable a dedicated daemon for deleting segments. Default: on
sqm_cmd_cache_size	The maximum size, in bytes, of parsed data that Replication Server can store in the SQM command cache. 32-bit Replication Server: • Default – 1,048,576 • Minimum – 0, which disables SQM command caching • Maximum – 2,147,483,647 64-bit Replication Server: • Default – 20,971,520 • Minimum – 0 • Maximum – 2,251,799,813,685,247 Replication Server ignores any value you set for **sqm_cmd_cache_size** if **cmd_direct_replicate** or **sqm_cache_enable** is off.

Configuration Parameter	Description
sqm_max_cmd_in_block	Specifies, in each SQM block, the maximum number of entries with which the parsed data can associate. Default: 320 Minimum: 0 Maximum: 4096 Set the value of **sqm_max_cmd_in_block** to the number of entries in the SQM block. Depending on the data profile, each block has a different number of entries because the block size is fixed, and the message size is unpredictable. If you set a value that is too large, there is memory waste. If you set a value that is too small, replication performance is compromised. Replication Server ignores any value you set for **sqm_max_cmd_in_block** if **cmd_direct_replicate** or **sqm_cache_enable** is off.
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

See also

- *Advanced Services Option* on page 216
- *Parallel DSI Threads* on page 162
- *Partitioning Rules: Reducing Contention and Increasing Parallelism* on page 173
- *Specify the Number of Transactions in a Group* on page 158
- *Control the Number of Commands the RepAgent Executor Can Process* on page 156

Route Parameters that Affect Performance

Replication Server provides several route configuration parameters that affect performance.

See *Replication Server Administration Guide Volume 1 > Manage Routes* for a complete list of route parameters.

Table 17. 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: 4096 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

There are several 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.

Set the Amount of Time SQM Writer Waits

Use the **init_sqm_write_delay** and **init_sqm_write_max_delay** Replication Server configuration parameters to set the amount of time SQM writer waits.

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.

Cache System Tables

Use the **sts_cache_size** and **sts_full_cache**_*table_name* Replication Server configuration parameters to cache system tables.

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, rs_users rs_objects`, `rs_columns`, and `rs_asyncfuncs` are fully cached. 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.

If you have a lot of replication definitions in the replication system, and you have many replication definition change requests, each change may cause a refresh of the whole cache.

System Tables that Can Be Cached

Only certain system tables can be cached.

Table 18. 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_diskaffin-ity`	`rs_asyncfuncs`	`rs_users`
`rs_sites`	`rs_queues`	`rs_repdbs`	`rs_dbreps`
`rs_repobjs`	`rs_systext`	`rs_publica-tions`	`rs_objfunc-tions`
`rs_clsfunc-tions`	`rs_transla-tion`		

Replication Definition Change Process

If you are making many changes to the RSSD, such as creating, altering, or dropping replication definitions, or customizing function strings, Sybase recommends that before you start the replication definition change process, disable **sts_full_cache** for `rs_objects`, `rs_columns`, and `rs_objfunctions`, and then set **sts_full_cache** for these tables to their original values after the replication definition change process.

Tip: Execute the Adaptive Server **update statistics** command on the RSSD tables periodically if there are many RSSD changes. For replication definition change requests, such as to create, alter, or drop replication definitions, the affected tables are `rs_objects`, `rs_columns`, and `rs_objfunctions`. For function string change requests, such as to create, alter, or drop function strings, the affected tables are `rs_funcstrings` and `rs_systext`.

To disable **sts_full_cache** where *system_table_name* is the name of the table:

```
configure replication server
set sts_full_cache_system_table_name to ‘off
```

See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Modify Replication Definitions > Alter Replication Definitions > Replication Definition Change Request Process*.

Executor Command Cache

Use the Executor command cache to cache column names and datatypes for a primary Adaptive Server database table, when a Sybase RepAgent initially sends an **insert**, **delete**, or **update** LTL command for that table.

Metadata such as column name and datatype are part of the table schema that RepAgent sends as well as the data associated with an **insert**, **delete**, or **update** command. However, with caching:

- RepAgent sends the metadata and data associated with an **insert**, **update**, or **delete** command only when the RepAgent processes an operation for that specific table the first time since the RepAgent started, or since a connection with Replication Server was restarted. Replication Agent does not send the table metadata when RepAgent subsequently processes transactions for that table.
- RepAgent can resend metadata and data if there is not enough memory in the RepAgent to keep all the schema definitions.
- RepAgent sends the metadata and data of a table when the RepAgent processes a modification on a specific table after the table schema has been changed, for example, after an Adaptive Server **alter table** operation.

To replicate subsequent operations on the same table, RepAgent sends only the column data, since the Replication Server Executor command cache stores the metadata. The combination of RepAgent metadata reduction and caching with the Replication Server Executor command cache improves replication performance because caching:

- Reduces the time spent by RepAgent packing metadata into the Log Transfer Language (LTL) packet.
- Reduces network traffic by increasing the amount of data sent in each packet.
- Allows RepAgent to dedicate the time saved to scanning the primary database log instead of packing metadata.
- Allows the Replication Server Executor to process tables with large number of columns more efficiently.

Note: The cache contains only metadata from tables that have been modified by an **insert**, **update**, or **delete** operation.

System Requirements

Table metadata reduction requires LTL version 740 or later, and Adaptive Server 15.7 or later.

Enabling Table Metadata Reduction

Enable table metadata reduction for Sybase RepAgent. Replication Server enables the Executor command cache automatically if you enable table metadata reduction in RepAgent.

1. At the Adaptive Server, execute:

   ```
   sp_config_rep_agent database_name, 'ltl metadata reduction',
   'true'
   ```

   where *database_name* is the primary Adaptive Server database.

   Note: By default, **ltl metadata reduction** is set to false and RepAgent for Adaptive Server does not enable table metadata reduction.

2. Restart RepAgent for the change to take effect:

   ```
   sp_start_rep_agent database_name
   ```

Setting the Executor Command Cache Size

Use **exec_max_cache_size** to specify the amount of memory to allocate for the Executor command cache.

Insufficient memory allocated for the cache affects replication performance, and you see this error message appearing frequently:

```
Executor Command Cache exceeds its maximum limit defined by
exec_max_cache_size (current value is current_exec_max_cache_size).
```

To avoid further replication performance degradation, either add more memory to the cache with **exec_max_cache_size**, or disable table metadata reduction in your Replication Agent .

You can set values from 0 to 2,147,483,647 bytes for a 32-bit Replication Server, and 0 to 2,251,799,813,685,247 bytes for a 64-bit Replication Server. The default is 1,048,576 bytes for both 32-bit and 64-bit Replication Servers.

For example, to set the Executor command cache size to 2,097,152 bytes at the:

- Server level – for all primary database connections to Replication Server, enter:

```
configure replication server
set exec_max_cache_size to '2097152'
```

- Connection level – for a specific primary database connection, enter:

```
alter connection to dataserver_name.database_name
set exec_max_cache_size to '2097152'
```

Replication Server always uses the connection level setting if there are settings at both levels. You need not restart Replication Server for the change to take effect.

Stable Queue Cache

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.

Configure Stable Queue Cache Parameters

There are several stable queue cache parameters you can configure.

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.

SQM Command Cache

Use the SQM command cache to store parsed data from the Executor thread that the Distributor thread can retrieve directly, and therefore improve replication performance.

The Executor thread transfers LTL commands from a Replication Agent to Replication Server. The Executor thread parses the LTL commands and stores them in an internal parsed format. The parsed data is then packed in binary format. The Executor thread sends the binary data to the SQM thread so that the Executor thread can receive new data from the Replication Agent. The SQM thread stores the binary data in the SQM cache until the data is written to the inbound stable queue. The Distributor thread retrieves the binary data, restores the data to the original format, and determines where to send the data.

Set **cmd_direct_replicate** on for the Executor thread to send internal parsed data along with the binary data. Replication Server stores the parsed data in a separate SQM command cache. The parsed data in the SQM command cache maps to the binary data stored in SQM cache. When required, the Distributor module can retrieve and process data from parsed data directly, and save time otherwise spent parsing binary data.

Use the **sqm_cmd_cache_size** and **sqm_max_cmd_in_block** parameters to set the the SQM command cache memory configuration. You can configure **cmd_direct_replicate**, **sqm_cmd_cache_size** and **sqm_max_cmd_in_block** in the same command or separately.

Guidelines for Setting SQM Command Cache Memory Configuration

The SQM command cache memory configuration settings depend on the total amount of memory available to Replication Server, the number of inbound queues, and the transaction profile, which depends on the command size. When setting the SQM command cache memory configuration:

- Increase **sqm_cmd_cache_size** if the Replication Server has a large total SQM cache. Total SQM cache = **sqm_cache_size** (in pages) * **sqm_page_size** (in blocks) * **block_size** (in kilobytes)
- Decrease **sqm_max_cmd_in_block** if command size or table row size is large.
- Increase **sqm_max_cmd_in_block** if **block_size** is large.

After you have set the initial values, tune the values based on replication performance and data from monitoring counters:

- Increase **sqm_cmd_cache_size** if SQMNoDirectReplicateInCache shows a large value.
- Increase **sqm_max_cmd_in_block** if SQMNoDirectReplicateInBlock shows a large value.

Use **configure replication server** to change the **sqm_cache_size**, **sqm_page_size**, and **block_size** for all database connections to Replication Server. Otherwise, use **alter connection** to set the configuration for a specific database connection.

See *Replication Server Reference Manual > Replication Server Commands* for the default value and valid range of values for the parameters.

Example 1

To set the configuration for all connections and queues for a 64-bit Replication Server:

```
configure replication server
set cmd_direct_replicate to 'on'
set sqm_cmd_cache_size to '40971520'
set sqm_max_cmd_in_block to '640'
go
```

Example 2

To set the configuration for the connection to the `pdb1`primary database in the TOKYO_DS data server and for inbound queue number 2 for a 32-bit Replication Server:

```
alter connection to TOKYO_DS.pdb1
set cmd_direct_replicate to 'on'
go
alter queue 2, 1,
set sqm_cmd_cache_size to '2048576'
set sqm_max_cmd_in_block to '640'
go
```

See also

- *Increase Queue Block Size* on page 232
- *Monitor Performance Using Counters* on page 273
- *Configure Stable Queue Cache Parameters* on page 151

SQM Command Cache Counters to Monitor Performance

If **sqm_cache_enable** and **cmd_direct_replicate** are on, and **sqm_cmd_cache_size** and **sqm_max_cmd_in_block** are set to nonzero values, you can use several counters to monitor replication performance, as the Executor and Distributor threads interact with the parsed data.

Table 19. SQM Command Cache Counters

Counter	Description
RACmdsDirectRepSend	The number of commands sent from the Executor thread associated with parsed data.
DISTCmdsDirectRepRecv	Number of commands received by Distributor that have parsed data associated with the statement directly from Executor, and where the parsing processing can be skipped.
SQMNoDirectReplicateInCache	The number of commands that have parsed data sent from the Executor thread, but the parsed data cannot be sent further along the replication pathway towards the Distributor because the command cache exceeds **sqm_cmd_cache_size**
SQMNoDirectReplicateInBlock	The number of commands that have parsed data sent from the Executor thread, but the parsed data cannot be sent further along the replication pathway towards the Distributor because the number of parsed data entries for the current SQM block exceeds **sqm_max_cmd_in_block**

Set Wake up Intervals

Use the **rec_daemon_sleep_time** and **sub_daemon_sleep_time** Replication Server configuration parameters to set wake up intervals.

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.

Size the SQT Cache

Use the **sqt_max_cache_size** Replication Server configuration parameter and the **dsi_sqt_max_cache_size** database connection configuration parameter to size the SQT cache.

Monitor SQT cache usage by reviewing counter 24005 – CacheMemUsed. 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**.

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 15.5, setting **sqt_max_cache_size** too high can slow down replication. This advice does not apply to Replication Server 15.5 and later.

Control the Number of Outstanding Bytes

Use the **exec_nrm_request_limit**, **exec_sqm_write_request_limit**, and **md_sqm_write_request_limit** database connection configuration parameters to control the number of outstanding bytes of memory.

exec_nrm_request_limit is a separately licensed option you can use enhance RepAgent Executor thread efficiency.

See also

- *Enhanced RepAgent Executor Thread Efficiency* on page 230

exec_sqm_write_request_limit Database Configuration Parameter

exec_sqm_write_request_limit controls the amount of memory available for messages waiting to be written to an inbound queue.

md_sqm_write_request_limit Database Configuration Parameter

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.

Use Counters to Monitor Performance

You can use counters to monitor RepAgent Executor and NRM thread performance.

Monitor the number of times and duration of time RepAgent Executor sleeps and waits for normalization to complete, by reviewing this counter:

- 58038 – RAWaitNRMTime

Monitor the number of times and duration of time the thread which is sleeping, which can be either RepAgent Executor or NRM, waits before writing messages into an inbound queue by reviewing this counter:

- 58019 – RAWriteWaitsTime

If RAWriteWaitsTime is consistently large, review the StableDevice I/O.

See also

- *Monitor Performance Using Counters* on page 273

Control the Number of Network Operations

Use the **dsi_cmd_batch_size** database connection configuration parameter to control the size of a DSI command batch.

dsi_cmd_batch_size 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.

Counters to Monitor Batching and Batch Size

Replication Server provides counters to monitor batching and batch size.

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 – DSIEOCmd-Count	57063 – DSIEResultTime
57070 – DSIEBatchTime	57092 – DSIEBFMaxBytes	57076 – DSIEBatchSize

Control the Number of Commands the RepAgent Executor Can Process

Use the **exec_cmds_per_timeslice** database connection configuration parameter to control the number of commands the RepAgent executor thread can process.

By default, the value of the **exec_cmds_per_timeslice** parameter is 2,147,483,647 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

Specify the Number of Stable Queue Segments Allocated

Use the **sqm_recover_segs** Replication Server configuration parameter to specify 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 start up 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.

Select Disk Partitions for Stable Queues

Use the **disk_affinity** database connection configuration parameter to specify the logical name of the partition to which the next segment should be allocated when the current partition is full.

The Replication Server partition affinity feature 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.

See also

- *Allocation of Queue Segments* on page 268

Make SMP More Effective

Use the **smp_enable** Replication Server configuration parameter to enable symmetric multiprocessing (SMP).

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 operating system 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.

Specify 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

Use the **dsi_max_xacts_in_group** to specify the maximum number of transactions in a group.

Larger numbers may reduce commit processing at the replicate database, and thereby improve throughput.

Use **dsi_max_xacts_in_group** to control group size. Set **dsi_xact_group_size** to the maximum value of 2,147,483,647 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 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

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.

Set Transaction Size

For single DSI connections, set the value of **dsi_large_xact_size** to the maximum value of 2,147,483,647. 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.

Enable Nonblocking Commit

Use the **dsi_non_blocking_commit** Replication Server configuration parameter to enable nonblocking commit by specifying the number of minutes to extend the period of time Replication Server saves messages after a commit.

The nonblocking commit feature improves replication performance when the delayed commit feature is available in Adaptive Server 15.0 and later, or the equivalent delayed commit feature is available in Oracle 10g v2.

Range of values: 0– 60 minutes.

Default: 0 – Disable non-blocking commit.

Memory Consumption Controls

Replication Server can show warning messages when the memory consumption exceeds a defined threshold, and you can control the memory used by the EXEC, DSI, and SQT threads.

Memory Threshold Warning Messages

Configure Replication Server to show warning messages when the memory consumption exceeds a defined threshold percentage of the total available memory.

To configure warning messages, use:

- **mem_warning_thr1** – specifies the threshold percentage of the total memory used before the first warning message is generated.
  Default: 80% of **memory_limit** value.
  Range: 1 – 100.
- **mem_warning_thr2** – specifies the threshold percentage of the total memory used before the second warning message is generated.
  Default: 90% of **memory_limit** value.
  Range: 1 – 100.

Replication Server Threads Memory Control

You can avoid the automatic shutdown of Replication Server when the consumption of memory by Replication Server threads exceeds the available memory defined by **memory_limit**.

In Replication Server, the threads that require significant amount of memory are:

- DSI
- EXEC
- SQT

These threads execute memory control by performing a memory usage check before receiving or processing new data. During memory control, if the memory usage is found to be high, thread functioning is adjusted by:

- Stopping the thread from grouping new data, and cleaning and processing existing data; or,
- Making the thread go into a sleep mode such that it does not receive new data until memory is available.

To manage flow control in the EXEC, DST, and SQT threads, use:

- **mem_thr_dsi** – specifies the percentage of the total memory used to force the DSI thread to stop populating the SQT cache.
  Default: 80% of **memory_limit** value.
- **mem_thr_exec** – specifies the percentage of the total memory used to force the EXEC thread to stop receiving commands from RepAgent.
  Default: 90% of **memory_limit** value.
- **mem_thr_sqt** – specifies the percentage of the total memory used to force the SQT thread to flush the largest transaction from its cache.
  Default: 85% of **memory_limit** value.

Use **memory_control** to manage the memory control behavior of threads. Valid values for **memory_control** are enable (the default value) or disable. In this way, Replication Server controls the memory consumption and does not shut down because of memory issues.

Use **configure replication server** to alter the default values for the configuration parameters. Use **admin config** to view the default or existing values.

See *Replication Server Reference Manual > Replication Server Commands > **configure replication server***.

Monitor Thread Information

Use **admin who** to provide information on the memory control behavior of the thread.

State	Description
Controlling Mem	The thread is executing memory control.
Sleeping For Mem	The thread is sleeping until memory is available.

See *Replication Server Reference Manual > Replication Server Commands > **admin who***.

Memory Management Statistics

Use **admin stats** to view the memory management statistics.

Memory counters are enabled in the `rsh` module. To report the memory counters, use:

```
admin stats,rsh display_name instance_id
```

where:

- *display_name* – is the name of a counter. Use **rs_helpcounter** to obtain valid display names. *display_name* is used only with *module_name*.
- *instance_id* – identifies a particular instance of a module such as SQT or SQM. To view instance IDs, execute **admin who** and view the *Info* column. For `rsh` module, the *SPID* must be used. To view *SPID*, execute **admin who** and view the *Spid* column.

See *Replication Server Reference Manual* > *Replication Server Commands* > ***admin stats***.

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 of Using Parallel DSI Threads

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.

Use these configuration parameters with **alter connection** to tune parallel DSI threads for individual connections.

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'
```

Note: You can also set individual parallel DSI configuration parameters using the **configure replication server** command.

Parallel DSI Configuration Parameters

Replication Server provides several parallel DSI configuration parameters.

Table 20. Parallel DSI Configuration Parameters

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

Parameter	Description
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 statements allowed in a transaction before it is considered to be a large transaction. Default: 100 Minimum: 4 Maximum: 2,147,483,647 (in bytes)
dsi_max_xacts_in_group	Specifies the maximum number of transactions in a group. Larger numbersmay improve data latency at the replicate database. Range of values: 1 – 1000.Default: 20
dsi_max_cmds_in_batch	Defines maximum number of source commands for which output commands can be batched. Range: 1 – 1000 Default: 100
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

Parameter	Description
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**. 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. **Note:** You can only set **dsi_serialization_method** to **no_wait** if **dsi_commit_control** is set to "on". • **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. • **wait_after_commit** – specifies that a transaction cannot start until the transaction scheduled to commit immediately preceding it has committed completely. 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**.

Parameter	Description
dsi_sqt_max_cache_size	The maximum SQT cache size for the outbound queue 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 For 32-bit Replication Server: • Minimum – 0 • Maximum – 2,147,483,647 (in bytes) For 64-bit Replication Server: • Minimum – 0 • Maximum – 2,251,799,813,685,247 (in bytes)
parallel_dsi	A shorthand method for configuring parallel DSI threads. 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 (on 32-bit platform) and 20 million bytes (64-bit platform). 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

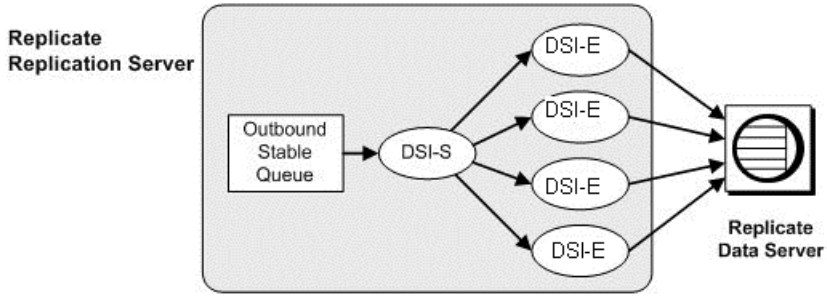
See also

- *Partitioning Rules: Reducing Contention and Increasing Parallelism* on page 173
- *Size the SQT Cache* on page 154
- *Configuration of Parallel DSI for Optimal Performance* on page 182

Components of Parallel DSI

Learn about the components of parallel DSI.

Figure 13: Parallel DSI Components



DSI Scheduler Thread

The DSI scheduler thread (DSI-S) 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 also

- *Partitioning Rules: Reducing Contention and Increasing Parallelism* on page 173

DSI Executor Threads

The DSI executor threads (DSI-E) 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.

Process 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.

Select 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’
```

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.

Set Isolation Levels for Non-Sybase Replicate Data Servers

Isolation levels may vary depending on the replicate data server. This has an impact on configuring parallel DSI in Replication Server.

The isolation levels you can set for non-Sybase replicate data servers are:

- Oracle – READ COMMITTED and SERIALIZABLE
- Microsoft SQL Server – READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SNAPSHOT, and SERIALIZABLE
- IBM DB2 UDB – REPEATABLE READ, READ STABILITY, CURSOR STABILITY, and UNCOMMITED READ

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**.

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.

To set an isolation level, create a function string in the appropriate function-string class. For example, in:

- Oracle – to set the SERIALIZABLE isolation level:

```
create function string rs_set_isolation_level
for rs_oracle_function_class
output language
‘set transaction isolation level serializable’
```

- Microsoft SQL Server – to set the SERIALIZABLE isolation level:

```
create function string rs_set_isolation_level
for rs_msss_function_class
output language
‘set transaction isolation level serializable
```

- IBM DB2 UDB – to set the REPEATABLE READ isolation level:

```
create function string rs_set_isolation_level
for rs_udb_function_class
output language
‘set current isolation = RR’
```

Transaction Serialization Methods

Replication Server provides different serialization methods for specifying the level of parallelization.

The 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.

The serialization methods are:

- **no_wait**
- **wait_for_start**
- **wait_for_commit**
- **wait_after_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.

See also

- *Partitioning Rules: Reducing Contention and Increasing Parallelism* on page 173

no_wait

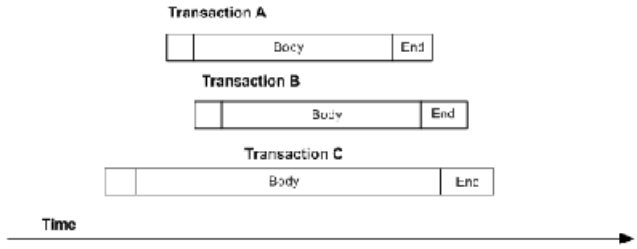
The **no_wait** method instructs the DSI to initiate the next transaction without waiting for the previous transaction to commit.

This method 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 the figure.

no_wait provides the better opportunity for increased performance, but also provides the greater risk of creating contentions.

Note: You can only set **dsi_serialization_method** to **no_wait** if **dsi_commit_control** is set to "on".

Figure 14: Thread Timing with the no_wait Serialization Method

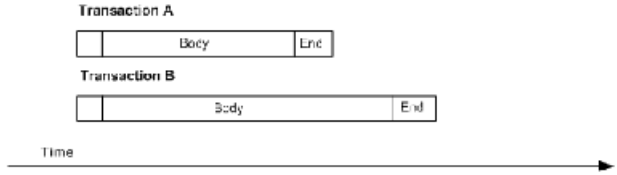


wait_for_start

wait_for_start specifies that a transaction can start as soon as the transaction scheduled to commit immediately before it has started.

Sybase recommends that you do not concurrently set **dsi_serialization_method** to **wait_for_start** and **dsi_commit_control** to **off**.

Figure 15: Thread Timing with the wait_for_start Serialization Method



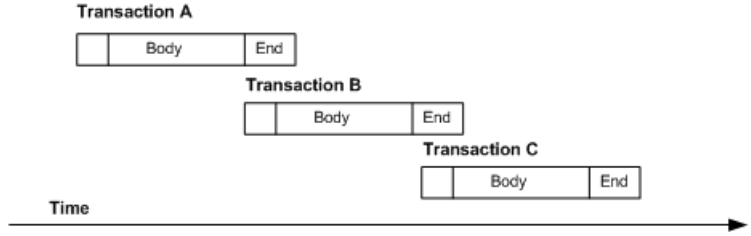
wait_for_commit

In the **wait_for_commit** 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 the figure.

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.

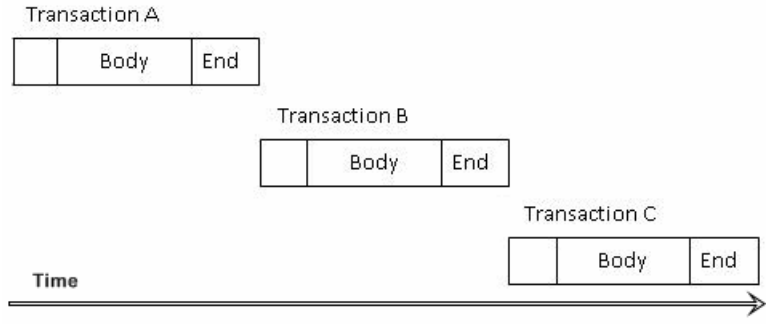
Figure 16: Thread Timing with the wait_for_commit Serialization Method



wait_after_commit

wait_after_commit specifies that a transaction cannot start until the transaction scheduled to commit immediately preceding it has committed completely.

Figure 17: Thread Timing with the wait_after_commit Serialization Method



Partitioning Rules: Reducing Contention and Increasing Parallelism

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.

Transaction-Partitioning Rules

Replication Server allows you to partition transactions for each connection according to one or more attributes.

The attributes are:

- 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.

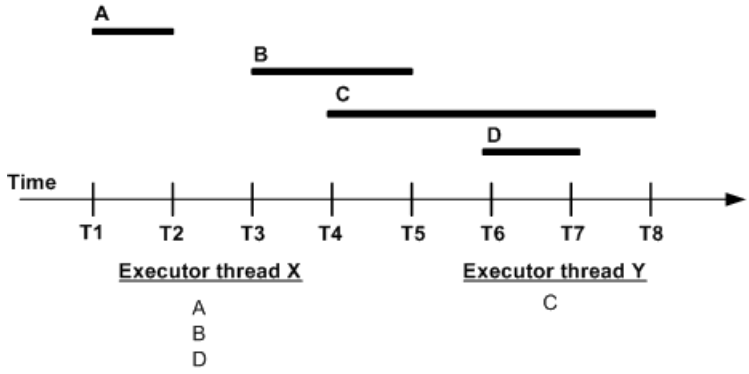
Partitioning Rule: Origin Time Begin and Commit Times

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 the figure are all from the same primary database.

Figure 18: 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 transaction 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

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.

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.

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.

Use 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.

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 also

- *Small Transactions* on page 168

Resolution of 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.

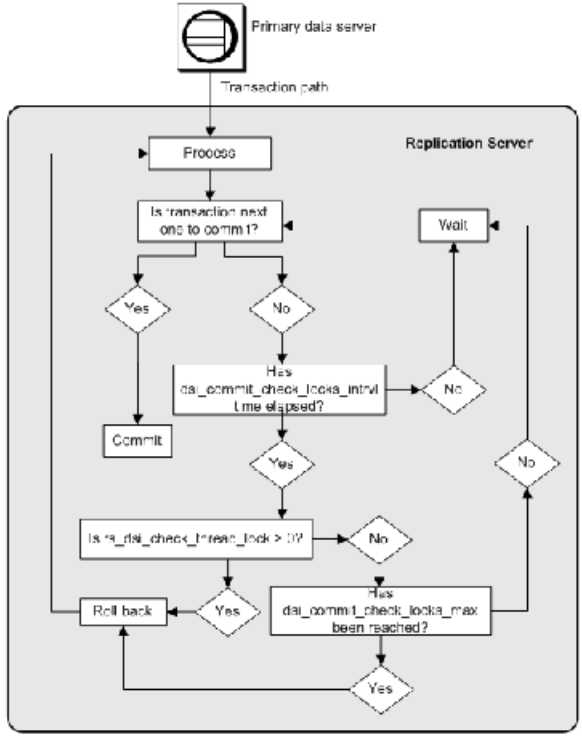
Resolution of Conflicts Internally

Learn how Replication Server resolves commit order deadlocks in Replication Server 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.

This figure describes the logic Replication Server uses to resolve commit order deadlocks.

Figure 19: 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.

Maintenance of 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.

Resolution of 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. 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.

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.

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**.

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.

Resolution of Conflicts Externally

Learn how Replication Server resolves commit order deadlocks externally using the `rs_threads` table.

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.

Resolution of Deadlocks

Learn how Replication Server resolves 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 several system functions.

These functions have function-string-class scope. They are executed only when more than one DSI thread is defined for a connection.

Note: These function strings are needed only when the external, `rs_threads` method is used for commit control.

Table 21. 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_no-holdlock	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.

Configuration of Parallel DSI for Optimal Performance

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.

Preparing to Configure Parallel DSI for Optimal Performance

Before you begin tuning for performance, there are several considerations.

1. 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.

2. 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.

3. 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.

4. Reset the **dsi_serialization_method** parameter.

   Note: You can only set **dsi_serialization_method** to **no_wait** if **dsi_commit_control** is set to "on".

   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.

5. 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**.

Reduce Contention

Start tuning parallel DSI parameters to reduce contention when you have completed the tasks to prepare configuration of parallel DSI for optimal performance, 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.

See also

- *Preparing to Configure Parallel DSI for Optimal Performance* on page 182

Use Partitioning Rules

Partitioning rules can also reduce contention, but require a clear understanding of your transaction profile.

The Transaction Name Rule

Find out if transactions have transaction names, and if the contention is 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.

Combine 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

Set values for the parallel DSI configuration parameters if your transactions frequently conflict with each other.

Set these parallel DSI configuration parameters:

- **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

Set values for the parallel DSI configuration parameters if your transactions conflict with each other only occasionally.

Set these parallel DSI configuration parameters:

- **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 through the use of the **rs_set_isolation_level** custom function string.
  - Oracle and Microsoft SQL Server – SERIALIZABLE level equals the ANSI SQL Isolation Level 3.
  - DB2 – REPEATABLE READ level equals the ANSI SQL Isolation 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.

See also

- *Set Isolation Levels for Non-Sybase Replicate Data Servers* on page 170

Set 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.

Set the Size for Large Transactions

Setting **dsi_large_xact_size** to a large number, even the maximum (2,147,483,647), 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**.

DSI Bulk Copy-in

Replication Server supports bulk-copy-in, which improves performance when replicating large batches of **insert** statements on the same table in the replicate database.

In normal replication, when replicating to a replicate database, Replication Server forms a SQL **insert** command, sends the command to the replicate database, and waits for the replicate database to process the row and send back the result of the operation. This affects Replication Server performance when large batches of data are being replicated, such as in end-of-day batch processing or trade consolidation.

Database Support

Bulk-copy-in is supported for Adaptive Server databases, and Oracle replicate databases that are updated by ExpressConnect for Oracle. If you turn on DSI bulk=copy-in and the replicate database is not supported, DSI shuts down with an error. See *Replication Server Options > ExpressConnect for Oracle Installation and Configuration Guide > System Requirements*.

DSI Bulk Copy-in Configuration Parameters

Database connection parameters that control bulk operations in DSI.

Parameter	Description
dsi_bulk_copy	Turns the bulk copy-in feature on or off for a connection. If **dynamic_sql** and **dsi_bulk_copy** are both on, Replication Server applies bulk copy-in when appropriate and uses dynamic SQL if Replication Server cannot use bulk copy-in. Default: off.
dsi_bulk_threshold	The number of consecutive **insert** commands in a transaction that, when reached, triggers Replication Server to use bulk copy-in. When Stable Queue Transaction (SQT) encounters a large batch of **insert** commands, it retains in memory the number of **insert** commands specified to decide whether to apply bulk copy-in. Because these commands are held in memory, Sybase suggests that you do not configure this value much higher than the configuration value for **dsi_large_xact_size**. Minimum: 1 Default: 20

Set Up Bulk Copy-in

Use **alter connection** or **configure replication server** to set the values of the bulk copy-in parameters.

Use:

- **alter connection** to change the bulk copy-in connection parameters at the connection level:

```
alter connection to dataserver.database
set {dsi_bulk_copy | dsi_bulk_threshold} to value
```

- **configure replication server** to change the server defaults:

```
configure replication server
set {dsi_bulk_copy | dsi_bulk_threshold} to value
```

To check the values of **dsi_bulk_copy** and **dsi_bulk_threshold**, use **admin config**.

When **dsi_bulk_copy** is on, SQT counts the number of consecutive **insert** statements on the same table that a transaction contains. If this number reaches the **dsi_bulk_threshold**, DSI:

1. Bulk copies the data to Adaptive Server until DSI reaches a command that is not **insert** or that belongs to a different replicate table.
2. Continues with the rest of the commands in the transaction.

Adaptive Server sends the result of bulk copy-in at the end of the bulk operation, when it is successful, or at the point of failure.

Note: The DSI implementation of bulk copy-in supports multistatement transactions, allowing DSI to perform bulk copy-in even if a transaction contains commands that are not part of the bulk copy.

Changes to Subscription Materialization

Bulk copy-in improves the performance of subscription materialization.

When **dsi_bulk_copy** is on, Replication Server uses bulk copy-in to materialize the subscriptions if the number of **insert** commands in each transaction exceeds **dsi_bulk_threshold**.

Note: In normal replication, bulk operation is disabled for a table if **autocorrection** is on. However, in materialization, bulk operation is applied even when **autocorrection** is enabled, if **dsi_bulk_threshold** is reached and the materialization is not a nonatomic subscription recovering from failure.

For more information about subscription materialization, see *Replication Server Administration Guide Volume 1 > Manage Subscriptions*.

Counters for Bulk Copy-in

Counters that support bulk copy-in.

Counter	Description
DSINoBulkDatatype	The number of bulk operations skipped due to the data containing datatype is incompatible with bulk copy-in.

Counter	Description
DSINoBulkFstr	The number of bulk operations skipped due to tables that have customized function strings for **rs_insert** or **rs_writetext**.
DSINoBulkAutoc	The number of bulk operations skipped due to tables that have **autocorrection** enabled.
DSIEBFBulkNext	The number of batch flushes that is executed because the next command is a bulk copy.
DSIEBulkSucceed	The number of times the Data Server Interface executor (DSI/E) invoked **blk_done(CS_BLK_ALL)** at the target database.
DSIEBulkCancel	The number of times DSI/E invoked **blk_done(CS_BLK_CANCEL)** at the target database.
DSIEBulkRows	The number of rows that DSI/E sent to the replicate data server using bulk copy-in.
BulkTime	The amount of time, in milliseconds, that DSI/E spent in sending data to the replicate data server using bulk copy-in.

Limitations for Bulk Copy-in

There are several limitations to be aware of when you use bulk copy-in.

The Replication Server DSI does not use bulk copy-in when:

- Autocorrection is on and the data is not part of subscription materialization.
- **rs_insert** has a user-defined function string.
- `text` column has a user-defined function string for **rs_writetext** with output `none` or `rpc`.
- The data row contains `opaque` datatype or a user-defined datatype (UDD) that has an `rs_datatype.canonic_type` value of 255.

The bulk copy-in feature is not supported under the conditions listed below. In these instances, disable bulk copy-in.

- The replicate database does not support bulk copy-in. In this case, if the DSI bulk copy-in is enabled, DSI terminates with an error message. See *Replication Server Heterogeneous Replication Guide*.
- The data size changes between Replication Server and the replicate Adaptive Server character sets, and the datarow contains text columns. In this case, if the DSI bulk copy-in is enabled, DSI terminates with this message:

```
Bulk-Lib routine 'blk_textxfer' failed.
Open Client Client-Library error: Error: 16843015,
Severity 1 -- 'blk_textxfer(): blk layer: user
error: The given buffer of xxx bytes exceeds the
total length of  the value to be transferred.'
```

- The `owner.tablename` length is larger than 255 bytes and the replicate database is earlier than version 15.0.3 Interim Release. If the DSI bulk copy-in is enabled, Replication Server terminates with this message:

  ```
  Bulk-Lib routine 'blk_init' failed.
  ```

  To specify not to use bulk copy-in when `owner.tablename` length is larger than 255 bytes:

  1. Turn trace on:

     ```
     trace "on", rsfeature, ase_cr543639
     ```

  2. Add this to the Replication Server configuration file:

     ```
     trace=rsfeature,ase_cr543639
     ```

Other limitations:

- Unlike the **insert** command, bulk copy-in does not generate timestamps; NULL values are inserted to the `timestamp` column if the `timestamp` column is not included in the replication. Either disable bulk copy-in, or set up your replication definition to include the `timestamp` column.
- `Text` and `image` columns are always logged, even if you change the **writetext** function string to **no log**.
- Bulk copy does not invoke **insert** trigger in Adaptive Server.
- The configuration parameter **send_timestamp_to_standby** has no effect on bulk copy-in. `timestamp` data is always replicated to standby.

SQL Statement Replication

Replication Server supports SQL statement replication in Adaptive Server which complements log-based replication and addresses performance degradation caused by batch jobs.

Sybase recommends that you use SQL statement replication when:

- DML (data manipulation language) statements affect a large number of rows on replicated tables.
- You have difficulty altering the underlying application to enable stored procedure replication

Note: Log Transfer Manager does not support SQL statement replication and SQL Statement replication to non-Adaptive Server databases is not supported.

Introduction to SQL Statement Replication

In SQL statement replication, Replication Server receives the SQL statement that modified the primary data, rather than the individual row changes from the transaction log.

Replication Server applies the SQL statement to the replicated site. The Adaptive Server RepAgent sends both the SQL data manipulation language (DML) and individual row

changes. Depending on your configuration, Replication Server chooses either individual row change log replication or SQL statement replication.

SQL statement replication includes row count verification to ensure that the number of rows changed in the primary and replicate databases match after replication. If the number of rows do not match, you can specify how Replication Server handles this error.

To enable and configure SQL statement replication:

- Configure the primary database to log SQLDML.
- Configure Replication Server to replicate SQLDML:
  1. Create replications definitions with SQLDML for table and multisite availability (MSA) replication.
  2. In Replication Server, set **WS_SQLDML_REPLICATION** parameter on for warm standby replication.

Performance Issues with Log-Based Replication

Learn how log-based replication affects performance and how to address the performance issues.

Sybase replication technology is log-based. Modifications performed on replicated tables are logged in the database transaction log. Adaptive Server generates a log record for each modification to each affected row; a single DML statement may result in Adaptive Server generating multiple log records. Depending on the type of DML statement, the Adaptive Server may log one “before” image and one “after” image for every affected row. The Sybase Replication Agent reads the log and forwards it to the Replication Server. The Replication Server identifies the DML operation (**insert**, **delete**, **update**, **insert**, **select**, or stored procedure execution) and generates the corresponding SQL statement for every operation.

Log-based replication has these inherent issues:

- When a single DML statement affects multiple rows, Replication Server applies multiple DML statements on the replicate site, not just the single original DML statement. For instance, if table `t` is replicated:

  ```
  1> delete tbl where c < 4
  2> go
  (3 rows affected)
  ```

  The **delete** statement logs three records in the transaction log, one for each of the rows deleted. These log records are used for database recovery and replication. Replication Agent sends the information pertaining to the three log records to the Replication Server, which converts the information back into three **delete** statements:

  ```
  delete t where c = 1
  delete t where c = 2
  delete t where c = 3
  ```

- Adaptive Server cannot perform optimizations on the replicate site that result in asymmetric loading of resources on the replicate database.

- Processing large numbers of statements affecting multiple rows increases latency in the system.
- Adaptive Server only partially logs information about **select into**; therefore, the replication system cannot successfully replicate the DML command.

There are two different approaches to address all of these issues:

- Stored procedure replication
- SQL statement replication

Stored Procedure Replication

You may use stored procedure replication to encapsulate complex DML operations or those affecting a large number of rows.

Stored procedure replication improves performance by replicating only the call to the stored procedure and ignoring modifications to individual rows. Network traffic is decreased and Replication Server needs less processing to apply the stored procedure at the replicate site.

In warm standby configurations that replicate DDL, **select into** operations cannot be replicated as they are minimally logged. Stored procedure replication cannot be used because of transaction management restrictions inherent to replication processing and to the **select into** command.

Additionally, some third-party applications cannot be easily modified to support replication of stored procedures. Consequently, even though stored procedure replication improves Replication Server performance, it cannot be used in all circumstances.

How Replication Server Topologies Affect SQL Statement Replication

To use SQL statement replication, you must take into account the underlying Replication Server topology.

Replication Server supports a wide range of topologies, including “basic primary copy” models that may include several Replication Servers, warm standby configurations, and multisite availability (MSA) configurations.

Like traditional replication, SQL statement replication is log-based; the information needed to replicate SQL statements (executed in the primary databases) is stored in the transaction log. The log reader, the Sybase Replication Agent, or other applications read the transaction log to notify Replication Server about modifications to a replicated table.

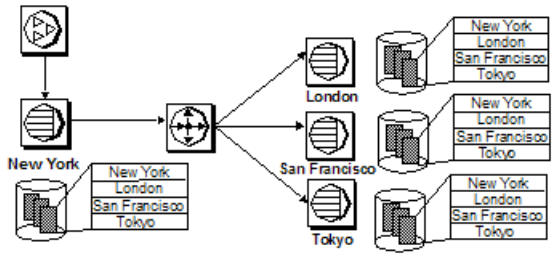
In simple MSA or warm standby configurations, source and destination data are identical, and a DML statement executed on the primary table affects the same data set on the replicate table.

Note: SQL statement replication applies only to DML statements.

Identical Data in Replicate Sites

This figure shows a Replication Server topology with a single primary database in New York. Tables are replicated to three other sites: London, Tokyo, and San Francisco. All tables are fully replicated.

Figure 20: Basic Primary Copy Model: Identical Data in Replicate Sites



Consider the following statement executed by a client connected to the New York site:

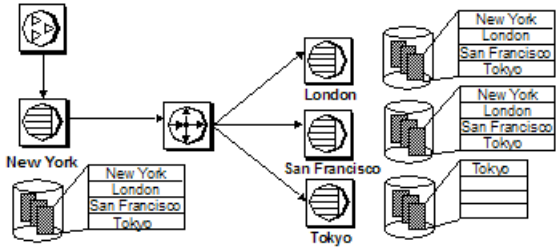
```
delete t1 where a>5
```

If this command is executed at Tokyo, London, and San Francisco, the same data set is affected at all the replicated sites, as data is identical in all the sites. In this case, all replicated sites can be configured to use SQL statement replication.

Nonidentical Data in Replicate Sites

This figure represents a system wherein the replicated site Tokyo subscribes only to a subset of data where the `site` is equal to “Tokyo”.

Figure 21: Basic Primary Copy Model: Nonidentical Data in Replicate Sites



Consider the following statement executed at the New York site:

```
delete t1 where a>5
```

Replication Servers can execute the same statement in London and San Francisco, but not in Tokyo, as this site subscribes only to a subset of data. If SQL statement replication is used in this case, some replicated databases, like the Tokyo site, receive individual log record modifications from the primary transaction log, based on traditional replication. Other replicated databases, like the London site, receive the SQL statement.

Different sets of data on the primary and replicate tables may also be affected when the primary and replicate databases have different object schema, or the user executes a DML statement using a **join** with another table. In these situations, different data is affected on the primary and replicate. The table used for the **join** may not be marked for replication, or values in that table may be partial or different from the primary database.

You must activate SQL statement replication in the Adaptive Server that holds primary data, and in the Replication Server. Once you enable SQL statement replication on the primary Adaptive Server, Adaptive Server logs additional information in the transaction log for each executed DML statement for which SQL statement replication was activated. The Replication Agent or other log readers deliver individual log record modifications and information for SQL statement replication to the Replication Server.

Note: The Sybase Replication Agent sends SQL statement replication information for Replication Server 15.2 and later.

Adaptive Server disallows SQL statement replication in situations where the statement may affect a different data set when applied on the replicate site.

Enable SQL Statement Replication

You can enable SQL statement replication at the database, table or session level. Session settings override both table and database settings. Table settings override database settings.

Several Adaptive Server stored procedures support SQL statement replication.

Enable SQL Statement Replication at the Database-level

Use **sp_setrepdbmode** to enable SQL statement replication at the database-level for a specific DML operation.

The DML operations that apply to SQL statement replication include:

- **U** – **update**
- **D** – **delete**
- **I** – **insert select**
- **S** – **select into**

For example, to replicate **delete** statements as SQL statements and also enable replication of **select into**, enter:

```
sp_setrepdbmode pdb, 'DS', 'on'
```

When an user executes a **delete** on a table in database `pdb`, Adaptive Server logs additional information for SQL statement replication. The RepAgent sends both individual log records, and the information needed by the Replication Server, to build the SQL statement.You can set SQL statement replication at the database level only when the database has been marked for replication by setting **sp_setreptostandby** to **ALL** or **L1**.

The **threshold** parameter defines the minimum number of rows that a DML statement must affect, to activate SQL statement replication. The default threshold is 50 rows, which means that Adaptive Server automatically uses SQL statement replication if the DML statement affects at least 51 rows.

For example, to set the threshold at the database-level to trigger SQL statement replication when a data manipulation language (DML) statement affects more than 100 rows:

```
sp_setrepdbmode pubs2, ‘threshold’, ‘100’
go
```

See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***sp_setrepdbmode***.

See also

- *Set SQL Statement Replication Threshold* on page 197

Display SQL Statement Replication Status

Use **sp_reptostandby** to display the SQL statement replication status at the database level.

For example:

```
sp_reptostandby pdb
go
The replication status for database 'pdb' is 'ALL'.
The replication mode for database 'pdb' is 'off'.
```

Enable SQL Statement Replication at the Table Level

Use **sp_setrepdefmode** to configure SQL statement replication at the table-level. Table-level settings override database-level settings.

sp_setrepdefmode includes options to:

- Enable or disable SQL statement replication for specific DML operations
- Configure the threshold that must be reached to activate SQL statement replication

The DML operations that apply to SQL statement replication include:

- **U** – **update**
- **D** – **delete**
- **I** – **insert select**

For example, to enable SQL statement replication for **update**, **delete,** and **insert select** operations on table `t`, use:

```
sp_setrepdefmode t, 'UDI', 'on'
go
```

When a user executes **deletes**, **updates**, or **insert select** DML statements on table `t`, Adaptive Server logs additional information for SQL statement replication. RepAgent reads the log and sends both individual log records and the information needed by Replication Server to build the SQL statement.

The **threshold** parameter defines the minimum number of rows that a DML statement must affect, to activate SQL statement replication. The default threshold is 50 rows, which means that Adaptive Server automatically uses SQL statement replication if the DML statement affects at least 51 rows.

For example, to set the threshold to 100, use:

```
sp_setreptable t, true
go
sp_setrepdefmode t, 'UD', 'on'
go
sp_setrepdefmode t, 'threshold','100'
go
```

In this example, **update** and **delete** statements executed on table `t` use SQL statement replication if the statement affects at least 101 rows.

See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures > **sp_setrepdefmode***.

Note: You cannot configure a **select into** operation at the table level because the target table does not yet exist.

See also

- *Set SQL Statement Replication Threshold* on page 197

Enable SQL Statement Replication at the Session Level

Use **set repmode** to configure SQL statement replication for the DML operation specified, for the duration of the session. Session settings override both database-level and object-level settings.

You can specify session-level settings either during login by using a login trigger, or at the beginning of a batch.

For example, to replicate only **select into** and **delete** as SQL statements for the duration of the session, use:

```
set repmode on 'DS'
```

Use **set repmode off** to remove all SQL statement replication settings at the session level.

The **set** options are active for the duration of the session. Options that you set inside a stored procedure are reverted to the default values when the stored procedure finishes.

Note: When you set options inside a login trigger, the option settings are maintained after the trigger has finished executing.

Executing **set repmode on** enables SQL statement replication only if session-level option **set replication on** is set. This example does not enable SQL statement replication:

```
set replication off
go
set repmode on 'S'
go
```

This example enables SQL statement replication:

```
sp_reptostandby pdb, 'ALL'
go
set repmode on 'S'
go
```

The **threshold** parameter defines the minimum number of rows that a DML statement must affect, to activate SQL statement replication. The default threshold is 50 rows, which means that Adaptive Server automatically uses SQL statement replication if the DML statement affects at least 51 rows.

This example shows how to define the threshold at the session-level as 1000 rows:

```
set repmode ‘threshold’, ‘1000’
go
```

See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***set repmode***.

See also

- *Set SQL Statement Replication Threshold* on page 197

Set SQL Statement Replication Threshold

You can trigger SQL statement replication without having to set the threshold on individual tables.

You can set the threshold at the:

- Database level – using Adaptive Server 15.0.3 ESD #1 and later.
- Session level – using Adaptive Server 15.0.3 ESD #2 and later.

In Adaptive Server 15.0.3, you could only set the threshold at the table level.

By default, SQL statement replication is triggered when the SQL statement affects more than 50 rows. You can set different threshold values at the session, database, and table-levels.

However, the threshold set at the session level overrides the threshold at the table level and database level, and the threshold set for any table overrides the threshold set at the database level.

Set Thresholds and Operations at Database Level

Use the **threshold** parameter for the **sp_setrepdbmode** command to set thresholds at the database level.

These examples show how to set the threshold at the database and table levels, and at the same time define operations at the different levels.

Example 1

This example shows how to set a different threshold at the database and table levels for the `pubs2` database and `table1` table:

1. Reset the threshold at the database level to the default of 50 rows:

   ```
   sp_setrepdbmode pubs2, 'threshold', '0'
   go
   ```

2. Enable SQL statement replication of **update**, **delete**, **insert**, and **select into** operations for `pubs2`:

   ```
   sp_setrepdbmode pubs2, 'udis', 'on'
   go
   ```

3. Trigger SQL statement replication for `table1` in `pubs2` only for the operations you defined in step 2 when these operations affect more than 1,000 rows:

   ```
   sp_setrepdefmode table1, 'threshold', '1000'
   go
   ```

Example 2

This example shows how to define the threshold at the database level for `pubs2`, and at the same time define different operations for tables, such as `table1` and `table2` located in the `pubs2` database:

1. Set the threshold at the database-level to trigger SQL statement replication when a data manipulation language (DML) statement affects more than 100 rows:

   ```
   sp_setrepdbmode pubs2, 'threshold', '100'
   go
   ```

2. Define a different set of operations for two specific tables, where you want operations replicated using SQL statement replication. Update, delete, and insert operations are for `table1` and delete operations are for `table2`:

   ```
   sp_setrepdefmode table1, 'udi', 'on'
   go
   sp_setrepdefmode table2, 'd' 'on'
   go
   ```

When a **delete** operation executes against `table2` or any DML on `table1` executes, the threshold of 100 rows that you defined at the database-level triggers SQL statement replication when reached.

Set Thresholds and Operations at Session Level

Use **set repthreshold** to set thresholds at the session level.

The threshold that you define at the session level overrides the threshold you set at the table or the database level. The threshold set at the table level overrides the threshold set at the database level.

These examples show how to set the threshold at the session, database, and table levels, and at the same time define operations at the different levels.

Example 1

This example shows how to define the threshold at the session-level to 23, in the absence of any threshold setting at the database and table levels, or to override the threshold settings at the table and database levels:

```
set repthreshold 23
go
```

Example 2

This example shows how to reset the threshold to the default of 50, at the session level:

```
set repthreshold 0
go
```

Example 3

This example shows how to set a different threshold at the database and table levels for the `pubs2` database and `table1` table, and then have a different operation defined for this session only:

1. Reset the threshold at the database level to the default 50 rows:

   ```
   sp_setrepdbmod pubs2, 'threshold', '0'
   go
   ```
2. Enable SQL statement replication of **update**, **delete**, **insert**, and **select into** operations for `pubs2`:

   ```
   sp_setrepdbmode pubs2, 'udis', 'on'
   go
   ```
3. Trigger SQL statement replication for `table1` in `pubs2` only when DML operations affect more than 1,000 rows:

   ```
   sp_setrepdefmode table1, 'threshold', '1000'
   go
   ```
4. Enable SQL statement replication only for **update** operations on any table and only for this session. This overrides the database-level setting in step 2:

   ```
   set repmode on 'u'
   go
   ```

Example 4

You can invoke **set reptheshold** within an Adaptive Server stored procedure. This example shows how to create the **set_rep_threshold_23** stored procedure and invoke it within the **my_proc** stored procedure:

1. Create the **set_rep_threshold_23** stored procedure:

```
create procedure set_rep_threshold_23
as
set repthreshold 23
update my_table set my_col = 2  (statement 2)
go
```

2. Create the **my_proc** stored procedure:

```
create procedure my_proc
as
update my_table set my_col = 1   (statement 1)
exec set_rep_threshold_23
update my_table set my_col = 3   (statement 3)
go
```

3. Execute **my_proc** to invoke **set_repthreshold_23**:

```
exec my_proc
go
```

Within the **my_proc** stored procedure, statement 1 executes first with a threshold of 50. Statement 2 executes next with a threshold of 23. Statement 3 executes next with a threshold of 50, because the **set repthreshold 23** command is valid only while executing the **set_rep_threshold_23** procedure.

Example 5

The session-level threshold is exportable. Therefore, you can set the **export_options** setting to 'on' for a procedure, and set the SQL statement replication threshold, so that procedures in the outer scope use the SQL statement replication threshold set by the stored procedure:

1. Create the **set_repthreshold_23** stored procedure and set **export_options** on:

```
create procedure set_repthreshold_23
as
set repthreshold 23             (statement 4)
set export_options on
update my_table set my_col = 2  (statement 2)
go
```

2. Create the **my_proc** stored procedure:

```
create procedure my_proc
as
update my_table set my_col = 1   (statement 1)
exec set_rep_threshold_23
update my_table set my_col = 3   (statement 3)
go
```

3. Execute **my_proc** to invoke **set_repthreshold_23**:

```
exec my_proc
go
```

Statement 1 executes first, with a threshold of 50. Statement 2 executes next with a threshold of 23. Statement 3 executes next with a threshold of 23, because the scope of the **set repthreshold 23** command is the scope of the session.

Example 6

You can create a login trigger to automatically set the replication threshold for a specific login ID.

1. Create the **threshold** stored procedure with a threshold setting of 23 and enable export:

   ```
   create proc threshold
   as
   set repthreshold 23
   set export_options on
   go
   ```

2. Instruct Adaptive Server to automatically run the **theshold** stored procedure when user "Bob" logs in:

   ```
   sp_modifylogin Bob, 'login script', threshold
   go
   ```

When Bob logs in to Adaptive Server, the SQL statement replication threshold for the session is set to 23.

Set Thresholds and Configure Replication

You can have a database that is not configured for replication and set the threshold for SQL statement replication at the database level at the same time.

For example:

```
sp_reptostandby pubs2, 'none'
go
sp_setrepdbmode pubs2, 'threshold', '23'
go
```

However, to define operations at the database level, you must also configure replication at the database level. For example, you cannot execute:

```
sp_reptostandby pubs2, 'none'
go
sp_setrepdbmode pubs2, 'udis', 'on'
go
```

Configure Replication Definitions for SQL Statement Replication

You can change SQL statement replication options at the database and table levels for replication definitions.

Database Replication Definition

To replicate SQL statements in an MSA environment, you must include the **replicate SQLDML** clause with the **create database replication definition** or **alter database replication definition** commands.

The syntax for **create database replication definition** or **alter database replication definition** must include the clause:

```
[[not] replicate setname [in (table list)] ]
```

where:

setname = **DDL** | **tables** | **functions** | **transactions** | **system procedures** | **SQLDML** | **'options'**.

The **'options'** parameter is a combination of:

- **U** – update
- **D** – delete
- **I** – insert select
- **S** – select into

The **SQLDML** parameter is also defined as a combination of **U, D**, **I**, and, **S** statements.

This example shows how to use the **'options'** parameter to replicate SQLDML on tables `tb1` and `tb2`:

```
replicate 'UDIS' in (tb1,tb2)
```

This example shows how to use the **SQLDML** parameter that produces the same result as the **'options'** parameter in the previous example:

```
replicate SQLDML in (tb1,tb2)
```

You can use multiple replicate clauses in a **create database replication** definition. However, for an **alter database replication** definition, you can use only one clause.

If you do not specify a filter in your replication definition, the default is the **not replicate** clause. Apply **alter database replication definition** to change the SQLDML filters. You can either specify one or multiple SQLDML filters in a **replicate** clause.

This example shows how to filter out the **select into** statement for all tables. The second clause, **not replicate 'U' in (T)**, filters out updates on table `T`:

```
create database replication definition dbrepdef
       with primary at ds1.pdb1
       not replicate 'S'
       not replicate 'U' in (T)
go
```

This example enables **update** and **delete** statements on all tables using the replicate 'UD' clause:

```
create database replication definition dbrepdef_UD
       with primary at ds2.pdb1
       replicate 'UD'
go
```

You can use multiple clauses to specify a table multiple times in the same definition. However, you can use each of **U**, **D**, **I**, and **S** only once per definition.

```
create database replication definition dbrepdef
       with primary at ds2.pdb1
       replicate tables in (tb1,tb2)
       replicate 'U' in (tb1)
       replicate 'I' in (tb1,tb2)
go
```

This example applies **update** and **delete** statements for tables `tb1` and `tb2`:

```
alter database replication definition dbrepdef
      with primary at ds1.pdb1
      replicate 'UD' in (tb1,tb2)
go
```

Table Replication Definition

To support SQL statement replication, you must include the **replicate SQLDML** clause when you create a table replication definition.

The syntax for **create replication definition** must include the clause:

```
[replicate {SQLDML [‘off’] | ‘options’}]
```

The **‘options’** parameter is a combination of these statements:

- **U** – update
- **D** – delete
- **I** – insert select

Note: If your replication definition has the **[replicate {minimal | all} columns]** clause, then the **[replicate {minimal | all} columns]** clause must always precede the **[replicate {SQLDML [‘off’] | ‘options’}]** clause.

This is a sample **create replication definition** for a table:

```
create replication definition repdef1
       with primary at ds3.pdb1
       with all tables named 'tb1'

           (id_col int,
           str_col char(40))

       primary key (id_col)
       replicate all columns
       replicate ‘UD’
go
```

A table replication definition with the **send standby** clause can specify a **replicate ‘I’** statement. You can replicate an **insert select** statement as a SQL replication statement only in

warm standby or MSA environments. A table replication definition without a **send standby** clause cannot replicate the **insert select** statement.

Warm Standby and SQL Statement Replication

Learn how to configure warm standby applications support for SQL statement replication.

By default, warm standby applications do not replicate the DML commands that support SQL statement replication. To use SQL replication, you can:

- Create table replication definitions using **replicate SQLDML** and **send standby** clauses.
- Set the **WS_SQLDML_REPLICATION** parameter to on. The default value is **UDIS**.
  However, **WS_SQLDML_REPLICATION** has a lower precedence than the table replication definition for SQL replication. If your table replication definition contains a **send standby** clause for a table, the clause determines whether or not to replicate the DML statements, regardless of the **WS_SQLDML_REPLICATION** parameter setting.

Row Count Validation for SQL Statement Replication

You can specify how Replication Server responds to SQLDML row count errors that may occur during SQL statement replication.

SQLDML row count errors occur when the number of rows changed in the primary and replicate databases do not match after SQL statement replication. The default error action is to stop replication.

Use the **assign action** command at the primary site for the Replication Server error class to specify other error actions for SQLDML row count errors:

assign action

```
   {ignore | warn | retry_log | log | retry_stop | stop_replication}
   for error_class
   to server_error1 [, server_error2]...
```

where:

- *error_class* is the error class name for which the action is being assigned. You can specify Replication Server error classes such as the default **rs_repserver_error_class** error class.
- *server_error* is the error number. You can specify error numbers for Replication Server.

For example, to assign the **warn** error action if Replication Server encounters error number 5186, enter:

```
assign action warn for rs_repserver_error_class to 5186
```

If there is a row count error, this is an example of the error message generated:

```
DSI_SQLDML_ROW_COUNT_INVALID 5186
Row count mismatch for SQLDML command executed on
'mydataserver.mydatabase'.
The command impacted 1000 rows but it should impact 1500 rows.
```

See also

- *Data Server Error Handling* on page 291

Error Actions for SQL Statement Replication

Replication Server supports several error actions for SQL statement replication errors.

Table 22. Error Actions for SQL Statement Replication

server_error	Error Message	Default Error Action	Description
5186	`Row count mismatch for the command executed on 'dataserver.database'. The command impacted x rows but it should impact y rows.`	**stop_replication**	Row count verification error for SQL statement replication if the affected number of rows is different from what is expected.
5193	`You cannot enable autocorrection if SQL Statement Replication is enabled. Either enable SQL Statement Replication only or disable SQL StatementReplication before you enable autocorrection.`	**stop_replication**	Cannot enable autocorrection if SQL statement replication is enabled. Either enable SQL statement replication only or disable SQL statement replication before you enable autocorrection

Scope of SQL Statement Replication

Learn how SQL statement replication applies to DML statements in batch processing, triggers, and stored procedures.

Batch Processing

There are some requirements when you apply SQL statement replication to any DML statement that is executed in a batch.

- The configuration must allow SQL statement replication.
- The DML statement does not conform to any of the conditions or exceptions to using SQL statement replication.

In the example below, while executing the batch statement with **delete** and **insert**, only the first statement uses SQL statement replication. `table2` uses traditional replication because `table2` is not configured to use SQL statement replication:

```
create table table1 (c int, d char(5))
go
create table table2 (c int, d char(5))
go
```

```
insert table1 values (1, 'ABCDE')
go 100
sp_setreptable table1, true
go
sp_setreptable table2, true
go
sp_setrepdefmode table1, 'UDI', 'on'
go
delete table1 where c=1
insert table2 select * from table1
go
```

See also

- *Exceptions to Using SQL Statement Replication* on page 209

Stored Procedures

The replication status of a stored procedure determines if DML statements within it are replicated as statements.

- If a stored procedure is not marked for replication, a DML statement within it is replicated as a statement, provided that:
  - The configuration allows SQL statement replication.
  - The DML statement does not conform to any of the conditions or exceptions to using SQL statement replication.
- If a stored procedure is marked for replication, only the call to it is replicated, not the individual statements that make up the stored procedure.

See also

- *Exceptions to Using SQL Statement Replication* on page 209

Triggers

There are some requirements when Adaptive Server uses SQL statement replication for DML statements within triggers.

- The configuration allows SQL statement replication.
- The DML statement does not conform to any of the conditions or exceptions to using SQL statement replication.

In the example below, when a **delete** statement is executed on `table1`, it is replicated using traditional replication. The **delete** executed on `table2` via the trigger is replicated using SQL statement replication as the table is configured for SQL statement replication and the **delete** meets the conditions to be replicated as a statement:

```
create table table1 (c int, d char(5))
go
create table table2 (c int, d char(5))
go
sp_setreptable table1, true
go
```

```
sp_setreptable table2, true
go
insert table1 values (1,'one')
go
insert table2 values (2,'two')
go 100
sp_setrepdefmode table2, 'udi', 'on'
go
create trigger del_table1 on table1
for delete
as
begin
delete table2
end
go
delete table1 where c=1
go
```

See also

- *Exceptions to Using SQL Statement Replication* on page 209

Recompilation of Stored Procedures and Triggers

Stored procedures and triggers are automatically recompiled if SQL replication settings have changed from “off” to “on” between two successive executions of the stored procedure or trigger.

SQL Statement Replication Setting at Compile Time	SQL Statement Replication Setting at Runtime	Automatically Recompile Stored Procedure/ Trigger?
Off	On	Yes
On	Off	No

Cross-Database Transactions

A single transaction may affect tables from different databases. Modifications to tables located in a different database are logged in the databases that hold the tables.

The Sybase Replication Agent configured for the database sends the Replication Server information stored in its transaction log.

In this example, `db1` and `db2` are replicated databases with configured Sybase Replication Agents. Database `db1` is configured to use SQL statement replication:

```
use db2
go
begin tran
go
delete t1 where c between 1 and 10000000
delete db1..t1 where c between 1 and 1000000
commit tran
go
```

The second **delete** (on database `db1`) uses SQL statement replication whereas the first **delete** (on database `db2`) uses traditional replication. The Sybase Replication Agent running on `db1` replicates the statement.

Replication Server does not guarantee the integrity of transactions across different databases. For example, if the DSI for the first **delete** suspends while the DSI for the second **delete** is active, the second **delete** replicates ahead of the first **delete**.

Issues Resolved by SQL Statement Replication

In some cases, data cannot be replicated using traditional replication methods. SQL statement replication provides a way to replicate data successfully in such situations .

Replication of the select into Operation In Warm Standby Configurations

select into creates a new table based on the columns specified in the **select** list and the rows chosen in the **where** clause. This operation is minimally logged for recovery purposes, and cannot be replicated using traditional replication.

select into can be replicated in warm standby configurations by using SQL statement replication. To configure SQL statement replication at the database level, use:

```
sp_setrepdbmode pdb, 'S', 'on'
go
```

Once the option is active at database level, all **select into** operations in database `pdb` will be replicated using SQL statement replication. Review the exceptions to using SQL statement replication to verify that the query can be replicated using SQL statement replication. If only a subset of **select into** needs to be replicated, use **set repmode** instead.

See also

- *Exceptions to Using SQL Statement Replication* on page 209

Replication of Deferred Updates on Primary Keys

Updates on tables that have a unique column index are not supported by traditional replication, and the Replication Server reports errors.

For example, table `t` has a unique index on column `c`, with values: 1, 2, 3, 4 and 5. A single **update** statement is applied to the table:

```
update t set c = c+1
```

Using traditional replication, this statement results in:

```
update t set c = 2 where c = 1
update t set c = 3 where c = 2
update t set c = 4 where c = 3
update t set c = 5 where c = 4
update t set c = 6 where c = 5
```

The first update attempts to insert a value of c=2 into the table; however, this value already exists in the table. Replication Server displays error 2601—an attempt to insert a duplicate key.

You can use SQL statement replication to address this issue. If the table has a unique index, and SQL statement replication is configured for **update** statements, the Adaptive Server replicates the update using SQL statement replication.

Exceptions to Using SQL Statement Replication

There are several limitations to using SQL statement replication.

SQL statement replication is not supported when:

- A replicate database has a different table schema than the primary database.
- Replication Server must perform data or schema transformation.
- Subscriptions or articles include **where** clauses.
- Updates include one or more `text` or `image` columns.
- Function strings `rs_delete`, `rs_insert`, and `rs_update` are customized.
- A DML statement matches one or more conditions listed here. In these cases, traditional replication is used:
  - The statement refers to views, temporary tables, or tables in other databases.

    ```
    insert tbl select * from #tmp_info
    where column = 'remove'
    ```

  - The user executes the statement with **set rowcount** option set to a value greater than zero.

    ```
    set rowcount 1
    update customers
    set information = 'reviewed'
    where information = 'pending'
    ```

  - The statement uses the **top n** clause in **select** or **select into** statements, a Java function, or a SQL User-Defined Function(UDF):

    ```
    delete top 5
    from customers
    where information = 'obsolete'
    ```

  - The base table includes encrypted columns, and the statement references one of those columns in a **set** or **where** clause.
  - The statement references system catalogs or fake tables such as ‘deleted’ or ‘inserted’. In this example, the **delete** executed by the trigger will not use SQL statement replication because it is using the fake table `deleted`:

    ```
    create trigger customers_trg on customers for delete as
    delete customers_hist
    from customers_hist, deleted
    where deleted.custID = customers_hist.custID
    go
    delete customers where  state = 'MA'
    go
    ```

- The statement is an **insert** statement that generates a new `identity` or `timestamp` value.
- The statement is an **update** statement that changes a `timestamp` or `identity` value.
- The statement is an **update** statement that assigns a value to a local variable. For example:
  ```
  update t set @a = @a + 2, c = @a where c > 1
  ```
- The statement makes references to materialized computed columns.
- The statement is a **select into** statement that affects a replicate table with encrypted columns.
- The statement is an **insert select** or **select into** using a **union** clause:
  ```
  select c1, c2 from tbl2
  union
  select cc1, cc2 from tbl3
  ```
- The statement is an **update**, **insert select**, or **select into** on a table with `text/image` columns.
- The statement is a query that uses built-ins:
  If the built-in can be resolved to a constant value, the query is replicated as a SQL statement. For example:
  ```
  update tbl set value = convert(int, "15")
  ```
  However, the following query will not be replicated using SQL statement replication:
  ```
  update tbl set value = convert(int, column5)
  ```
  In warm standby topologies, queries containing the following built-ins can be replicated using SQL statement replication even if the built-in cannot be resolved to a constant value:

abs	cot	ltrim	sqrt
acos	datalength	patindex	str
ascii	degrees	power	strtobin
asin	exp	replicate	stuff
atan	floor	reverse	substring
atn2	hextoint	right	tan
bintostr	inttohex	round	to_unichar
ceiling	len	rtrim	upper
char	log	sign	
convert	log10	soundex	

cos	lower	space	

SQL Statement Replication Does not Support Autocorrection

SQL statement replication cannot perform autocorrection. If Data Server Interface (DSI) encounters a DML command for SQL statement replication and autocorrection is on, by default, DSI is suspended and stops replication.

Use **assign action** with error number 5193 to specify how Replication Server handles this error.

Replication Server does not replicate SQLDML until the table level subscription is validated.

RSSD System Table Modifications

There are several changes to system tables in the Replication Server System Database (RSSD) to support SQL statement replication.

These system tables in the RSSD support SQL statement replication:

- `rs_dbreps` – `status` column includes 4 new sets of 2-bit sets, each of which corresponds to a DML filter. The first bit of a set indicates if it is an empty filter and the second bit indicates if it is a negative statement set.
- `rs_dbsubsets` – `type` column includes four new types: **U**, **L**, **I**, and, **S** corresponding to the DML **UDIS** filters. In this case, **L** is used for delete instead of **D**.
- `rs_objects` – `attributes` column includes five new bits; one for each **U**, **D**, **I**, or **S** operation, and one to indicate if a table replication definition has fewer columns than the number of incoming data rows.
  A system function replication definition, **rs_sqldml**, also supports SQL statement replication.

Adaptive Server Monitoring Tables for SQL Statement Replication

Use Adaptive Server monitoring tables to provide a statistical snapshot of the state of Adaptive Server during SQL statement replication. The tables allow you to analyze Adaptive Server and Adaptive Server performance.

Table	Description
`monSQLRepActivity`	Provides statistics for all open objects on DML statements replicated using SQL statement replication.
`monSQLRepMisses`	Provides statistics for replicated operations for which SQL statement replication was not used. The `threshold`, `querylimitation`, and `configuration` columns indicate the number of times that one of these factors prevented SQL statement replication for the object.

See *Adaptive Server Enterprise > Performance and Tuning Series: Monitoring Tables > Introduction to Monitoring Tables > Monitoring Tables in Adaptive Server*.

Product and Mixed-Version Requirements

SQL statement replication requires Adaptive Server version 15.0.3 and later, primary and replicate Replication Server version 15.2 and later, and route version 15.2 and later.

Downgrades and SQL Statement Replication

Learn the downgrade procedures if you are using SQL statement replication and you wish to downgrade Adaptive Server to a version earlier than 15.0.2 ESD #3, or Replication Server to a version earlier than 15.2.

Downgrade of Adaptive Server

You can downgrade Adaptive Server to an earlier version while there still are transaction records related to SQL statement replication in the log.

If you downgrade to a version earlier than 15.0.2 ESD #3, Sybase recommends that you use the standard documented procedure to downgrade an Adaptive Server with replicated databases. This procedure includes draining the transaction log. See the *Adaptive Server Enterprise 15.0.3 Installation Guide*.

Adaptive Server 15.0.3 provides the following downgrade support for Sybase Replication Agents version 15.0.2 ESD #3 and later:

- Sybase Replication Agents continue to replicate data even if the log contains information for SQL statement replication.
- When a Sybase Replication Agent reads a transaction containing SQL statement replication, it sends atomic modifications for that statement and ignores information related to SQL statement replication.

Downgrade of Replication Server

You may downgrade a Replication Server to a version earlier than 15.2.

The Sybase Replication Agent controls the amount and type of information sent to Replication Server based on the Log Transfer Language (LTL) version negotiated when the connection is established.

For Replication Servers earlier than 15.2, Sybase Replication Agent does not send information for SQL statement replication, and proceeds with standard replication.

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. In addition, DSI optimizes dynamic SQL statements by generating the language command only when the dynamic SQL command fails, and generating the prepared statement only once when the prepared statement is used for the first time.

If enabled, dynamic SQL is 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.

Dynamic SQL Configuration Parameters

Use the dynamic SQL configuration parameters to enable and tune dynamic SQL.

- **dynamic_sql** – turns dynamic SQL on or off for a replicate connection. Other dynamic SQL 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 limits 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.**

Set up the Configuration Parameters to Use Dynamic SQL

You can enable or configure dynamic SQL at the server or database connection level.

Dynamic SQL is off by default at a server and connection level.

Set the dynamic SQL configuration parameters at the server-level to provide the default values for the connections created or started in the future. To configure dynamic SQL at the server level, enter:

```
configure replication server
set { dynamic_sql |
      dynamic_sql_cache_size |
      dynamic_sql_cache_management }
to value
```

To configure dynamic SQL at the connection level, enter:

```
alter connection to server.db
set { dynamic_sql |
      dynamic_sql_cache_size |
      dynamic_sql_cache_management }
to value
```

Table-Level Dynamic SQL Control

create replication definition and **alter replication definition** allow you to control the application of dynamic SQL on each table through replication definitions.

See **create replication definition** and **alter replication definition** in *Replication Server Reference Manual > Replication Server 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. Change the default only 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 counters.

Use **rs_helprep**, **rs_helpsub**, and **rs_helppubsub** to display dynamic SQL settings for each replication definition.

See *Replication Server Reference Manual > RSSD Stored Procedures*.

replicate minimal columns Clause and Dynamic SQL

Replication processing uses dynamic SQL when the replication definition contains replicate minimal columns or, when you set **replicate_minimal_columns** on for a connection.

You can use **replicate_minimal_columns** for physical connections and warm standby environments. DSI can use the parameter to determine whether to use minimal columns when there is no replication definition, or when the replication definition does not contain the **replicate minimal columns** clause.

By default, **replicate_minimal_columns** is on for all connections. The **replicate_minimal_columns** setting for a connection overrides replication definitions set with the **replicate all columns** clause.

With custom function strings, the behavior of the current replication environment may change when you set **replicate_minimal_columns** on for a connection. If the application is relying on a command to be sent to the replicate database for trigger processing, the default **replicate_minimal_columns** setting of on does not send the command when there are no changes to any columns in the row. To restore the original behavior, set **replicate_minimal_columns** off for the connection.

For example, to enable **replicate_minimal_columns** for the connection to the `pubs2` database in the SYDNEY_DS data server:

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

replicate_minimal_columns can affect trigger processing if you expect triggers to fire even if there is no change in values to any columns in the row.

You can use **admin config** to display the **replicate_minimal_columns** setting.

Note: When you set **dsi_compile_enable** 'on', Replication Server ignores the **replicate_minimal_columns** setting.

Limitations for Dynamic SQL

There are several limitations to be aware of when you use dynamic SQL.

- 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.

Disable Dynamic SQL

There are several commands you can use to disable 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.

Advanced Services Option

Replication Server includes the Advanced Services Option which contains enhancements to replication performance.

To activate any of the enhancements in the Advanced Services Option, you must have the REP_HVAR_ASE license file. See *Replication Server Installation Guide > Planning Your Installation > Obtaining a License*.

See also

- *High Volume Adaptive Replication to Adaptive Server* on page 216
- *Enhanced Retry Mechanism* on page 224
- *Enhanced DSI Efficiency* on page 229
- *Enhanced RepAgent Executor Thread Efficiency* on page 230
- *Enhanced Distributor Thread Read Efficiency* on page 231
- *Enhanced Memory Allocation* on page 232
- *Increase Queue Block Size* on page 232

High Volume Adaptive Replication to Adaptive Server

Replication Server includes high volume adaptive replication (HVAR), which provides better performance compared to the current continuous replication mode when replicating into databases with identical database schema.

In continuous replication mode, Replication Server sends each logged change to the replicate database according to the log order in primary database. HVAR achieves better performance by using:

- Compilation – rearranges replicate data by each table, and each **insert**, **update**, and **delete** operation, and compiling the operations into net-row operations.

- Bulk apply – applies the net result of the compilation operations in bulk using the most efficient bulk interface for the net result. Replication Server uses an in-memory net-change database to store the changes, which it then applies to the replicate database.

Instead of sending every logged operation, compilation removes the intermediate **insert**, **update**, or **delete** operations in a group of operations and sends only the final compiled state of a replicated transaction. Depending on the transaction profile, this generally means that Replication Server sends a smaller number of commands to the replicate database to process.

HVAR groups as many compilable transactions as possible, compiles the transactions in the group into a net change, and then uses the bulk interface in the replicate database to apply the net changes to the replicate database.

As Replication Server compiles and combines a larger number of transactions into a group, bulk operation processing improves; therefore, replication throughput and performance also improves. You can adjust group sizes to control the amount of data that is grouped together for bulk apply.

HVAR is especially useful for creating online transaction processing (OLTP) archiving and reporting systems where the replicate databases have the same schemas as the primary databases.

Database and Platform Support

HVAR supports replication into Adaptive Server 12.5 and later and you can achieve optimal performance using 64-bit hardware platforms.

See *Replication Server 15.5 New Features Guide > New Features in Replication Server Version 15.5 > Operating System and Platform Support > 64-Bit Support*.

HVAR Compilation and Bulk Apply

During compilation, HVAR rearranges data to be replicated by clustering the data together based on each table, and each **insert**, **update**, and **delete** operation, and then compiling the operations into net row operations.

HVAR distinguishes different data rows by the primary key defined in a replication definition. If there is no replication definition, all columns except for `text` and `image` columns are regarded as primary keys.

For the combinations of operations found in normal replication environments, and given a table and row with identical primary keys, HVAR follows these compilation rules for operations:

- An **insert** followed by a **delete** results in no operation.
- A **delete** followed by an **insert** results in no reduction.
- An **update** followed by a **delete** results in a **delete**.

- An **insert** followed by an **update** results in an **insert** where the two operations are reduced to a final single operation that contains the results of the first operation, overwritten by any differences in the second operation.
- An **update** followed by another **update** results in an **update** where the two operations are reduced to a final single operation that contains the results of the first operation, overwritten by any differences in the second operation.

Other combinations of operations result in invalid compilation states.

Example 1

This is an example of log-order, row-by-row changes. In this example, `T` is a table created earlier by the command: **create table** `T(k int , c int)`

```
1. insert T values (1, 10)
2. update T set c = 11 where k = 1
3. delete T where k = 1
4. insert T values (1, 12)
5. delete T where k =1
6. insert T values (1, 13)
```

With HVAR, the **insert** in 1 and the **update** in 2 can be converted to **insert** `T values (1, 11)`. The converted **insert** and the **delete** in 3 cancel each other and can be removed. The **insert** in 4 and the **delete** in 5 can be removed. The final compiled HVAR operation is the last **insert** in 6:

```
insert T values (1, 13)
```

Example 2

In another example of log-order, row-by-row changes:

```
1. update T set c = 14 where k = 1
2. update T set c = 15 where k = 1
3. update T set c = 16 where k = 1
```

With HVAR, the **update** in 1 and 2 can be reduced to the **update** in 2. The updates in 2 and 3 can be reduced to the single **update** in 3 which is the net-row change of k = 1

Replication Server uses an **insert**, **delete**, and **update** table in an in-memory net-change database to store the net row changes which it applies to the replicate database. Net row changes are sorted by replicate table and by type of operations—**insert**, **update**, or **delete**—and are then ready for bulk interface. HVAR loads **insert** operations into the replicate table directly. Since Adaptive Server does not support bulk **update** and **delete**, HVAR loads **update** and **delete** operations into temporary worktables that HVAR creates inside the replicate database. HVAR then performs **join-update** or **join-delete** operations with the replicate tables to achieve the final result. The work tables are created and dropped dynamically.

In Example 2, where compilation results in `update T set c = 16 where k = 1`:

1. HVAR creates the `#rs_uT(k int, c int)` worktable.
2. HVAR performs an **insert** into the worktable with this statement:

```
insert into #rs_uT(k, c) location 'idemo.db' {select * from rs_uT}
```

3. HVAR performs the **join-update**:

```
update T set T.c=#rs_uT.c from T,#rs_uT where T.k=#rs_uT.k
```

As HVAR compiles and combines a larger number of transactions into a group, bulk operation processing improves; therefore, replication throughput and performance also improves. You can control the amount of data that HVAR groups together for bulk apply by adjusting HVAR sizes with configuration parameters.

There is no data loss, although HVAR does not apply row changes in the same order in which the changes are logged because for:

- Different data rows, the order in which the row changes are applied does not affect the result.
- The same row, applying **delete** before **insert** after compilation maintains consistency.

Net-Change Database

Replication Server has a net-change database that acts as an in-memory repository for storing the net row changes of a transaction, that is, the compiled transaction.

There is one net-change database instance for each transaction. Each replicate table can have up to three tracking tables within a net-change database. You can inspect the net-change database and the tables within the database to monitor HVAR replication and troubleshoot problems.

Monitor the Net-Change Database

Use the **sysadmin cdb** command to monitor a net-change database and to access net-change database instances.

See *Replication Server Reference Manual > Replication Server Commands > **sysadmin cdb***.

HVAR Processing and Limitations

HVAR applies only the net-row changes of a transaction while maintaining the original commit order, and guarantees transactional consistency even as it skips intermediate row changes.

This has several implications:

- To avoid high memory consumption by HVAR, Replicaton Server processes and applies large transaction through the continuous replication mode instead of HVAR. The threshold for large transactions depends on the size of the SQT cache that you can set with **dsi_sqt_max_cache_size** and the size of the net-change database that you can control with **dsi_compile_max_cmds** and **dsi_cdb_max_size**.
- **Insert** triggers do not fire, as the HVAR process performs a bulk load of net new rows directly into the table. **Update** and **delete** triggers continue to fire when Replication Server applies the net results of compilation to the replicate database. However, row

modifications that Replication Server compiles, and that are no longer in the net results, are invisible to the triggers. Triggers can detect only the final row images.

Suppose you use Replication Server to audit user updates using a `last_update_user` column in a table schema with a trigger logic that associates a user to any column in the table modified by the user. If userA modifies `colA` and `colC` in the table and then userB modifies `colB` and `colD`, when the trigger fires, the trigger logic can detect only the last user who modified the table, and therefore the trigger logic associates userB as the user that modified all four columns. If you define triggers that contain similar logic where every individual row modification must be detected, you may have to disable HVAR compilation for that table.

- HVAR does not apply row changes in the same order in which the row changes are logged. To apply changes to a replicated table in log order, disable HVAR compilation for that table.
- If there are referential constraints on replicate tables, you must specify the constraints in replication definitions. To avoid constraint errors, HVAR loads tables according to replication definitions.
- Replication Server does not support customized function strings or any parallel DSI serialization methods, except for the default **wait_for_commit** method, when you enable HVAR. HVAR treats customized function strings as noncompilable commands.
- Replication Server reverts to log-order row-by-row continuous replication when it encounters:
  - Noncompilable commands – stored procedures, SQL statements, data definition language (DDL) transactions, system transactions, and Replication Server internal markers.
  - Noncompilable transactions – a transaction that contains noncompilable commands.
  - Noncompilable tables – tables with HVAR disabled, with modified function strings, and with referential constraint relationships with tables that HVAR cannot compile.
- If the replication definition does not include the **replicate minimal columns** clause, HVAR automatically changes a primary-key **update** to a **delete** followed by an **insert**. A primary-key update is either one of:
  - An update that affects the primary key of a table where the primary key is defined in the replication definition of the table, or,
  - An update that affects any column, except for `text` and `image` columns, when no replication definition exists. In this case, Replication Server assumes all the columns are part of the primary key since there is no specific primary-key definition from a replication definition.
- If the replication definition includes the **replicate minimal columns** clause, and if the group being compiled by HVAR contains an **update** command that modifies the primary-key columns, HVAR automatically identifies the table as noncompilable at runtime for the remaining portion of the group. The **update** operation applied to the table is noncompilable because HVAR cannot transform the **update** to a pair of operations consisting of a **delete** and an **insert**. Within the transaction group that HVAR is processing, HVAR can successfully compile into the net-change database all operations that HVAR

processed before HVAR encountered the noncompilable primary-key **update** operation. However, within the transaction group, HVAR marks as noncompilable, the initial noncompilable primary-key **update** and all operations that follow it. The noncompilable state of the table is transient and lasts only for the duration of the same transaction group that HVAR is processing .

- HVAR ignores parameters, such as **dsi_partition_rule** that can stop transaction grouping.
- If errors occur during HVAR processing, Replication Server retries compilation with progressively smaller transaction groups until it identifies the transaction that failed compilation, then applies the transaction using continuous replication.
- To realize performance benefits, keep the primary and replicate databases synchronized to avoid the overhead of additional processing by Replication Server when errors occur. You can set **dsi_command_convert** to **i2di,u2di** to synchronize the data although this also incurs a processing overhead. If the databases are synchronized, reset **dsi_command_convert** to **none**.
- HVAR performs row-count validation to ensure replication integrity. The row-count validation is based on compilation. The expected row count is the number of rows remaining after compilation.
- When there are columns with `identity` datatype in a replication definition, Replication Server executes these commands in the replicate database:
  - **set identity_insert** *table_name* **on** before identity column inserts and **set identity_insert** *table_name* **off** after identity column inserts.
  - **set identity_update** *table_name* **on** before identity column updates and **set identity_update** *table_name* **off** after identity column updates.

See also

- *Tables with Referential Constraints* on page 227

Enable HVAR

Use **dsi_compile_enable** to enable HVAR for the connection to the replicate database.

If you set **dsi_compile_enable** off, Replication Server uses continuous log-order, row-by-row replication mode. For example, set **dsi_compile_enable** off for an affected table if replicating net-row changes causes problems, such as when there is a trigger on the table that requires all operations on the table to be replicated in log order, and therefore compilation is not allowed.

Note: When you set **dsi_compile_enable** on, Replication Server disables **dsi_cmd_prefetch** and **dsi_num_large_xact_threads**.

To enable and configure HVAR at the database level to affect only the specified database, enter:

```
alter connection to data_server.database
set dsi_compile_enable to ‘on’
go
```

You can also enable and configure HVAR at the server or table levels.

- Server level – affects all database connections to Replication Server:

```
configure replication server
set dsi_compile_enable to ‘on’
```

- Table level – affects only the replicate tables you specify. If you specify a parameter at both the table level and database level, the table-level parameter takes precedence over the database-level parameter. If you do not specify a table-level parameter, the setting for the parameter applies at the database level. To set a parameter for a table, use **alter connection** and the **for replicate table named** clause, for example:

```
alter connection to data_server.database
for replicate table named dbo.table_name
set dsi_compile_enable to ‘on’
```

  Using the **for replicate table name** clause alters connection configuration at the table level. The configuration changes apply to replicate data from all the subscriptions and all the replication definitions of the tables you specify.

Note: For table-level configuration, you can use only **alter connection**, as Replication Server does not support the **for** clause with **create connection**.

After you execute **dsi_compile_enable**, suspend and resume the connection to the replicate database.

HVAR Configuration Parameters

Replication Server automatically sets the Sybase-recommended default values of several parameters. You can change the values of these parameters to tune replication performance.

You must execute a separate **alter connection** command for each parameter you want to change. Do not enter more than one parameter after entering **alter connection**.

HVAR automatically sets the Sybase-recommended default values for **dsi_cdb_max_size**, **dsi_compile_max_cmds**, **dsi_bulk_threshold**, **dsi_command_convert**, and **dsi_compile_retry_threshold**. However, you can specify your own values to tune performance in your replication environment:

See *Replication Server Reference Manual > Replication Server Commands >* ***alter connection*** for full descriptions of the parameters.

dsi_bulk_threshold

dsi_bulk_threshold specifies the number of net row change commands after compilation has occurred on a table for a command type, that when reached, triggers Replication Server to use bulk copy-in on that table for the same command type.

Default is 20 net row change commands.

Example:

```
alter connection to SYDNEY_DS.pubs2
set dsi_bulk_threshold to ‘15’
go
```

dsi_cdb_max_size

dsi_cdb_max_size specifies, in megabytes, the maximum size of a transaction that HVAR can compile if the transaction does not exceed the DSI SQT cache or if the number of commands in the transaction does not exceed **dsi_compile_max_cmds**.

When the size of transactions in the current group that HVAR is compiling reaches **dsi_compile_max_cmds**, HVAR starts a new group. If there is no more data to read, and even if the group does not reach the maximum size set in **dsi_cdb_max_size**, HVAR completes grouping the current set of transactions into the current group.

Default is 1024MB.

Example:

```
alter connection to SYDNEY_DS.pubs2
set dsi_cdb_max_size to ‘2048’
go
```

dsi_compile_max_cmds

dsi_compile_max_cmds specifies, in number of commands, the maximum size of a transaction that HVAR can compile if the transaction does not exceed the DSI SQT cache or if the transaction size does not exceed **dsi_cdb_max_size**. Replication Server replicates noncompilable transactions through the continuous replication mode.

When the number of commands in the current group that HVAR is compiling reaches **dsi_compile_max_cmds**, HVAR starts a new group. If there is no more data to read, and even if the group does not reach the maximum number of commands set in **dsi_compile_max_cmds**, HVAR completes grouping the current set of transactions into the current group.

Default is 10,000 commands.

Example:

```
alter connection to SYDNEY_DS.pubs2
set dsi_compile_max_cmds to ‘50000’
go
```

dsi_compile_retry_threshold

dsi_compile_retry_threshold specifies a threshold value for the number of commands in a group. If the number of commands in a group containing failed transactions is smaller than the value of **dsi_compile_retry_threshold**, Replication Server does not retry processing the group in HVAR mode, and saves processing time, thus improving performance. Instead, Replication Server switches to continuous replication mode for the group. Continuous replication mode sends each logged change to the replicate database according to the primary database log order.

Default is 100 commands.

Example:

```
alter connection to SYDNEY_DS.pubs2
set dsi_compile_retry_threshold to '200'
go
```

dsi_command_convert

dsi_command_convert – specifies how to convert a replicate command. A combination of these operations specifies the type of conversion:

- **d** – delete
- **i** – insert
- **u** – update
- **t -** truncate
- **none** – no operation

Combinations of operations for **dsi_command_convert** include **i2none**, **u2none**, **d2none**, **i2di**, **t2none**, and **u2di**. The operation before conversion precedes the "2" and the operations after conversion are after the "2". For example:

- **d2none** – do not replicate the **delete** command. With this option, you need not customize the **rs_delete** function string if you do not want to replicate **delete** operations.
- **i2di**,**u2di** – convert both **insert** and **update** to **delete** followed by **insert**, which is equivalent to an autocorrection. If you disable row count validation by setting **dsi_row_count _validation** off, Sybase recommends that you set **dsi_command_convert** to **i2di,u2di** to avoid duplicate key errors and allow autosynchronization of databases during replication.
- **t2none** – do not replicate **truncate table**.

Default for **dsi_command_convert** is **none** which means there is no command conversion.

Example:

```
alter connection to SYDNEY_DS.pubs2
set dsi_command_convert to 'i2di,u2di'
go
```

Enhanced Retry Mechanism

The enhanced retry mechanism improves replication performance by reducing the number of times Replication Server retries compilation and bulk apply.

HVAR attempts to group as many compilable transactions as possible together, compile the transactions in the group into a net change, and then use the bulk interface in the replicate database to apply the net changes to the replicate database. HVAR invokes the retry mechanism when a replicate transaction resulting from HVAR processing fails. If transactions in a group fail, HVAR splits the group into two smaller groups of equal size, and retries the compilation and bulk application on each group. The retry mechanism identifies the failed transaction, allows Replication Server to execute error action mapping, and applies all transactions preceding the failed transaction in case DSI shuts down.

The net-change database in HVAR acts as an in-memory repository for storing the net row changes of a transaction, that is, the compiled transaction. The content of the net-change

database is an aggregation of commands from different primary transactions that HVAR is not applying in log order. Therefore, there is no means to identify a failed transaction without a retry mechanism. The retry mechanism splits a group and retries compilation and bulk application continuously as long as a transaction in the group fails. This continuous retry process can degrade performance.

The enhanced retry mechanism splits the group into three groups of equal size when HVAR encounters a group containing transactions that fail, enabling the mechanism to more efficiently identify the group containing the failed transaction.

In addition, you can use the **dsi_compile_retry_threshold** parameter to specify a threshold value for the number of commands in a group. If the number of commands in a group containing failed transactions is smaller than the value of **dsi_compile_retry_threshold**, Replication Server does not retry processing the group in HVAR mode, and saves processing time, thus improving performance. Instead, Replication Server switches to continuous replication mode for the group. Continuous replication mode sends each logged change to the replicate database according to the primary database log order.

Memory Consumption Control

To reduce memory consumption in HVAR, control the size of compilable groups.

Memory consumption refers to Replication Server data structures such as the net-change database, and the data that the structures store. Net-change databases are in-memory data structures. Net-change database memory consumption can increase drastically when Replication Server compiles commands applied on a table with a large number of columns, or tables with large `text` and `image` datatype values. For example, compiling 1,000,000 rows in a table with 100 columns may consume approximately 10 times more memory than compiling the same number of rows in a table with 10 columns. Replication performance suffers when there is insufficient memory available for other processes and modules.

Replication Server marks a transaction as noncompilable if the transaction is larger than the DSI SQT cache size. If a transaction can fit into the DSI SQT cache, Replication Server checks the size of the transaction against the values of **dsi_cdb_max_size** and **dsi_compile_max_cmds**. Replication Server marks the transaction as noncompilable if Replication Server estimates that the size of the net-change database the transaction requires is larger than **dsi_cdb_max_size**, or if the transaction contains more commands than **dsi_compile_max_cmds**. Replication Server applies this large noncompilable transaction in the continuous replication mode. Using the continuous replication mode avoids generating a single large net-change database to accommodate the large transaction and reduces memory consumption.

Replication Server tries to group as many compilable transactions as possible into a compilable group. Replication Server also uses **dsi_cdb_max_size** and **dsi_compile_max_cmds** as thresholds for the size of compilable groups. Once a group reaches either the size you set in **dsi_cdb_max_size** or **dsi_compile_max_cmds**, Replication

Server stops compiling transactions into the group and applies each compilable group as a single transaction to the replicate database.

Memory Control Parameters and Replication Server Processing

Replication modes and actions depend on the values you set for memory control parameters.

Replication Server Processing

1. Replication Server reads a transaction from the outbound queue and estimates the net-change database size.
2. Replication Server flags the transaction as compilable if the transaction only contains **insert**, **update** , and **delete** commands.
3. Replication Server flags a transaction as noncompliable when:
   - The number of commands in the transaction exceeds **dsi_compile_max_cmds**,
   - The estimated net-change database size for the transaction exceeds **dsi_cdb_max_size**, or
   - The transaction size is larger than the DSI SQT cache.

   Replication Server processes noncompilable transactions in continous log order mode.
4. After Replication Server checks if a transaction is compilable, it aggregates successive compilable transactions into a compilable group. However, Replication Server stops increasing the size of a compilable group based on two thresholds:
   - If Replication Server calculates that the number of commands in the group of compilable transaction it is processing added to the number of commands in the new transaction exceeds the **dsi_compile_max_cmds** threshold value, Replication Server closes and dispatches the group, and adds the new transaction to a new empty group. Otherwise, Replication Server adds the new compilable transaction to the group.
   - If the estimated size of the next net-change database resulting from aggregation of the new transaction to the new group exceeds **dsi_cdb_max_size**, Replication Server closes and dispatches the group, and adds the new transaction to a new empty group. Otherwise, Replication Server adds the new compilable transaction to the group.
5. If there are no more compilable transactions in the outbound queue, Replication Server immediately closes and dispatches the group it is processing. Replication Server does not wait for new transactions to enter the outboud queue.

Setting dsi_cdb_max_size to Different Values

Examples that show Replication Server applying a transaction with 100,000 updates on two tables. Table1 has 100 columns and requires approximately 4GB of memory, and Table2 has 10 columns requiring approximately one-tenth the memory—400MB.

dsi_cdb_max_size Value (MB)	Table Name	Impact on Replication Processing
1024 (default)	Table1	Replication Server applies the transaction in continuous log-order replication mode.

dsi_cdb_max_size Value (MB)	Table Name	Impact on Replication Processing
1024 (default)	Table2	Prerequisite: Set **memory_limit** in Replication Server to a value large enough to allow the construction of 400MB net-change databases. Replication Server applies the transaction using HVAR.
4096	Table1	Replication Server applies the transaction using continuous log order replication mode.
4096	Table2	Prerequisite: Set **memory_limit** in Replication Server to a value large enough to allow the construction of 400MB net-change databases. Replication Server applies the transaction using HVAR.

Tables with Referential Constraints

You can use a replication definition to specify tables that have referential contraints, such as a foreign key and other check constraints, so that HVAR is aware of these tables.

Usually the referencing table contains referential constraints for a referenced table within the same primary database. However, HVAR extends referential constraints support to referenced tables from multiple primary databases.

You can specify the referencing table in a replication definition for each primary database. However, if multiple referential constraints conflict with each other, Replication Server randomly selects one.

See also

- *HVAR Processing and Limitations* on page 219

Replication Definitions Creation and Alteration

Use the **create replication definition** command with the **references** parameter to specify the table with referential constraints.

create replication definition

```
   ...
   (column_name [as replicate_column_name]
       ...
       [map to published_datatype]] [quoted]
       [references [table_owner.]table_name [(column_name)]] …)
   ....]
```

Use the **alter replication definition** command with the **references** parameter to add or change a referencing table. Use the **null** option to drop a reference.

alter replication definition

```
   .....
add column_name [as replicate_column_name]
```

```
[map to published_datatype] [quoted]
[references [table_owner.]table_name [(column_name)]
...
| alter columns with column_name references
{[table_owner.]table_name [(column_name)] | NULL}
[, column_name references {[table_owner.]table_name [(column_name)]
| NULL}
     ...
```

For both **alter replication definition** and **create replication definition** with the **reference** clause, Replication Server:

- Treats the **reference** clause as a column property. Each column can reference only one table.
- Does not process the column name you provide in the *column_name* parameter within the **reference** clause.
- Does not allow referential constraints with cyclical references. For example, the original referenced table cannot have a referential constraint to the original referencing table.

During replication processing, HVAR loads:

- Inserts to the referenced tables before the referencing table you specify in the replication definition.
- Deletes to the referenced tables after the table you specify in the replication definition.

In some cases, updates to both tables fail because of conflicts. To prevent HVAR from retrying replication processing, and thus decreasing performance, you can:

- Stop replication updates by setting **dsi_command_convert** to "u2di," which converts updates to deletes and inserts.
- Turn off **dsi_compile_enable** to avoid compiling the affected tables.

HVAR cannot compile tables with customized function strings, and tables that have referential constraints to an existing table that it cannot compile. By marking out these tables, HVAR optimizes replication processing by avoiding transaction retries due to referential constraint errors.

Display HVAR Information

You can display information on configuration parameter properties and table references.

Display Configuration Parameter Properties

Use **admin config** to view information about database-level and table-level configuration parameters as shown in the examples.

- Database-level:
  - To display all database-level configuration parameters for the connection to the `nydb1` database of the NY_DS data server (`NY_DS.nydb1`), enter:

    ```
    admin config, "connection", NY_DS, nydb1
    ```

- To verify that **dsi_compile_enable** is **on** for the connection to `NY_DS.nydb1`, enter:
  ```
  admin config, “connection”, NY_DS, nydb1,dsi_compile_enable
  ```
- To display all the database-level configuration parameters that have "enable" as part of the name, such as **dsi_compile_enable**, enter:
  ```
  admin config, “connection”, NY_DS, nydb1,"enable"
  ```

  Note: You must enclose "enable" in quotes because it is a reserved word in Replication Server. See *Replication Server Reference Manual > Topics > Reserved Words*.

- Table-level:
  To display all configuration parameters after using **dsi_command_convert** to set **d2none** on the `tb1` table in the `nydb1` database of the NY_DS data server, enter:
  ```
  admin config, “table”, NY_DS, nydb1
  ```

See *Replication Server Reference Manual > Replication Server Commands > **admin config***.

Display Table References

Use **rs_helprep**, which you can execute on the Replication Server System Database (RSSD), to view information about table references and RTL information.

To display information about the **authors_repdef** replication definition created using **create replication definition**, enter:

```
rs_helprep authors_repdef
```

See *Replication Server Reference Manual > RSSD Stored Procedures > **rs_helprep***.

System Table Support in Replication Server

Replication Server uses the `rs_tbconfig` table to store support table-level configuration parameters, and the `ref_objowner` and `ref_objname` columns in the `rs_columns` table to support referential constraints.

See *Replication Server Reference Manual > Replication Server System Tables* for full table descriptions.

Mixed-Version Support and Backward Compatibility

HVAR can replicate referential constraints specified in replication definitions only if the outbound route version is later than 15.5.

HVAR works if the outbound route version is earlier than 15.5. However, no referential constraint information is available to a Replication Server with version 15.5 or later.

Continuous replication is the default replication mode available to all supported versions of Replication Server. HVAR is only available with Replication Server 15.5 and later.

Enhanced DSI Efficiency

Enabling **dsi_cmd_prefetch** reduces data replication latency which decreases the length of time that Replication Server waits for results from the replicate data server through the

ct_results routine, and subsequently reduces the length of time the data server waits for Replication Server.

dsi_cmd_prefetch works by :

- Allowing Replication Server to prepare the next batch of commands for the replicate data server before Replication Server processes results of the current batch from the replicate data server.
- Improving concurrency between the DSI executor (DSI/E) and DSI scheduler (DSI/S) threads.

Set **dsi_cmd_prefetch** on with **alter connection** or **create connection**.

For example, to enable **dsi_cmd_prefetch** for the connection to the `pubs2` database in the SYDNEY_DS data server, enter:

```
alter connection to SYDNEY_DS.pubs2
set dsi_cmd_prefetch to ‘on’
```

Default: off.

dsi_cmd_prefetch is a dynamic parameter. You need not suspend and resume the database connection after you enable the parameter for the change to take effect.

If you also tune your data server to enhance performance, you may gain an additional performance increase when you enable **dsi_cmd_prefetch**.

Note: When you set **dsi_compile_enable** to ‘on’, Replication Server ignores what you set for **dsi_cmd_prefetch**.

Enhanced RepAgent Executor Thread Efficiency

You can improve performance in Replication Server by using the NRM thread to normalize and pack Log Transfer Language (LTL) commands in parallel with parsing by the RepAgent executor thread.

When you instruct Replication Server to enable the NRM thread, a thread splits from the RepAgent executor thread to become the NRM thread. Parallel processing by the NRM thread reduces the response time of the RepAgent executor thread.

After you enable the NRM thread, you can specify the memory available for the message queue from the RepAgent executor thread to the NRM thread.

Enable NRM Thread

Set **nrm_thread** to on with **configure replication server** to enable the NRM thread.

Enter:

```
configure replication server
set nrm_thread to ‘on'
```

Default: off

nrm_thread is a server-level parameter. You must restart Replication Server after you change the parameter value.

Specify Memory Available to RepAgent Executor

After you set **nrm_thread** to on, use the **exec_nrm_request_limit** parameter with **configure replication server** or **alter connection** to specify the total amount of memory available to RepAgent Executor thread for the message queue the NRM threads.

If the total amount of memory used by commands on the message queue is larger than than what you specify with **exec_nrm_request_limit**, the RepAgent Executor thread sleeps,and waits for memory to become available. As the NRM thread processes commands on the message queue, it frees the memory for the RepAgent Executor thread.

For example, to set **exec_sqm_nrm_request_limt** to 1GB for the connection to the `pubs2` database in the SYDNEY_DS data server, enter:

```
alter connection to SYDNEY_DS.pubs2
set exec_nrm_request_limit to ‘1073741824’
```

exec_nrm_request_limit values:

- Default:
  - 32-bit – 1,048,576 bytes (1MB)
  - 64-bit – 8,388,608 bytes (8MB)
- Maximum – 2,147,483,647 bytes (2GB)
- Minimum – 16,384 bytes (16KB)

After you change the configuration for **exec_nrm_request_limit**, suspend and resume the Replication Agent. To suspend and resume:

- RepAgent for Adaptive Server, execute in Replication Server; **sp_stop_rep_agent** and then **sp_start_rep_agent**.
- Replication Agent for supported non-ASE databases; execute **suspend** and then **resume** in the Replication Agent.

Enhanced Distributor Thread Read Efficiency

Enables the distributor (DIST) thread to read SQL statements directly from the Stable Queue Thread (SQT) cache. This reduces the workload from SQT and the dependency between the two, and improves the efficiency of both SQT and DIST.

Use the **dist_direct_cache_read** parameter with **configure replication server** to use this enhancement:

Enter:

```
configure replication server
set dist_direct_cache_read to ‘on’
```

By default, **dist_direct_cache_read** is set to ‘off’. If you disable the parameter, the distributor thread requests SQL statements from SQT through the message queue. This leads to inbound and outbound queue contention.

dist_direct_cache_read is a server-level parameter. You must restart Replication Server after you enable or disable the parameter.

Enhanced Memory Allocation

Use the **mem_reduce_malloc** parameter with **configure replication server** to allocate memory in larger chunks in Replication Server.

This reduces the number of memory allocations needed, and leads to improved Replication Server performance.

Enter:

```
configure replication server
set mem_reduce_malloc to ‘on’
```

By default, **mem_reduce_malloc** is set to ‘off’.

mem_reduce_malloc is a dynamic parameter. You do not need to suspend or resume the database connection when you change parameter settings.

Increase Queue Block Size

Increase the queue block size to improve replication performance.

The queue block size is the number of bytes in a contiguous block of memory used by stable queue structures. Setting a larger queue block size allows Replication Server to process more transactions in a single block. You can increase the queue block size from the default of 16KB to 32KB, 64KB, 128KB, or 256KB. Performance improvement is also dependent on the transaction profile and the environment.

Note: You must have the Advanced Services Options license, named REP_HVAR_ASE, to use the increase queue block size feature.

Recommendations

Sybase strongly recommends that you:

- Verify you have sufficient memory before you increase the queue block size.
- Experiment with different queue block sizes to determine the optimum value for your replication system.

Restrictions

- Make sure that there is no data flowing into Replication Server while the queue block size change is in progress.
- You cannot change the queue block size while a subscription is being materialized, if dematerialization is in progress, or if routes are being created or destroyed. The queue

block size change terminates with an error message while Replication Server continues operating.
- Once you start the procedure to change the queue block size, Replication Server does not accept another command to change the queue block size until the first change is completed.
- Do not use any other procedures to change the queue block size in the RSSD directly, as these procedures may result in inconsistencies in the queue block size configuration and cause Replication Server to shut down.

Note: All queues are drained after the block size changes.

Changing the Queue Block Size

Modifying the queue block size is a major change to the Replication Server configuration and affects all connections to the Replication Server. You must suspend log transfer and quiesce Replication Server.

In the queue block size change procedure, "upstream" refers to all replication system components that feed data to the Replication Server where you want to change the queue block size and "downstream" refers to the components that receive data from the affected Replication Server.

1. To maintain data integrity, stop data flowing into the Replication Server you want to configure before you change the queue block size.:
   a) Suspend log transfer from all Replication Agents to the Replication Server you want to configure.
   b) Suspend all upstream log transfer from Replication Agents.
   c) Quiesce all upstream Replication Servers.
   d) Suspend all incoming routes to the Replication Server you want to configure.
   e) Quiesce the Replication Server you want to configure.
2. Use **configure replication server** with the **set block_size to** '*value*' clause to set the queue block size on the Replication Server you want to configure.

   This command:
   - Verifies that there is no subscription materialization in progress.
   - Verifies that all log transfer is suspended.
   - Verifies that all incoming routes are suspended.
   - Verifies that the Replication Server is quiesced.
   - Purges queues.
   - Zeros the values in the `rs_locater` RSSD system table to allow Replication Agents to resend transactions that may have not been applied to the replicate database when you started the queue block size change procedure.
   - Sets the queue block size to the value you entered.

- (Optional) If you include the **with shutdown** option, Replication Server shuts down. The queue block size change takes effect when you restart the Replication Server. Shutting down ensures that Replication Server clears all memory.

3. After you change the queue block size to a larger value, delete, or delete and recreate raw partitions that you had created with the smaller block size value.

   Partitions register the correct number of segments only if you create the partitions after you change the block size.

4. Resume data flow:

   a) If you used the **with shutdown** option, restart the Replication Server.
   b) Resume log transfer from Replication Agents.
   c) Resume all incoming routes.

5. Check for data loss at all downstream Replication Server RSSDs and data servers. Usually, there is data loss from the RSSD of the Replication Server you configured. Ignore the data loss from a replicate RSSD that receives data from the RSSD of the configured Replication Server.

   Follow the procedures to fix data loss at the data servers. If there is data loss at an RSSD, you see a message similar to this in the log of the affected Replication Server:

   ```
   E. 2010/02/12 14:12:58. ERROR #6067 SQM(102:0 primaryDS.rssd) - /
   sqmoqid.c(1071)
   Loss detected for replicateDS.rssd from primaryDS.RSSD
   ```

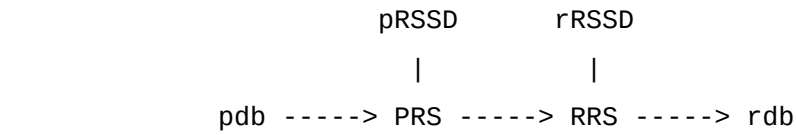
   replicateDS is the replicate data server name and *primaryDS* is the primary data server name.

Increasing Queue Block Size in a Simple Replication System

Set the queue block size of the primary and replicate Replication Servers in a simple replication system.

The replication system consists of:

- primary database – `pdb`
- replicate database – `rdb`
- primary Replication Server – PRS
- RSSD of primary Replication Server – `pRSSD`
- replicate Replication Server – RRS
- RSSD of replicate Replication Server – `rRSSD`



In this example, RSSD refers to both Adaptive Server as the Replication Server System Database (RSSD) and SQL Anywhere® as the Embedded Replication Server System

Database (ERSSD). See the *Replication Server Reference Manual* for the full syntax, examples, and usage information for all commands.

1. Configure the primary Replication Server:
   a) Suspend log transfer from all Replication Agents. At the primary Replication Server, execute:
      ```
      suspend log transfer from all
      ```
   b) Quiesce the primary Replication Server:
      ```
      admin quiesce_force_rsi
      ```
   c) Set the queue block size at the primary Replication Server to 64KB:
      ```
      configure replication server
      set block_size to '64'
      ```
      (Optional) Use the **with shutdown** option to set the block size and shut down the primary Replication Server. For example:
      ```
      configure replication server
      set block_size to '64' with shutdown
      ```
   d) Look at the transaction log to verify that the primary Replication Server is not materializing, that log transfer and routes are suspended, and that the primary Replication Server is quiesced.
   e) Restart the primary Replication Server if you have shut it down. See *Replication Server Administration Guide Volume 1 > Manage a Replication System > Starting Replication Server*.
   f) Look at the primary Replication Server transaction log to verify that the block size is changed.
   g) Resume log transfer to allow Replication Agents to connect to the primary Replication Server. At the primary Replication Server execute:
      ```
      resume log transfer from all
      ```
   h) Check the replicate Replication Server log file for information about data losses. Ignore data loss occurring from the primary Replication Server RSSD to the replicate Replication Server RSSD by executing the **ignore loss** command on the replicate Replication Server:
      ```
      ignore loss from PRS.pRSSD to RRS.rRSSD
      ```
2. Configure the replicate Replication Server:
   a) Suspend log transfer from all Replication Agents. At the primary Replication Server and at the replicate Replication Server, execute:
      ```
      suspend log transfer from all
      ```
   b) Quiesce the primary Replication Server:
      ```
      admin quiesce_force_rsi
      ```
   c) At all Replication Servers that originate routes to the replicate Replication Server, suspend the routes:
      ```
      suspend route to RRS
      ```

d) Quiesce the replicate Replication Server:

```
admin quiesce_force_rsi
```

e) Set the block size at the replicate Replication Server to 64KB:

```
configure replication server
set block_size to '64'
```

(Optional) Use the **with shutdown** option to shut down the replicate Replication Server. For example:

```
configure replication server
set block_size to '64' with shutdown
```

f) Look at the transaction log to verify that the replicate Replication Server is not materializing, that log transfer and routes are suspended, and that the replicate Replication Server is quiesced.

g) Restart the replicate Replication Server if you have shut it down.

h) Look at the the replicate Replication Server transaction log to verify that the block size is changed.

i) Resume log transfer to allow Replication Agents to connect to the replicate Replication Server. At the replicate Replication Server, execute:

```
resume log transfer from all
```

j) Resume log transfer to allow Replication Agents to connect to the primary Replication Server. At the primary Replication Server execute:

```
resume log transfer from all
```

k) Resume the routes you suspended:

```
resume route to RRS
```

l) Check the the primary and replicate Replication Server log files for information about data losses. Ignore data loss occurring between the primary RSSD and the replicate RSSD if the replicate RSSD is replicated to the primary RSSD by executing the **ignore loss** command on the primary Replication Server.

```
ignore loss from RRS.rRSSD to PRS.pRSSD
```

See also

- *Ignore a Loss* on page 345

Increasing Queue Block Size in a Replication System with an Intermediate Route

Set the queue block size of the primary Replication Server in a replication system with an intermediate route.

The replication system consists of:

- primary database – `pdb`
- replicate database – `rdb`
- primary Replication Server – PRS

- RSSD of primary Replication Server – `pRSSD`
- replicate Replication Server – RRS
- RSSD of replicate Replication Server – `rRSSD`
- intermediate Replication Server – IRS
- RSSD of intermediate Replication Server – `iRSSD`

```
          pRSSD       iRSSD       rRSSD
            |           |           |
pdb -----> PRS -----> IRS -----> RRS -----> rdb
```

In this example, RSSD refers to both Adaptive Server as the Replication Server System Database (RSSD) and SQL Anywhere as the Embedded Replication Server System Database (ERSSD). See the *Replication Server Reference Manual* for the full syntax, examples, and usage information for all commands.

1. Suspend log transfer from all Replication Agents. At the primary Replication Server, execute:

   ```
   suspend log transfer from all
   ```

2. Quiesce PRS:

   ```
   admin quiesce_force_rsi
   ```

3. Set the block size at the primary Replication Server to 64KB:

   ```
   configure replication server
   set block_size to ‘64’
   ```

   (Optional) Use the **with shutdown** option to set the block size and shut down the primary Replication Server. For example:

   ```
   configure replication server
   set block_size to ‘64’ with shutdown
   ```

4. Look at the transaction log to verify that the primary Replication Server is not materializing, that log transfer and routes are suspended, and that the primary Replication Server is quiesced.
5. Restart the primary Replication Server if you have shut it down. See *Replication Server Administration Guide Volume 1 > Manage a Replication System > Starting Replication Server*.
6. Look at the primary Replication Server transaction log to verify that the block size is changed.
7. Resume log transfer to allow Replication Agents to connect to the primary Replication Server. At the primary Replication Server execute:

   ```
   resume log transfer from all
   ```

8. Check the intermediate and replicate Replication Server log files for information about data losses. Ignore data loss occurring from the primary Replication Server RSSD to the replicate Replication Server and from the primary RSSD to the intermediate RSSD by executing the **ignore loss** command twice on the intermediate Replication Serve:

```
ignore loss from PRS.pRSSD to RRS
go
ignore loss from PRS.pRSSD to IRS.iRSSD
```

See also

- *Ignore a Loss* on page 345

Multi-Path Replication

Use multiple replication paths to increase replication throughput and performance, and reduce contention.

In a single-path replication environment, transactions replicate serially from the primary database to the replicate database to ensure the primary database transaction commit order, and therefore to ensure that the replicate database is consistent with the primary database. The serial mode of applying transactions to the replicate database remains, even though multiple applications typically execute their respective transactions in parallel at the primary database, or even if there are transactions arriving from multiple primary databases.

There are replication environments that can maintain data consistency within a subset of tables, without serializing all transactions that originate from the same primary database. A typical example of this environment is when different applications that access different sets of data modify a single primary database. The different sets of data within the subset of tables that are modified by a specific application continue to replicate serially. Data in different subsets of tables can replicate in parallel.

Multi-Path Replication™ supports the replication of data through different streams, while still maintaining data consistency within a path, but not adhering to the commit order across different paths.

A replication path encompasses all the components and modules between the Replication Server and the primary or replicate database. In multipath replication, you can create multiple primary replication paths for multiple Replication Agent connections from a primary database to one or more Replication Servers, and multiple replicate paths from one or more Replication Servers to the replicate database. You can configure multi-path replication in warm standby and multisite availability (MSA) environments. You can convey transactions over dedicated routes between Replication Servers to avoid congestion on shared routes, and you can dedicate an end-to-end replication path from the primary database through Replication Servers to the replicate database, to objects such as tables and stored procedures.

License

Multi-Path Replication is licensed as part of the Advanced Services Option. See *Replication Server Installation Guide > Planning Your Installation > Obtaining a License*.

System Requirements
Replication Server supports multipath replication between Adaptive Server databases where the primary data server is Adaptive Server 15.7 and later.

Multi-Path Replication Quick Start

Set up a multi-path replication replication system comprising of two primary and replicate paths for end-to-end replication.

1. Select or create two sets of tables or stored procedures that you want to replicate through two replication paths.
2. Use **rs_init** to add the primary and replicate Adaptive Server databases to the replication system.
3. Enable multithreaded RepAgent.
   At the primary Adaptive Server, enter:
   ```
   sp_config_rep_agent primary_database_name, 'multithread rep
   agent', 'true'
   ```
4. Set the number of replication paths for RepAgent.
   For example, to enable two paths, enter:
   ```
   sp_config_rep_agent primary_database_name, 'max number of
   replication paths', '2'
   ```
5. Create an alternate replication path from the primary database to Replication Server.
   a) Create the alternate physical RepAgent replication path named *alternate_path_name*.
      At the primary Adaptive Server, enter:
      ```
      sp_replication_path "primary_database_name", 'add',
      "alternate_path_name", "repserver_name",
      "repserver_user", "repserver_password"
      ```
   b) Create the corresponding alternate primary connection from Replication Server to the primary database and bind it to the alternate physical RepAgent replication path by using the same RepAgent replication path name—*alternate_path_name*.
      At the Replication Server, enter:
      ```
      create alternate connection to
      primary_dataserver.primary_database
      named primary_dataserver.alternate_path_name
      set error class to rs_sqlserver_error_class
      set function string class to rs_sqlserver_function_class
      set username to primary_db_maintenance_user
      set password to primary_db_maintenance_password
      with primary only
      ```
   The replication system contains two primary replication paths—the default and *alternate_path_name*
6. Create an alternate replicate connection from Replication Server to the replicate database using the same alternate replication path name—*alternate_path_name*.

```
create alternate connection to
replicate_dataserver.replicate_database
named replicate_dataserver.alternate_path_name
```

The replication system contains two replicate replication paths—the default and *alternate_path_name*

7. Bind one set of objects such as tables or stored procedures to the alternate replication path.

```
sp_replication_path pdb, 'bind', "table",
"[table_owner].table_name", "alternate_path_name"
```

The other set of objects uses the default replication path. You can only bind objects to alternate replication paths. All objects that you do not bind to an alternate replication path, use the default path instead.

8. Create a replication definition against the primary database.
For example to create the **authors_rep** replication definition for the `authors` table:

```
create replication definition authors_rep
with primary at primary_dataserver.primary_database
with all tables named 'authors'
...
go
```

If the default primary connection and the alternate primary connection are on different Replication Servers, create replication definitions on each Replication Server.

9. Create a subscription against the default primary connection and the default replicate connection.

```
create subscription subscription_default_path for
replication_definition
with primary at primary_dataserver.primary_database
with replicate at replicate_dataserver.replicate_database
```

10. Create a subscription against the alternate primary connection and the alternate replicate connection.

```
create subscription subscription_alternate_path for
replication_definition
with primary at primary_dataserver.alternate_path_name
with replicate at replicate_dataserver.alternate_path_name
```

Parallel Transaction Streams

Multi-path replication can improve replication performance as long as transactions can be divided into parallel streams, and do not have to be serially committed across different streams.

You can improve replication performance by:

- Dividing transactions into parallel replication paths to reduce congestion. You can divide transactions according to parallelization rules such as transaction attributes or derived data

values. For example, you can divide transactions by the session ID on the primary database or by the user who originated the transaction.

- Dedicating paths to specific objects such as tables or stored procedures.
- Dedicating a Replication Server to each path.
- Allocating dedicated paths or less congested paths for priority replication.

Default and Alternate Connections

In multipath replication, connections include the default and one or more alternate connections.

A connection that accepts data from a Replication Agent is a primary connection, and a connection that applies data to a database is replicate connection. A default or alternate connection can be a primary or replicate connection.

The default connection is the one that you create from a Replication Server to a specific primary or replicate database when you add the database to the replication domain. You can use **rs_init**, the Replication Manager Sybase Central plug-in, **create connection**, or **create connection ... using profile** to create the default connection depending on whether the data server is Adaptive Server or a supported non-ASE data server.

The default connection uses the data server and database names in the form of *dataserver.database* as the connection name, where *dataserver* and *database* are the actual data server and database names, respectively.

You can create multiple alternate connections after you create the default connection, which is required. Each alternate connection must have a unique name.

After you create an alternate connection, you can alter the connection properties or drop the connection. You can also display the status of all connections and create subscriptions for the connection.

Multiple Connections to the Replicate Database

Create multiple connections from Replication Server to the replicate database.

When you create multiple connections to the replicate database, each replication connection subscribes to one transaction stream. Transactions in the same stream adhere to the primary commit order. Transactions in different streams are applied in parallel and may not be in primary commit order.

Default and Alternate Replicate Connections

You can also create replicate connections from multiple Replication Servers to the same replicate database in the same replication domain. Only one Replication Server in the replication domain can possess and control the default replicate connection. You cannot create multiple default replicate connections from other Replication Servers in the domain. Other Replication Servers can have only alternate replicate connections.

After you create an alternate replicate connection, you can alter the connection properties or drop the connection. You can also display the status of all connections and create subscriptions for the connection.

Creating Alternate Replicate Connections

Use **create alternate connection** to create alternate connections from Replication Server to the replicate database.

Enter:

```
create alternate connection to dataserver.database
named conn_server.conn_db
[set error class [to] error_class
set function string class [to] function_class
set username [to] user
set password [to] pwd]
[set database_param [to] 'value']
```

where:

- *dataserver* and *database* – are the replicate data server and database.
- *conn_server.conn_db* – the alternate replicate connection, which comprises the data server name and a connection name.
  - If *conn_server* is different from *dataserver*, there must be an entry for *conn_server* in the interface file.
  - If *conn_server* is the same as *dataserver*, *conn_db* must be different from *database*.
  - Each replicate connection name must be unique in a replication system.
- **set function string class [to]** *function_class*, **set username [to]** *user*, and **set password [to]** *pwd* – existing clauses for **alter connection** and **create connection** that you can use when you create alternate connections.
  - If you omit these clauses, the alternate replicate connection inherits the values that you set with the default replicate connection.
  - If you omit these clauses when you create an alternate connection on a (current) Replication Server that is different from the (controller) Replication Server that controls the default connection, the current Replication Server returns an error.
  - The alternate connection can inherit the values from the default connection only if the same Replication Server controls both alternate and default connections.
  - If you do not set the maintenance user for the alternate connection, it inherits the default connection maintenance user. The alternate connection uses any new maintenance user that you specify for the alternate connection.
- *set param* – clauses for existing optional connection parameters for **alter connection** and **create connection**.
  - Any value you set for the alternate replicate connection overrides inherited values from the default connection or the default values.
  - The alternate connection can inherit the values from the default connection only if the same Replication Server controls both alternate and default connections.

For example, to create an alternate replicate connection named `FINANCE_DS2.rdb_conn2` to the `rdb` replicate database in the `FINANCE_DS` data server:

```
create alternate connection to FINANCE_DS.rdb
named FINANCE_DS2.rdb_conn2
go
```

Note: You must define FINANCE_DS and FINANCE_DS2 in the `interfaces` or `sql.ini` file

Altering or Dropping Alternate Replicate Connections

Alter or drop default or alternate replicate connections using the **alter connection** and **drop connection** commands.

The data server and database names that you specify in the commands can be the default or alternate replicate connection names.

You can use configuration parameters available to **alter connection** when you configure an alternate or default replicate connection.

For example, to set **dsi_max_xacts_in_group** to 40 for the `TOKYO_DS.rdb_conn2` alternate replicate connection, enter:

```
alter connection to TOKYO_DS.rdb_conn2
set dsi_max_xacts_in_group to '40'
go
```

Displaying Replicate Connection Information

Use the **replicate** parameter with **admin show_connections** to display information on all replicate connections.

For example, at the Replication Server controlling the replicate databases in the FINANCE_DS and NY_DS data servers, enter:

```
admin show_connections, 'replicate'
```

You see:

```
Connection Name         Server       Database   User
-------------------     ----------   --------   ----------
FINANCE_DS.fin_rdb      FINANCE_DS   fin_rdb    rdb_maint
NY_DS.ny_rdb_conn2      NY_DS        ny_rdb     rdb_maint
```

FINANCE_DS.fin_rdb is the default connection between the Replication Server and the `fin_rdb` database of the FINANCE_DS data server because the connection matches the combination of the data server and database names.

NY_DS.ny_db_conn2 is an alternate connection between the Replication Server and `ny_rdb` database of the NY_DS data server because the connection name does not match the combination of the data server and database names.

Optionally, use the `rs_databases` system table to list both default and alternate connections to the Replication Server.

Creating a Replication System with Multiple Replicate Connections

Create default and alternate replicate connections, and create the corresponding subscriptions to build a multiple replicate connection replication system.

Prerequisites

Ensure that the transactions can be run in parallel, and then divide the replicate transactions into two sets.

Task

This example scenario, which you can use as a model for creating a replication system with multiple replicate connections, consists of the `pdb` primary database on the `PDS` primary data server, containing the `T1` and `T2` tables with corresponding **repdef1** and **repdef2** replication definitons. There is a transaction set affecting each table. The corresponding subscriptions are **sub1** and **sub2**. The `rdb` replicate database is on the `RDS` replicate data server and the primary and replicate Replication Servers are RS1 and RS2.

1. In RS1, use **rs_init** or **create connection** to create the default replicate connection to the replicate database.

   ```
   create connection to RDS.rdb
   using profile ase_to_ase;standard
   set username to rdb_maint
   set password to rdb_maint_ps
   go
   ```

2. In RS1, create an alternate replicate connection named `RDS.rdb1` to the `rdb` replicate database.

   ```
   create alternate connection to RDS.rdb
   named RDS.rdb1
   go
   ```

   Optionally, create another alternate replicate connection to the replicate database from RS2. In RS2, enter:

   ```
   create connection to RDS.rdb
   named RDS.rdb2
   set error class to rs_sqlserver_error_class
   set function string class to rs_sqlserver_function_class
   set username to rdb_maint
   set password to rdb_maint_ps
   go
   ```

3. Create the **sub1** subscription and specify the default replicate connection to replicate transactions in the first transaction set.

```
create subscription sub1 for repdef1
with replicate at RDS.rdb
go
```

4. Create the **sub2** subscription and specify an alternate replicate connection to replicate transactions in the second transaction set.

```
create subscription sub2 for repdef2
with replicate at RDS.rdb2
go
```

Multiple Connections from the Primary Database

Create and manage multiple connections from Replication Server to the primary database that you can associate with RepAgent paths from the primary database to Replication Server.

Creating Alternate Primary Connections

Use **create alternate connection** to create alternate connections from Replication Server to the primary database.

Enter:

```
create alternate connection to dataserver.database
named conn_server.conn_db
[with {log transfer on | primary only}]
```

where:

- *dataserver* and *database* – are the primary data server and database.
- *conn_server.conn_db* – is the alternate primary connection name, which comprises the data server name and a connection name.
  - If *conn_server* is the same as *dataserver*, *conn_db* must be different from *database*.
  - *conn_server.conn_db* must match the name of the connection between the Replication Agent and Replication Server.
  - Each primary connection name must be unique in a replication system.
- **with log transfer on** – instructs Replication Server to create an alternate primary connection and an alternate replicate connection to the database you specify in *dataserver.database*, with both connections having the name you specify in *conn_server.conn_db*
- **primary only** – instructs Replication Server to create only an alternate primary connection to the primary database with the name you specify in *conn_server.conn_db*.

For example, to create an alternate primary connection named `SALES_DS.pdb_conn2` to the `pdb` database in the `SALES_DS` data server, enter:

```
create alternate connection to SALES_DS.pdb
named SALES_DS.pdb_conn2
with primary only
go
```

Altering or Dropping Alternate Primary Connections

Alter or drop default or alternate primary connections using the existing **alter connection** and **drop connection** commands respectively.

For example, you can use **alter connection** to enable or disable a default primary connection to the primary database you specify in *dataserver.database*:

```
alter connection to dataserver.database
set primary only [on|off]
```

Set to off to enable the replicate connection.

Displaying Primary Connection Information

Use the **primary** parameter with **admin show_connections** to display information on all primary connections.

For example, at the Replication Server controlling the primary databases in the SALES_DS data server, enter:

```
admin show_connections, 'primary'
```

You see:

```
Connection Name     Server         Database     User
----------------    ----------     ---------    ----------
SALES_DS.pdb        SALES_DS       pdb          pdb_maint
SALES_DS.pdb_conn2  SALES_DS       pdb          pdb_maint
```

SALES_DS.pdb is the default connection between the Replication Server and the `pdb` database of the SALES_DS data server because the connection name matches the combination of the data server and database names.

SALES_DS.pdb_conn2 is an alternate connection between the Replication Server and the `pdb` database of the SALES_DS data server because the connection name does not match the combination of the data server and database names.

Optionally, use the `rs_databases` system table to list both default and alternate connections to the Replication Server.

Replication Definitions and Subscriptions

Use replication definitions and subscriptions to define replication across multiple alternate connections

Replication Definitions and Subscriptions for Alternate Connections

A replication definition that you create for a primary database applies to all primary connections, default and alternate, between the Replication Server that controls the replication definition and the primary database. Therefore, you must drop all replication definitions for the primary database before you drop the last primary connection to the primary database.

With system version 1570, you can create replication definitions and publications only against a database. The name you specify for the **with primary at** clause of the **create replication definition** command must be the primary database name.

Since all primary connections between a primary database and a Replication Server share all replication definitions, you must specify in the subscription which primary connection is the data source and which replicate connection is the replication target. Specify the corresponding default or alternate connection name in the **with primary** and the **with replicate** clause of **create subscription**. If you do not specify a connection name in the **with primary** clause, Replication Server creates the subscription against the default primary connection to the primary database.

```
create subscription sub_name
for {table_repdef | func_repdef | publication pub |
database replication definition db_repdef}
with primary at data_server.database
with replicate at data_server.database
[where {column_name | @param_name}
       {< | > | >= | <= | = | &} value
[and {column_name | @param_name}
     {< | > | >= | <= | = | &} value]...]
[without holdlock | incrementally | without materialization]
[subscribe to truncate table]
[for new articles]
```

When you upgrade from a Replication Server version that does not support alternate connections, all subscriptions remain defined against the default primary connection and default replicate connection in the upgraded Replication Server.

Example 1 – Subscribe to an Alternate Primary Connection

To create the **sub_conn2** susbcription against the **repdef_conn2** replication definition on the `LON_DS.pdb_conn2` alternate primary connection to the LON_DS primary data server where NY_DS.rdb is the default replicate connection, enter:

```
create subscription sub_conn2 for repdef_conn2
with primary at LON_DS.pdb_conn2
with replicate at NY_DS.rdb
without materialization
go
```

Example 2 – Subscribe to an Alternate Replicate Connection

To create the **sub_conn2** susbcription for the **repdef_conn2** replication definition on the `NY_DS.rdb_conn2` alternate replicate connection, enter:

```
create subscription sub_conn2 for repdef_conn2
with replicate at NY_DS.rdb_conn2
without materialization
go
```

Moving Subscriptions Between Connections

Use **alter subscription** to move a subscription between replicate connections of the same replicate database that use the same Replication Server, without the need to rematerialize.

Execute **alter subscription** at the replicate Replication Server:

```
alter subscription sub_name
for {table_repdef|func_repdef|{{publication pub|
database replication definition db_repdef}
with primary at primary_dataserver_name.primary_database_name}}
move replicate from ds_name.db_name
to ds_name1.db_name1
```

where you are moving the subscription from the *ds_name.db_name* replicate connection to the *ds_name1.db_name1* replicate connection.

For example, to move the **sub1** subscription for the **rep1** replication definition from the `RDS.rdb1` connection to the `RDS.rdb2` connection, enter:

```
alter subscription sub1 for rep1
move replicate from RDS.rdb1
to RDS.rdb2
```

You cannot use **alter subscription** if the primary Replication Server version is earlier than 1570. Instead, you must drop and re-create the subscription at the connection you want.

To move multiple subscriptions that must replicate through the same path, suspend log transfer for the primary connections and then resume log transfer after you move all the subscriptions.

Multiple Primary Replication Paths

Create multiple primary replication paths from the primary database to one or more Replication Servers to increase replication throughput and avoid contention, or to route data to different Replication Servers.

Each primary replication path consists of a RepAgent path from the primary database to a Replication Server and an associated primary connection from the Replication Server to the primary database. You can bind objects, such as tables or stored procedures, to one or more of these paths.

A physical path defines the Replication Server that is going to receive data that you bind to the path, and the RepAgent sender thread that connects to the same Replication Server. Use the same connection name to associate the RepAgent physical path from the primary database to the Replication Server with the corresponding connection from the Replication Server to the primary database.

A logical path groups one or more physical paths under a single name to distribute data to multiple Replication Servers. If you need to replicate a table to multiple destinations, you can bind the table to a logical path that groups the relevant physical paths, instead of binding the table to the physical path for each destination

Creating Multiple Primary Replication Paths

You can create multiple primary replication paths from the primary database to one or more Replication Servers. Each primary path consists of a RepAgent path and an associated primary connection.

1. Enable multithreaded RepAgent and enable RepAgent for multiple replication paths.
2. Create the default primary connection and RepAgent replication path with **rs_init**.
3. Create alternate primary replication paths where each consists of an alternate primary connection linked to an alternate RepAgent replication path.
4. Bind objects that you want to replicate through a specific primary replication path.

Enabling Multithreaded RepAgent and Multiple Paths for RepAgent

Enable multithreaded RepAgent and configure it to use additional paths from the primary database.

By default, the Adaptive Server RepAgent consists of a single thread that scans the primary database log, generates LTL, and sends the LTL to Replication Server. With multithreaded RepAgent, the scanning and sending activities are performed by separate threads.

If you use multithreaded RepAgent, there is always a default path to send data to the Replication Server. RepAgent creates the default path when you initially enable RepAgent on a database.

Multiple replication paths can process only SQL statement replication transactions that are generated by Adaptive Server 15.7 and later.

1. *Set the Memory Available to RepAgent*

   You must provide sufficient memory for the RepAgent thread in Adaptive Server before you enable and configure multiple RepAgent sender threads.
2. *Enable Multithreaded RepAgent*

   Enable or disable a multithreaded RepAgent which uses separate threads for the RepAgent scanner and sender activities.
3. *Set the Number of Send Buffers*

   Set the maximum number of send buffers that the scanner and sender tasks of multithreaded RepAgent can use.
4. *Set the Maximum Number of Replication Paths for RepAgent*

   Set the maximum number of paths that you allow RepAgent to use to replicate data out of the primary database. RepAgent generates one RepAgent sender thread for each RepAgent path.
5. *Display Configuration Parameter Settings*

Use Adaptive Server stored procedures to display the settings of the RepAgent configuration parameters and other information on RepAgent multithreaded and multiple path status.

Setting the Memory Available to RepAgent

You must provide sufficient memory for the RepAgent thread in Adaptive Server before you enable and configure multiple RepAgent sender threads.

The default size for the memory pool dedicated to the RepAgent thread in Adaptive Server is 4096 pages.

1. Display the current RepAgent thread pool size and the settings of other RepAgent thread parameters. At the primary Adaptive Server, enter:

   ```
   sp_configure 'Rep Agent Thread administration'
   go
   ```

   You see:

   ```
   Group: Rep Agent Thread Administration

   Parameter       Default  Memory  Config  Run    Unit      Type
   Name                     Used    Value   Value

   enable rep      0        0       1       1      switch    dynamic
   agent threads

   replication     4096     8194    4096    4096   memory    dynamic
   agent memory                                    pages(2k)
   size
   ```

   This example shows that **enable rep agent threads** is a dynamic parameter that you switch on or off. Changes to dynamic parameters do not require a RepAgent restart.

2. Change the memory that Adaptive Server allocates to the RepAgent thread pool. For example, to set the pool size to 8194 pages, at the primary Adaptive Server enter :

   ```
   sp_configure 'replication agent memory size', 8194
   go
   ```

   You see:

   ```
   Group: Rep Agent Thread Administration

   Parameter       Default  Memory  Config  Run    Unit      Type
   Name                     Used    Value   Value

   replication     4096     16430   8194    8194   memory    dynamic
   agent memory                                    pages(2k)
   size
   (1 row affected)
   Configuration option changed. ASE need not be rebooted since the
   option is dynamic.
   Changing the value of 'replication agent memory size' to '8194'
   increases the amount of memory ASE uses by 8236 K.
   ```

Enabling Multithreaded RepAgent

Enable or disable a multithreaded RepAgent which uses separate threads for the RepAgent scanner and sender activities.

Log in to the primary Adaptive Server and enter:

```
sp_config_rep_agent dbname, 'multithread rep agent', {'true' |
'false'}
```

where *dbname* is the Adaptive Server primary database.

Set to true to enable multithreaded RepAgent. The default is false. You must restart RepAgent for the change to take effect.

Setting the Number of Send Buffers

Set the maximum number of send buffers that the scanner and sender tasks of multithreaded RepAgent can use.

You can set the number of send buffers when you enable multithreaded RepAgent or even after you complete the process of enabling and configuring RepAgent for multi-path replication.

At the primary Adaptive Server, enter:

```
sp_config_rep_agent dbname, 'number of send buffers',
'num_of_send_buffers'
```

where *dbname* is the Adaptive Server primary database.

For example, to set the number of send buffers to 40 for the `pdb1` database, enter:

```
sp_config_rep_agent pdb1, 'number of send buffers', '40'
```

The default for *number of send buffers* is 50 buffers. You can set values between 1 and the value of MAXINT which is 2,147,483,647. The parameter is dynamic; you need not restart RepAgent.

Each send buffer is the same size, which you can set using the **send buffer size** RepAgent parameter. See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures >* ***sp_config_rep_agent***.

Setting the Maximum Number of Replication Paths for RepAgent

Set the maximum number of paths that you allow RepAgent to use to replicate data out of the primary database. RepAgent generates one RepAgent sender thread for each RepAgent path.

At the primary Adaptive Server, enter:

```
sp_config_rep_agent dbname, 'max number replication paths', 'max
number replication paths value'
```

where *dbname* is the Adaptive Server primary database.

For example, to set *max number replication paths* to 3 on the `pdb1` database, enter:

```
sp_config_rep_agent pdb1, 'max number replication paths', '3'
```

If **max number replication paths** is greater than 1, RepAgent continues to use the default path for all replicated objects that you do not specifically bind to a path.

If **max number replication paths** is less than the number of paths with replication objects bound to the paths, RepAgent reports an error and terminates.

Display Configuration Parameter Settings

Use Adaptive Server stored procedures to display the settings of the RepAgent configuration parameters and other information on RepAgent multithreaded and multiple path status.

- **sp_config_rep_agent** – specify only the database name to display the settings of parameters you set with **sp_config_rep_agent**
- **sp_help_rep_agent** – to display additional information on the RepAgent status, specify:
  - **send** – displays the number of send buffers that you have allocated to RepAgent.
  - **config** – displays information on the RepAgent multiple path configuration parameters.
  - **process** – displays information about the multiple Rep Agent processes when you enable **multithread rep agent**.
- **sp_who** – displays information on RepAgent processes and threads running in Adaptive Server

See *Replication Server Administration Guide Reference Manual > Adaptive Server Commands and System Procedures* for **sp_config_rep_agent** and **sp_help_rep_agent**.

See *Adaptive Server Reference Manual: Procedures > System Procedures > **sp_who***.

Creating Alternate Replication Paths for the Primary Database

Use the **add** parameter with **sp_replication_path** to create alternate physical paths between the primary database and a Replication Server by associating a RepAgent replication path with a primary connection from the Replication Server.

Prerequisites

Create the default replication path between the primary database and Replication Server with **rs_init**.

Task

Using the example replication system consisting of the PDS primary data server, `pdb` database, and RS1 and RS2 Replication Servers, create two alternate replication paths on the primary database to make a total of three primary replication paths including the default primary replication path.

1. Create an alternate primary replication path between `pdb` and RS2 named pdb_1:
   a) Create an alternate physical replication path named pdb_1 between `pdb` and RS2. At PDS, enter:

```
sp_replication_path "pdb", 'add', "pdb_1", "RS2", "RS2_user",
"RS2_password"
```

b) Create the corresponding alternate primary connection named pdb_1 from RS2 to `pdb`
At RS2, enter:

```
create alternate connection to PDS.pdb
named PDS.pdb_1
set error class to rs_sqlserver_error_class
set function string class to rs_sqlserver_function_class
set username to pdb1_maint
set password to pdb1_maint_ps
with primary only
```

2. Create another primary replication path between `pdb` and RS1 named pdb_2:

a) Create an alternate physical replication path named pdb_2 between `pdb` and RS1.
At PDS, enter:

```
sp_replication_path "pdb", 'add', "pdb_2", "RS1", "RS1_user",
"RS1_password"
```

b) Create the corresponding alternate primary connection named pdb_2 from RS1 to `pdb`
At RS1, enter:

```
create alternate connection to PDS.pdb
named PDS.pdb_2
with primary only
```

Dropping a Replication Server Definition

Use the **drop** parameter with **sp_replication_path** to remove a Replication Server as a destination from a physical replication path that is not the default primary replication path.

You cannot drop a default primary replication path and you cannot drop any primary replication path if there are objects bound to the path.
To drop RS1 as a destination, at PDS enter:

```
sp_replication_path 'pdb', 'drop', "RS1"
```

Creating Logical Primary Replication Paths

Use the **add** and **logical** parameters with **sp_replication_path** to create logical primary replication paths that you can use to distribute data and objects bound to a physical path to multiple Replication Servers.

Prerequisites

Create the relevant physical primary replication paths to support the logical primary replication paths.

Task

If you bind a replication object such as the `dt1` dimension table to pdb_1, `dt1` always travels through pdb_1 to RS2. Using the example replication system and the three physical primary replication paths—default, pdb_1, and pdb_2, you can create a logical replication path named logical_1 to distribute `dt1` through pdb_2 to RS2.

Note: You cannot add the default path to a logical path.

1. Create the logical_1 logical path and add pdb_1 as a physical primary replication path:
   At PDS, enter:
   ```
   sp_replication_path 'pdb', 'add', 'logical', 'logical_1', 'pdb_1'
   ```
   logical_1 sends data through pdb_1 to RS1 only.
2. Add pdb_2 as a physical primary replication path for logical_1:
   At PDS, enter:
   ```
   sp_replication_path 'pdb, 'add', 'logical', 'logical_1', 'pdb_2'
   ```
   logical_1 sends data through pdb_1 to RS1 and pdb_2 to RS2.

Dropping Elements in a Logical Primary Replication Path

Use the **drop** and **logical** parameters with **sp_replication_path** to remove elements from a logical replication path.

In this example, the logical_1 logical path contains the pdb_1 and pdb_2 physical paths, which are called elements of logical_1. You can remove an element from the logical path.

Warning! If you remove a path from, or add a path to an existing logical path, the set of destinations can change and replicated objects may not get to the destinations that the replicated objects did go to before the change.

1. Remove pdb_1 from logical_1:
   ```
   sp_replication_path 'pdb', 'drop', 'logical', 'logical_1',
   'pdb_1'
   ```
   The logical_1 logical path now contains only the pdb_2 physical path. Any objects bound to logical_1 replicate only through pdb_2.
2. Remove pdb_2 from logical_1:
   ```
   sp_replication_path 'pdb', 'drop', 'logical', 'logical_1',
   'pdb_2'
   ```
   If you remove the last element, and if there are no objects bound to the logical path, Replication Server removes the last element and the entire logical path together because a logical path cannot exist without its elements.

Removing a Logical Path

Use the **drop** and **logical** parameters with **sp_replication_path** to remove an entire logical replication path.

To remove the entire logical path, do not specify a Replication Server or element in the command. If you remove the last element in a logical path, Replication Server removes the entire logical path.

Note: You cannot remove a logical path or a physical path if objects are still bound to it.

To remove the logical_1 logical path, enter:

```
sp_replication_path 'pdb', 'drop', 'logical', 'logical_1'
```

Binding Objects to a Replication Path

Use the **bind** parameter with **sp_replication_path** to associate an object with a physical or logical primary replication path. The bound object always follows the same path during replication.

You can bind objects, such as tables or stored procedures, to one or more primary replication paths. When you bind an object to a path, RepAgent sends any replicable actions that you perform upon that object through the path to the Replication Servers that you define in your multiple replication path configuration. If you do not bind an object to a path, RepAgent uses the default path to send the object to the Replication Server defined in the default path. You cannot bind an object to the default path and you do not need to do anything to send an object through the default path.

To bind an object to a primary replication path, enter:

```
sp_replication_path dbname, 'bind', "object_type",
"[table_owner].object_name", "path_name"
```

where:

- *object_type* – **table** or **sproc** (stored procedure).
- *[table_owner.]object_name* – the table or stored procedure name.

  Note: If you do not specify a table owner if the object is a table, the binding applies only to tables owned by `dbo`, the database owner.

- *path_name* – a physical or logical path name.

For example, to bind the:

- `t1` table to the pdb_2 replication path:

  ```
  sp_replication_path pdb, 'bind', "table", "t1", "pdb_2"
  ```

- `t2` table belonging to `owner1` to the pdb_2 replication path:

  ```
  sp_replication_path pdb, 'bind', "table", "owner1.t2", "pdb_2"
  ```

- **sproc1** stored procedure to the pdb_2 replication path:

  ```
  sp_replication_path pdb, 'bind', "sproc", "sproc1", "pdb_2"
  ```

- `dt1` dimension table object to the everywhere logical path:

  ```
  sp_replication_path pdb, 'bind', "table", "dt1", "everywhere"
  ```

Optionally, use the asterisk "*" or percent "%" wildcard characters, or a combination of both in *object_name* to specify a range of names or matching characters that you want to bind to a path. For example, to bind tables with names that match various wildcard character combinations to the pdb_2 replication path:

- `sp_replication_path pdb, 'bind', 'table', 'a*', "pdb_2"`
- `sp_replication_path pdb, 'bind', 'table', 'au%rs', "pdb_2"`
- `sp_replication_path pdb, 'bind', 'table', 'a*th%s', "pdb_2"`
- `sp_replication_path pdb, 'bind', 'table', 'authors%', "pdb_2"`

Unbinding Objects from a Replication Path

Use the **unbind** parameter with **sp_replication_path** to remove the association between a bound object and a physical or logical replication path.

To remove the binding between an object and one or more primary replication paths, enter:

```
sp_replication_path dbname, 'unbind', "object_type", "object_name",
{"path_name"|all}
```

where:

- *object_type* – specifies the type of object that can be either **path**, **table**, or **sproc** (stored procedure).
- *[table_owner.]object_name* – name of the table, stored procedure, or path that you want to unbind.

  Note: If you do not specify a table owner if the object is a table, the binding applies only to tables owned by `dbo`, the database owner.
- *path_name* | **all** – specifies a physical or logical path name, or all paths. If you specify **path** as the *object_type*, provide the path name as *object_name*, and specify the **all** option, Replication Agent unbinds all objects from the path name you specified.

For example:

- To remove the binding `t1` table from the pdb_2 replication path:

  ```
  sp_replication_path pdb, 'unbind', "table", "t1", "pdb_2"
  ```
- To remove all bindings on the `t1` table:

  ```
  sp_replication_path pdb, 'unbind', "table", "t1", "all"
  ```
- To remove the binding of all objects to the pdb_2 replication path:

  ```
  sp_replication_path pdb, 'unbind', 'path', 'pdb_2', "all"
  ```

Object Binding and Replication of SQL and DDL Statements

You can send SQL and DDL statements over the default path or over all paths for any object that is not bound to a path.

With SQL statement replication and DDL replication, when you bind an object such as a table to a specific replication path, any SQL or DDL statement that includes the object uses the specified replication path. The SQL or DDL statement uses either the default replication path or all replication paths if the statement includes any object that you have not bound to a path.

Use **ddl path for unbound objects** with **sp_config_rep_agent** to send SQL or DDL statements for unbound objects over all paths or the default path:

```
sp_config_rep_agent dbname, 'ddl path for unbound objects', {'all' |
'default'}
```

The default setting is **all**.

Object Binding and Database Resynchronization

Multi-path replication sends the **resync**, **resync purge**, and **resync init** database resynchronization markers through all available replication paths.

See *Replication Server Administration Guide Volume 2 > Replication System Recovery > Replicate Database Resynchronization for Adaptive Server > Configuring Database Resynchronization > Send the Resync Database Marker to Replication Server*.

Object Binding and rs_ticket

Multi-path replication sends the result of executing **rs_ticket** through all available replication paths. You must filter the data to obtain the data relevant to you.

See *Replication Server Reference Manual > RSSD Stored Procedures > **rs_ticket***.

Changing Configuration Values in a Replication Path

Use the **config** parameter with **sp_replication_path** to set parameter values in alternate replication paths.

You can change only the password and user ID for alternate replication paths.

To change the value of a parameter for an alternate path, enter:

```
sp_replication_path dbname, 'config', "path_name",
 "config_parameter_name", "config_value"
```

where *config_parameter_name* is **rs_username** or **rs_password**.

For example, to change the:

- User name that `pdb_1` uses to connect to RS1 to "RS1_user", enter at PDS:

  ```
  sp_replication_path pdb, 'config', "pdb_1", "rs_username",
  “RS1_user”
  ```

- Password that `pdb_1` uses to connect to RS1 to "january", enter at PDS:

  ```
  sp_replication_path pdb, 'config', "pdb_1", "rs password",
  “january”
  ```

Use **sp_config_rep_agent** to configure parameters for the default replication path. See *Replication Server Reference Manual > Adaptive Server Commands and System Procedures > **sp_config_rep_agent***.

Display Replication Path Information

Use the **list** parameter with **sp_replication_path** at the primary database to display information on bindings and replication objects.

```
sp_replication_path dbname, 'list', ['object_type'], ['object_name']
```

- *object_type* – specify the type of object: **path**, **table** , **sproc** (stored procedure).
- *object_name* – display the binding relationships for a particular object. You must specify *object_type* when you want to specify the name of an object.

Example 1

To display the path relationships of all bound objects, do not specify *object_type* or *object_name*:

```
sp_replication_path 'pdb','list'
go
```

You see:

```
Binding              Type     Path
-------------------- -------- --------------------
dbo.dt1              T        everywhere
dbo.sproc1           P        pdb_1
dbo.sproc1           P        pdb_2
dbo.t1               T        pdb_2
dbo.t2               T        pdb_1

(5 rows affected)

Logical Path                 Physical Path
---------------------------- --------------------
everywhere                   pdb_1
everywhere                   pdb_2

(2 rows affected)
Physical Path                  Destination
------------------------------ -------------------
pdb_1                          RS2
pdb_2                          RS1

(2 rows affected)
(return status = 0)
```

Example 2

To display information on all bound tables:

```
sp_replication_path 'pdb','list','table'
go
```

You see:

```
Binding              Type     Path
-------------------- -------- --------------------
dbo.dt1              T        everywhere
dbo.t1               T        pdb_2
dbo.t2               T        pdb_1

(3 rows affected)
(return status = 0)
```

Example 3

To display information on all stored procedures:

```
sp_replication_path 'pdb','list','sproc'
go
```

You see:

```
Binding              Type     Path
-------------------- -------- --------------------
dbo.sproc1           P        pdb_2
dbo.sproc1           P        pdb_1
dbo.sproc2           P        pdb_1

(3 rows affected)
(return status = 0)
```

Example 4

To display information on only the **sproc1** stored procedure:

```
sp_replication_path 'pdb','list','sproc','sproc1'
go
```

You see:

```
Binding              Type     Path
-------------------- -------- --------------------
dbo.sproc1           P        pdb_2
dbo.sproc1           P        pdb_1

(2 rows affected)
(return status = 0)
```

Example 5

To display information on all replication paths:

```
sp_replication_path 'pdb','list','path'
go
```

You see:

```
Path                 Type     Binding
-------------------- -------- ----------------------
everywhere           T        dbo.dt1
pdb_1                P        dbo.sproc1
pdb_1                T        dbo.t2
pdb_2                P        dbo.sproc1
pdb_2                T        dbo.t1

(5 rows affected)
Logical Path                  Physical Path
----------------------------- -------------------------
everywhere                    pdb_1
everywhere                    pdb_2

(2 rows affected)
Physical Path                 Destination
----------------------------- ----------------------
pdb_1                         RS2
pdb_2                         RS1

(2 rows affected)
(return status = 0)
```

Example 6

To display information only on the "everywhere" logical replication path:

```
sp_replication_path 'pdb','list','path','everywhere'
go
```

You see:

```
Path                 Type     Binding
-------------------- -------- -------------------------
everywhere           T        dbo.dt1

(1 rows affected)
Logical Path                  Physical Path
----------------------------- -------------------------
everywhere                    pdb_1
everywhere                    pdb_2

(2 rows affected)
Physical Path                 Destination
----------------------------- -------------------------
pdb_1                         RS2
pdb_2                         RS1

(2 rows affected)
(return status = 0)
```

Note: You also see the physical paths underlying the logical path.

Example 7

To display information only on the pdb_1 physical path:

```
sp_replication_path 'pdb','list','path','pdb_1'
go
```

You see:

```
Path                 Type     Binding
-------------------- -------- --------------------
pdb_1                P        dbo.sproc1
pdb_1                T        dbo.t2

(2 rows affected)
Physical Path                  Destination
------------------------------ -------------------
pdb_1                          RS2

(1 rows affected)
(return status = 0)
```

Creating Multiple Replication Paths for MSA Environments

Use replication definitions and subscriptions to bind a replicate connection for a replicate database and a primary connection for a primary database to create two complete replication paths in an MSA environment.

1. Divide transactions into two sets and ensure the transactions can be run in parallel.

   For example, you can divide the transactions into two sets of objects such as tables or stored procedures.
2. Create a default primary connection to the primary database and create a default replicate connection to the replicate database.
3. Create an alternate primary connection to the primary database and create an alternate replicate connection to the replicate database.
4. Enable multithreaded RepAgent and two replication paths for RepAgent, and bind the objects to the replication paths.
5. Create a replication definition against the primary database.

   If the default primary connection and the alternate primary connection are on different Replication Servers, create replication definitions on each Replication Server.
6. Create a subscription against the default primary connection and the default replicate connection.
7. Create a subscription against the alternate primary connection and the alternate replicate connection.

Multiple Replication Paths for Warm Standby Environments

You can improve replication performance in warm standby environments with alternate connections and alternate logical connections, or by building end-to-end replication paths.

Creating Alternate Logical Connections in a Warm Standby Environment

Use **create alternate logical connection** to create alternate logical connections for an existing default logical connection in a warm standby environment.

You can use different Replication Servers to control the default logical connection and the alternate logical connection. Both the active and standby databases must support having multiple Replication Agents. When you switch the active and standby databases, execute the **switch active command** for every logical connection—alternate and default. The warm standy process is complete when you switch all the paths.

At the Replication Server that manages the warm standby pair, enter:

```
create alternate logical connection to LDS.ldb
named conn_lds.conn_ldb
```

where:

- *LDS* and *ldb* – the logical data server and database names
- *conn_lds.conn_ldb* – the data server and database connection components of the alternate logical connection name.

Creating Alternate Connections in a Warm Standby Environment

Create an alternate primary connection to the active database or an alternate replicate connection to the standby database for an alternate logical connection.

At the Replication Server that manages the warm standby pair, enter:

```
create alternate connection to ds_name.db_name
named conn_server.conn_db
...
[as {active|standby} for conn_lds.conn_ldb]
```

where:

- *ds_name.db_name* – the data server name and either the active or standby database name.
- *conn_server.conn_db* – the alternate active or standby connection, which comprise a data server name a connection name. *conn_ds* must be the same as *ds_name* to support incoming Replication Agent connections.
- *conn_lds.conn_ldb* – the data server and database connection components of the alternate logical connection name.
- **active | standby** – specify whether to create an alternate connection to the active or standby database.

Creating Multiple Replication Paths for Warm Standby Environments

Use logical connections to bind a replicate connection for a standby database and a primary connection for an active database to create two complete replication paths between an active database and a standby database in a warm standby environment.

1. Divide transactions into two sets and ensure the transactions in the two sets can be run in parallel.

   For example, you can divide the transactions into two sets of objects such as tables or stored procedures.
2. Enable multithreaded RepAgent and two replication paths for RepAgent for both the active database and replicate database, and bind the objects to the replication paths.
3. Create a logical connection. See **create logical connection** in the *Replication Server Reference Manual*.
4. Use **rs_init** to add the active and the standby databases to the replication system.
5. Create an alternate logical connection
6. Create an alternate active connection for the alternate logical connection.
7. Use **admin who** to check for the REP AGENT thread and verify that the default and alternate connections to the active database of the warm standby pair are active. For example, you see:

   ```
   31 REP AGEN   Awaiting Command     TOKYO_DS.pubs2
   ```
8. Create an alternate standby connection for the alternate logical connection.

Switching Active and Standby Databases

You must switch all the replication paths in a warm standby environment with multiple replication paths when you switch from the active to the standby database.

The switching procedure is the same for alternate and default replication paths.

1. Switch all the alternate replication paths.
2. Switch the default replication path.

See also

- *Switch the Active and Standby ASE Databases* on page 84

Dedicated Routes

A dedicated route distributes only transactions for a specific primary connection. You can create a dedicated route to the replicate Replication Server to replicate high priority transactions or to maintain a less congested path for a specific primary connection.

A shared route is between a primary Replication Server and a replicate Replication Server that distributes transactions for all the primary connections originating from the primary Replication Server. You do not bind shared routes to a specific connection. Connections that you do not bind to a dedicated route use any available valid shared route.

You can create a dedicated route only if:

- A shared route exists from the primary Replication Server to the destination Replication Server and the shared route is a direct route. You cannot create a dedicated route if there is only an indirect route between the Replication Servers.
- The shared route is valid and not suspended.
- The route version of the shared route is 1570 or later.

Creating Dedicated Routes

Use **create route** and the **with primary at** clause to create a dedicated route.

For example, to create a dedicated route between the RS_NY primary Replication Server and the RS_LON replicate Replication Server for the `NY_DS.pdb1` primary connection, at RS_NY enter:

```
create route to RS_LON
 with primary at NY_DS.pdb1
 go
```

After you create a dedicated route for a specific connection, all transactions from the connection to the destination Replication Server follow the dedicated route.

Commands to Manage Dedicated Routes

Use **create route**, **drop route**, **resume route**, and **suspend route** to manage and monitor dedicated routes.

Include the **with primary at** *dataserver.database* clause in the command to specify a dedicated route, where *dataserver.database* is the primary connection name.

See *Replication Server Reference Manual > Replication Server Commands*.

Command	Syntax	Command and Parameter Changes
create route	create route to *dest_replication_server* { **with primary at *dataserver.database*** | set next site [to] *thru_replication_server* | [set username [to] *user*] [set password [to] *passwd*] [set *route_param* to '*value*' [set *route_param* to '*value*']...] [set *security_param* to '*value*' [set *security_param* to '*value*']...]}	

Command	Syntax	Command and Parameter Changes
drop route	`drop route to dest_replication_server [with primary at dataserver.database] [with nowait]`	You must drop the dedicated route before you drop a shared route. After you drop a dedicated route, transactions from the specified primary connection to the destination Replication Server go through the shared route. **Warning!** Use the **with nowait** clause only as a last resort. The clause forces Replication Server to drop a route even if the route contains transactions in the outbound queue of the route. As a result, Replication Server may discard some transactions from the primary connections. The clause instructs Replication Server to drop the dedicated route even if the route cannot communicate with the destination Replication Server. If you use the clause, use **sysadmin purge_route_at_replicate** at the former destination site to remove subscriptions and route information from the system tables at the destination. See *Replication Server Administration Guide Volume 1 > Manage Routes > Drop Routes > **drop route** comand*.
suspend route	`suspend route to dest_replication_server [with primary at dataserver.database]`	
resume route	`resume route to dest_replication_server [with primary at dataserver.database] [skip transaction with large message]`	

Display Dedicated Route Information

Use **admin who** to display information on dedicated routes between Replication Servers.

In this example, there is a dedicated route from the RS_NY primary Replication Server to the RS_LON replicate Replication Server for the `NY_DS.pdb1` primary connection. Enter **admin who** at the two Replication Servers and you see:

- At `RS_LON`:

```
Spid Name      State              Info
45 SQT         Awaiting Wakeup    103:1  DIST NY_DS.pdb1
13 SQM         Awaiting Message   103:1 NY_DS.pdb1
32 REP AGENT   Awaiting Command   NY_DS.pdb1
16 RSI         Awaiting Wakeup    RS_LON
11 SQM         Awaiting Message   16777318:0 RS_LON
55 RSI         Awaiting Wakeup    RS_LON(103) /* Dedicated RSI
thread */
53 SQM         Awaiting Message   16777318:103 RS_LON(103) /
*Dedicated RSI outbound queue */
```

- At `RS_NY`:

```
Spid Name     State               Info
37 RSI USER   Awaiting Command    RS_NY(103) /*Dedicated RSI user
*/
32 RSI USER   Awaiting Command    RS_NY
```

See *Replication Server Reference Manual > Replication Server Commands >* ***admin who***.

Adaptive Server Monitoring Tables for Multiple Replication Paths

Use Adaptive Server monitoring tables to provide a statistical snapshot of the state of Adaptive Server during replication using multiple paths involving RepAgent for Adaptive Server primary databases. The tables allow you to analyze Adaptive Server performance.

Table	Description
`monRepLogActivity`	Provides information from monitor counters updated by Replication Agent
`monRepScanners`	Provides statistics for the RepAgent Scanner task
`monRepScannersTotalTime`	Provides information on where the RepAgent Scanner task is spending its time
`monRepSenders`	Provides processing information on RepAgent Sender tasks

See *Adaptive Server Enterprise > Performance and Tuning Series: Monitoring Tables > Introduction to Monitoring Tables > Monitoring Tables in Adaptive Server*.

System Table Support for Alternate Primary and Replicate Connections

Replication Server creates a row for each primary and replicate connection in the `rs_databases` table and has a column—`conn_id` to uniquely identify the primary or replicate connection in a particular row.

Replication Server uses the `dsname` and `dbname` columns to identify an alternate connection by the connection name and identifies the default primary or replicate connection by the data server and database names. `dbid` identifies the ID of the database that the connection connects to. If the row is for a default connection, `connid` is equal to the `dbid`. If the row is for an alternate connection, `connid` is not equal to the `dbid`. See *Replication Server Reference Manual > Replication Server System Tables*.

Multiprocessor Platforms

You can run Replication Server on symmetric multiprocessor (SMP) or single-processor platforms because the Replication Server multithreaded architecture supports both hardware configurations. On a multiprocessor platform, Replication Server threads can run in parallel, thereby improving performance and efficiency.

On a single processor platform, Replication Server threads run serially.

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.

Enable Multiprocessor Support

Use **configure replication server** with the **smp_enable** option to specify whether Replication Server takes advantage of a multiprocessor machine.

Enter:

```
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 on.

smp_enable is a static option. You must restart Replication Server after changing the status of **smp_enable**.

Commands to Monitor Thread Status

You can verify Replication Server thread status using **admin who** commands or the **sp_help_rep_agent** stored procedure.

- **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 also

- *Verify and Monitor Replication Server* on page 5

Monitor Performance

Replication Server provides monitors and counters to monitor performance.

See also

- *Monitor Performance Using Counters* on page 273

Allocation of 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.

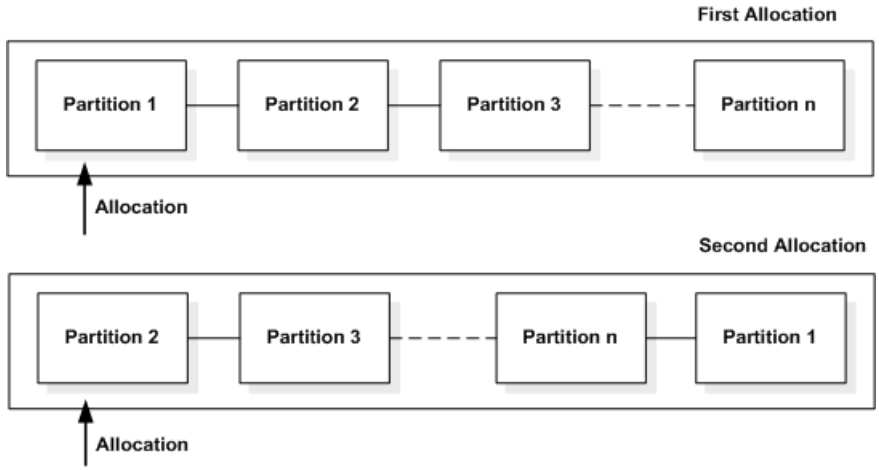
A Replication Server can have any number of partitions of varying sizes. The sum of the partition sizes is the Replication Server capacity for queued transactions.

Default Allocation Mechanism

By default, Replication Server assigns queue segments to the first partition in an ordered list of partitions.

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 22: Default Allocation Mechanism



Choose 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 allocation mechanism.

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.

Allocating Disk Partitions to Stable Queues

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.

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
   ```

   and

   ```
   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
   ```

   and

   ```
   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'
   ```

   and

   ```
   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
   ```

and

```
resume connection to SEATTLE_DS
```

Drop 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.

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 **get heartbeat** and **get heartbeat tickets** in *Replication Server Reference Manual > Replication Monitoring Services API* .

Monitor Performance Using Counters

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.

To monitor performance using the RepAgent counters, see *Replication Server Administration Guide Volume 1 > Manage RepAgent and Support Adaptive Server > Use Counters to Monitor RepAgent Performance*.

Commands to View Counter Values

You can view current counter values and other performance information at any time using several commands.

You can use:

- **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

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)

Replication Server Modules

Replication Server has several 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 23. 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

Module name	Acronym	Independent/dependent
SYNC Element	SYNCELE	Dependent of SYNC

Counters

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.

To view descriptive and status information about Replication Server counters, use the **rs_helpcounter** stored procedure.

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.

See also

- *View Information About the Counters* on page 284

Data 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.

Collect Statistics for a Specific Time Period

Use **admin stats** to collect 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 > Replication Server Commands* for complete syntax and usage information.

Specify the Counters to Be Sampled

You can specify all counters or as few as a single instance of a counter.

Use these parameters with **admin stats** to specify the counters:

- **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.

- **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
```

See also

- *Modules* on page 273

Specify the Sample Period

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
```

Specify How Statistics Are to Be Reported

You can send statistics to the computer screen or to the RSSD.

Display 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.

Save 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 with *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.

Table 24. Sampling Periods and Observation Intervals

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. You can use the **rs_dump_stats** procedure to dump information from these tables.

See also

- *Use the rs_dump_stats Procedure* on page 282

Collect 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 all statistics:

   Enter:

   ```
   admin statistics, reset
   ```
2. Turn on sampling:

   Enter:

   ```
   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.

Enter:

```
configure replication server
set stats sampling to "off"
```

2. Dump statistics to the screen.

   Enter:

   ```
   admin statistics, "all"
   ```

3. Clear all statistics:

   Enter:

   ```
   admin statistics, reset
   ```

4. Turn on sampling:

   Enter:

   ```
   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.

   Enter:

   ```
   configure replication server
   set stats sampling to "off"
   ```

2. Dump statistics to the screen.

   Enter:

   ```
   admin statistics, "all"
   ```

3. Clear all statistics:

   Enter:

   ```
   admin statistics, reset
   ```

View Statistics on Screen

Use **admin stats**to display 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 can choose whether to display statistics on the screen when you configure the sample run using **admin stats**.

See the *Replication Server Reference Manual* > *Replication Server Commands* > ***admin stats*** for example output and complete syntax and usage information.

See also

- *Collect Statistics for a Specific Time Period* on page 276

View 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
```

View 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
  ```

View the Number of Transactions in the Stable Queues

Use **admin stats** with the **backlog** option to 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
```

View Statistics Saved in the RSSD

There are several commands and procedures you can use to view statistics 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 *Replication Server Reference Manual > Replication Server System Tables* 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`

Use 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.

See *Replication Server Reference Manual > RSSD Stored Procedures >* ***rs_dump_stats*** for complete syntax and usage information.

To use **rs_dump_stats**, log in to the RSSD and execute the stored procedure. For example:

```
1> rs_dump_stats
2> go
```

Sample Output from rs_dump_stats

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 observation
intervals*
1                                                   *No of min between
                                                     observations*
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*
```

View Information About the Counters

You can view descriptive information about the counters stored in the `rs_statcounters` table using the **rs_helpcounter** system procedure.

- To view a list of modules that have counters and to view 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 }
  ```

See *Replication Server Reference Manual > RSSD Stored Procedures >* ***rs_helpcounter*** for detailed syntax and usage information.

Resetting of Counters

Use the **admin stats, reset** command to reset all counters, except those that are never reset, to 0 (zero).

Enter:

```
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.

Generate Performance Reports

Use the **rs_stat_populate** and **rs_stat_genreport** stored procedures to generate 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** 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 other information.

Errors and Exceptions Handling

Learn the various error handing methods for Replication Server.

See 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. You can use Sybase Central to monitor the replication system status, 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 automaticallyresume their work. Lengthier failures may require intervention if there is not enough disk space to queue messages, or if you must 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 with replication system 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:

- 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.

See also

- *Replication System Recovery* on page 309

Error Log Files

Learn about error log files in the replication system that you can access to 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. See the *Replication Server Troubleshooting Guide*.

Replication Server allows user-definable error processing in response to data server errors.

See also

- *Data Server Error Handling* on page 291

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.

Message Types in the Replication Server Error Log

There are several message types in the Replication Server error log. Each log message begins with a letter to indicate the message type.

Table 25. Message Types in the Replication Server Error Log

Error code	Description
I	An informational message.
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

Informational messages in the Replication Server error log.

The format of informational messages in the error log is:

```
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

Errors and warning messages in the Replication Server error log.

The format of messages other than informational messages is:

```
severity, date. ERROR #error_number thread_name(context) -
source_file(line) message
```

If the message is a warning, “ERROR” in the format becomes “WARNING.”

The parameters are:

- *severity* – W, E, H, F, or N, corresponding to the message types in the Replication Server error log.
- *date* – date and time that the error occurred.
- *error_number* – Replication Server error number.
- *thread_name* – name of the Replication Server thread that received the error. See *Replication Server Administration Guide Volume 1 > Replication Server Technical Overview* for details about Replication Server threads.
- *context* – provides some information about the thread’s context at the time the error occurred.
- *source_file* – program file in the Replication Server source code where the error was reported.
- *line* – line number in the program file in the Replication Server source code where the error was reported.
- *RS_language* – specifies the language for the Replication Server message.
- *message* – full text of a message from a Replication Server, 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 places question marks (?) in messages when 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.

This is a 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. You can assign other actions for data server errors.

Find 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 this UNIX example shows:

```
Log File Name
 -----------------------------
 /work/sybase/SYDNEY_RS/SYDNEY_RS.log
```

Change to a New Replication Server Log File

Use the **admin set_log_name** command to begin a new error log file.

This command closes the current log file and opens a new one. Subsequent messages are written in the new log file.

For example in UNIX, enter:

```
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.

This is 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. See *Adaptive Server Enterprise Troubleshooting: Error Messages Advanced Resolutions*.

Sample RepAgent Error Messages

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 a possible network problem to resolve.

Data Server Error Handling

Replication Server allows user-definable error processing for data server errors. Assign appropriate error class to a specified connection and customize the assigned error class. The error actions should match the errors returned by the data server.

RCL Commands and System Procedures for Error Processing

There are several RCL commands and Adaptive Server system procedures that manage errors and error classes.

Table 26. RCL Commands and System Procedures for Error Processing

Command	Description
assign action	Specifies an error processing action for one or more data server or Replication Server errors

Command	Description
alter error class	Changes an existing error class
create error class	Creates a new error class
drop error class	Drops an existing error class
alter connection	Associates an error class with an existing database connection
create connection	Associates an error class with a new database connection
rs_helpclass	Adaptive Server stored procedure that displays the name of each existing error class, function-string class, and their primary Replication Server, and in the case of inherited classes, the parent class.
rs_helperror	Adaptive Server stored procedure that displays the Replication Server error actions mapped to a given data server or Replication Server error number

Default Error Classes

Replication Server provides **rs_sqlserver_error_class** as the default Adaptive Server error class, **rs_repserver_error_class** as the default Replication Server error class, and default error classes for non-ASE databases. You cannot modify these default error classes.

Table 27. Non-ASE Error Classes

Database	Class name
IBM DB2	**rs_db2_error_class**
IBM UDB	**rs_udb_error_class**
Microsoft SQL Server	**rs_msss_error_class**
Oracle	**rs_oracle_error_class**
Sybase IQ	**rs_iq_error_class**

See also

- *Designate Primary Site for an Error Class* on page 293

Native Error Codes for Non-ASE Databases

When Replication Server establishes a connection to a non-ASE replicate server, Replication Server verifies whether the option to return native error codes from the non-ASE replicate server is enabled for the connection.

If the option is not enabled, Replication Server logs a warning message that the connection works but that error action mapping may be incorrect.

See *Replication Server Options > Enterprise Connect Data Access Option for ODBC Users Guide for Access Services > Configuring the Access Service Library > Configuration*

Property Categories > Target Interaction Properties > ***ReturnNativeError*** to set the option in the Enterprise Connect™ Data Access (ECDA) Option for ODBC for your replicate server.

Create an Error Class

Use **create error class** to create your own error classes.

An error class is a name used to group error action assignments. **create error class** copies the error actions from the template error class to the new 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.

Note: When you create a connection using a connection profile, the error class is assigned by the connection profile. The connection profile predefines the error class for a specific data server. See *Replication Server Administration Guide Volume 1 > Manage Database Connections > Prepare Databases for Replication > Prepare Non-ASE Servers for Replication > Connection Profiles*.

To create an error class, enter:

```
create [replication server] error class error_class
[set template to template_error_class]
```

The **replication server** option specifies that you want to create a Replication Server error class. You can use the **set template to** option, and another error class as a template to create an error class.

Examples

This example creates an error class named **pubs2_error_class** without a template error class:

```
create error class pubs2_error_class
```

This example creates the **my_rs_err_class** Replication Server error class based on **rs_repserver_error_class**, which is the default Replication Server error class:

```
create replication server error class my_rs_err_class
set template to rs_repserver_error_class
```

This example creates the **my_error_class** error class for an Oracle database based on **rs_oracle_error_class** as a template:

```
create error class my_error_class
set template to rs_oracle_error_class
```

Designate Primary Site for an Error Class

You must specify a primary site before you can modify a default error class.

Initially, `rs_sqlserver_error_class` and the other default non-ASE error classes, do not have a primary site. Since you can only create server-wide error classes at a primary site for

a class, use **create error class** to designate one of the Replication Servers as a primary site for an error class.

For Adaptive Server for example, execute **create error class rs_sqlserver_error_class** at the primary site. Verify that all other Replication Servers have direct or indirect routes from the primary site.

See also

- *Change the Primary Replication Server for an Error Class* on page 295

Assign Error Actions

You can assign different error actions for errors returned by a data server.

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 also

- *Assign Actions to Data Server Errors* on page 296

Alter Error Classes

alter error class copies error actions from a template error class to the error class you want to alter and overwrites error actions which have the same error code.

To alter an error class, enter:

```
alter [replication server] error class error_class
set template to template_error_class
```

The **replication server** option specifies that you want to alter a Replication Server error class.

For example, to alter `my_error_class` for an Oracle database based on `rs_sqlserver_error_class` as a template:

```
alter error class my_error_class
set template to rs_sqlserver_error_class
```

Initialize a New Error Class

After you have create 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 initialize 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.

Drop 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 [replication server] error class error_class
```

For example, to drop the `pubs2_error_class` error class, enter:

```
drop error class pubs2_error_class
```

You cannot drop the `rs_sqlserver_error_class` or any of the default non-ASE error classes.

See also

- *Default Error Classes* on page 292

Change 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 [replication server] 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.

The parameters are:

- **replication server** – option that specifies that you want to change the primary Replication Server for a Replication Server error class. Leave this if you want to modify a data server 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.

Display Error Class Information

Use the **rs_helpclass** stored procedure to display the names of existing error classes and function-string classes in their primary Replication Servers and in the replication system.

For example:

```
rs_helpclass error_class
```

```
Error Class(es)             PRS for class
 --------------              ---------
rs_sqlserver_error_class    Not Yet Defined
```

See *Replication Server Reference Manual > RSSD Stored Procedures >* ***rs_helpclass***.

Assign Actions to Data Server Errors

Use the **assign action** command to specify the action to take for errors that a data server can return to Replication Server.

```
assign action
  {ignore | warn | retry_log | log | retry_stop | stop_replication}
  for error_class
  to server_error1 [, server_error2]...
```

You must create a default error class at a primary site before you can use **assign action** to change default error actions. The *data_server_error* parameter is the data server error number.

You can assign error classes to specific connections on replication databases using **create connection** and **alter connection**.

Enter one of the six possible error actions at the Replication Server where the error class was created: **ignore** is the least severe action and **stop_replication** is the most severe. See *Replication Server Reference Manual > Replication Server Commands > **assign action*** for error numbers, error messages, corresponding default error actions, and descriptions.

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.

You can also specify how Replication Server responds to SQLDML row count errors that may occur during SQL statement replication. In SQLDML row count errors, the number of rows changed in the primary and replicate databases do not match after SQL statement replication. The Replication Server default error action is to stop replication. The default Replication Server error class is `rs_repserver_error_class`.

This is an example of a row count error message:

```
DSI_SQLDML_ROW_COUNT_INVALID 5186
Row count mismatch for the SQL Statement Replication
command executed on 'mydataserver.mydatabase'. The
command impacted 10 rows but it should impact 15 rows.
```

Examples of Assigning an Error Action

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
```

For example, to warn if Replication Server encounters row count errors, which is indicated by error number 5186:

```
assign action warn
   for rs_repserver_error_class to 5186
```

See also

- *Error Actions for Data Server Errors* on page 298
- *Default Error Classes* on page 292

Error Actions for Data Server Errors

There are several errror actions you can assign for data server errors.

Table 28. 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.

Display Assigned Actions for Error Numbers

Use the **rs_helperror** stored procedure to display the action assigned for an error number.

The syntax for **rs_helperror** 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. See *Replication Server Reference Manual > RSSD Stored Procedures >* ***rs_helperror***.

Row Count Validation

Replication Server enables row count validation by default and automatically displays error messages and performs default error actions in reaction to different row count validation errors such as row count mismatch. You can configure the Replication Server error class to enable different error actions.

A connection associates itself with two error class types—a data server error class and a Replication Server error class. You must associate a Replication Server error class with a

connection before Replication Server can query the Replication Server error class for overrides to the default Replication Server error actions. You can associate a connection with only one Replication Server error class. However, you can associate one Replication Server error class with multiple connections. Use the **set replication server error class** parameter for the **create connection** and **alter connection** commands to associate a Replication Server error class with a connection.

When Replication Server responds to errors, it looks first for the Replication Server error class assigned to the connection. If Replication Server does not find the Replication Server error class, Replication Server uses the default **rs_repserver_error_class** error class assigned to the server.

Note: Replication Server ignores row count validation for those commands that are in a customized function string.

Control Row Count Validation

Use **dsi_row_count_validation** to disable row count validation.

If you have table rows that are not synchronized, and you want to bypass the default error actions and messages, you can set **dsi_row_count_validation** to **off** to disable row count validation.

dsi_row_count_validation is set to **on**, by default, to enable row count validation.

Use **configure replication server** to set **dsi_row_count_validation** at the server level to affect all replicate database connections, or use **alter connection** to set the parameter for a connection to a database and data server that you specify. For example, to:

- Disable row count validation for all database connections, enter:

  ```
  configure replication server
  set dsi_row_count_validation to 'off'
  ```

  You must suspend and resume all database connections to Replication Server after you execute **configure replication server** with **dsi_row_count_validation**. The change in setting takes effect after you resume database connections.
- Enable row count validation for a specific connection — `pubs2` database in SYDNEY_DS data server, enter:

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

  You need not suspend and resume a database connection when you set **dsi_row_count_validation** for the connection; the parameter takes effect immediately. However, the new setting affects the batch of replicated objects that Replication Server processes after you execute the command. Changing the setting does not affect the batch of replicated objects that Replication Server is currently processing.

Table Names Display in Row Count Validation Error Messages

Row count validation error messages display table names.

If you are using:

- Continuous mode log-order row-by-row replication – Replication Server logs and displays the table name, table owner name, and the number that identifies the output command that caused the transaction to fail. Replication Server logs only the first 30 bytes of the table name. You can enable the DSI_CHECK_ROW_COUNT_FULL_NAME trace to expand the maximum length of the table name that displays to 255 bytes.
- High volume adaptive replication (HVAR) or real-time loading (RTL) – Replication Server logs and displays the internal **join-update** and **join-delete** statements that result from HVAR and RTL compilation. You cannot obtain the specific command that caused the failed transaction since HVAR or RTL have already compiled the command as part of HVAR and RTL processing. The maximum length of the **join-update** and **join-delete** statements that Replication Server can display is 128 bytes including the `"...\0"` tail string.

This example consists of:

- Primary site – `pdb1` primary database with a table named `ThisTableHasANameLongerThan30Characters` that has three columns and three rows.

id	name	age
1	John	40
2	Paul	38
3	George	37

- Replicate site – `rdb1` primary database with a table with the same name `ThisTableHasANameLongerThan30Characters` that has two rows with values of 1 and 3 for the `id` column.

If you execute this command against `pdb1`:

```
update ThisTableHasANameLongerThan30Characters set age = 20
```

the error messages appear differently for each type of replication mode. In:

- Continuous mode log-order row-by-row replication:

```
I. 2010/06/07 01:30:21. DSI received Replication Server
error #5185 which is mapped to WARN by error action mapping.
W. 2010/06/07 01:30:21. WARNING #5185 DSI EXEC(103(1)
ost_replnx6_61.rdb1) - /dsiexec.c(11941)
```

```
Row count mismatch for the command executed on
'ost_replnx6_61.rdb1'. The command impacted 0 rows but it
should impact 1 rows.
I. 2010/06/07 01:30:21. The error was caused by output
command #3 of the failed transaction on table
'dbo.ThisTableHasANameLongerThan30C'.
```

Note: The table name is truncated to the default of 30 bytes.

If you turn on the **DSI_CHECK_ROW_COUNT_FULL_NAME** trace to enable the maximum table name length of 255 bytes that the error message can display, the last line of the error message displays the full table name:

```
I. 2010/06/07 02:22:55. The error was caused by output
command #3 of the failed transaction on table
'dbo.ThisTableHasANameLongerThan30Characters'.
```

- HVAR or RTL replication:

```
W. 2010/06/07 02:06:56. WARNING #5185 DSI EXEC(103(1)
ost_replnx6_61.rdb1) - i/hqexec.c(4047)
Row count mismatch for the command executed on
'ost_replnx6_61.rdb1'. The command impacted 1 rows but it
should impact 2 rows.
I. 2010/06/07 02:06:56. (HQ Error): update
ThisTableHasANameLongerThan30Characters set age = w.age
from ThisTableHasANameLongerThan30Characters
t,#rs_uThisTab...
I. 2010/06/07 02:06:57. The DSI thread for database
'ost_replnx6_61.rdb1' is shutdown.
```

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.

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.

See also

- *Access the Exceptions Log* on page 303

Handling of Failed Transactions

Learn 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**:

```
suspend connection to data_server.database
```

The command requires **sa** permission and must be entered at the Replication Server that manages the database.

Analyzing and Resolving the Problem

Determine why a transaction failed, make corrections or adjustments, and resubmit the transaction.

1. Retrieve a list of the transactions from the exceptions log.
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.

For example, if a transaction failed because the maintenance user had insufficient permissions, grant the maintenance user the needed permissions and retry the transaction.

See also

- *Access the Exceptions Log* on page 303

- *Delete Transactions from the Exceptions Log* on page 305

Resume the Connection

Use **resume connection** to restart the flow of transactions for a suspended database connection.

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.

```
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.

Access the Exceptions Log

Replication Manager provides a graphical interface to view and manage the transactions in the exceptions log.

Display Transactions in the Exceptions Log

Use the **rs_helpexception** stored procedure to display a summary of all transactions in the exceptions log.

```
rs_helpexception [transaction_id, [, v]]
```

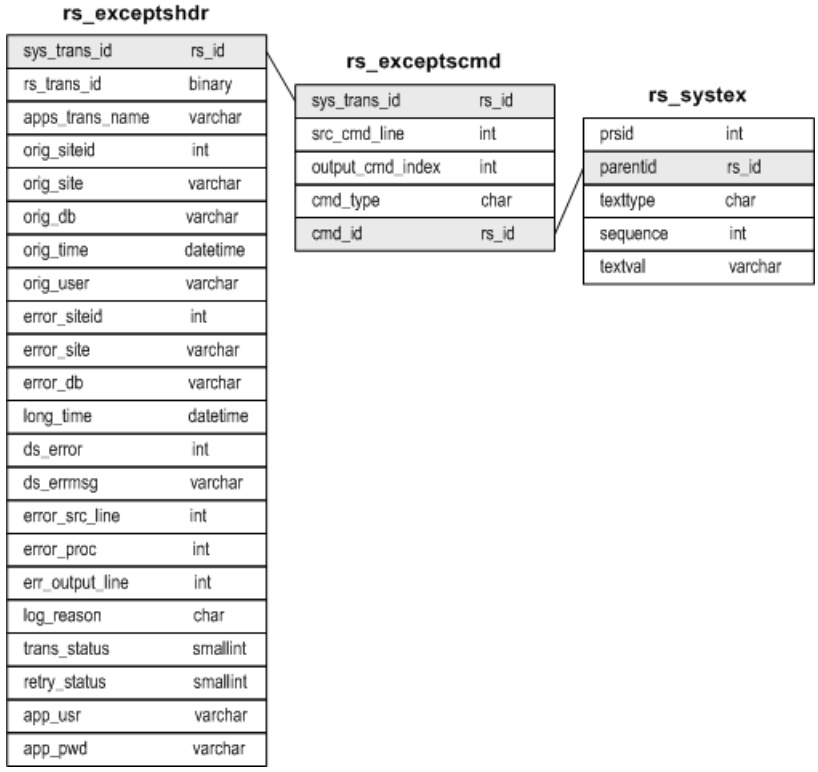
If you supply a valid *transaction_id* and **v** for "verbose" reporting, **rs_helpexception** displays a detailed description of a transaction. Use **rs_helpexception** with no parameters to obtain *transaction_id* numbers for all transactions in the exceptions log.

Query the Exceptions Log System Tables

You can join the `rs_exceptshdr` and `rs_exceptscmd` system tables on the `sys_trans_id` column to retrieve information about exceptions.

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 23: Exceptions Log System Tables



The `rs_exceptshdr` system table contains descriptive information about the transactions in the exceptions log, including:

- 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
```

See *Replication Server Reference Manual > Replication Server System Tables* for a list of all of the columns in these Replication Server system tables.

Delete Transactions from the Exceptions Log

Use stored procedures to delete transactions from the RSSD exceptions log.

- **rs_delexception** – deletes a single transaction from the exceptions log. The syntax is:

  ```
  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.

  For example, to delete the transaction with ID number 1234, enter:

  ```
  rs_delexception 1234
  ```

- **rs_delexception_id** – deletes a range of transactions identified by transaction ID. The syntax is:

  ```
  rs_delexception_id transaction_id_start [,transaction_id_end]
  ```

  For example, to delete all transactions with ID numbers between 1234 and 9800, inclusive,enter:

  ```
  rs_delexception_id 1234, 9800
  ```

- **rs_delexception_date** – deletes a range of transactions identified by transaction date. The syntax is:

  ```
  rs_delexception_date transaction_date_start
  [,transaction_date_end]
  ```

  For example, to delete from the exceptions log all transactions that have originating dates between 1st October 2010 and 31st October 2010, inclusive, enter:

  ```
  rs_delexception_date "10/01/2010", "10/31/2010"
  ```

- **rs_delexception_range** – deletes a range of transactions identified by originating site or user, or destination site. The syntax is:

  ```
  rs_delexception_range
  {{"origin"|"org"}, "origin_data_server.origin_database" |
  , {"destination"|"dest"},
  "destination_data_server.destination_database" |
  , "user", "origin_user"}
  ```

  For example, to delete from the exceptions log the transactions that originated from the `south_db` database of the SYDNEY_DS data server, enter:

  ```
  rs_delexception_range "org", "SYDNEY_DS.south_db"
  ```

See the descriptions of the stored procedures in *Replication Server Reference Manual > RSSD Stored Procedures* for complete usage information and more examples.

See *Replication Server Administration Guide Volume 1 > Manage Replication Environment with Sybase Central > Set up a Replication Environment > Manage Replication Server Objects > Queues > 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

Learn how to detect and resolve system transaction execution failures.

truncate table and certain supported DDL commands are not logged, although they can be replicated to standby and replicate databases. See 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.

The following message, after a system failure, indicates that a system command has not completed. The connection shuts down.

```
5152 DSI_SYSTRAN_SHUTDOWN,"There is a system
transaction whose state is not known. DSI will be
shutdown."
```

You must verify whether the command within the system transaction has executed at the replicate database.

- If the command has executed, or if you 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, 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.

See also

- *Supported DDL Commands and System Procedures* on page 61

Replication System Recovery

While Replication Server tolerates most failure conditions and recovers from them automatically, some failures require user intervention. Learn to identify those failures, and the procedures for recovery which are designed to maintain the integrity of the replication system by recovering lost and corrupted data and restoring that data to its previous state.

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.

In discussions about recovery, 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.

You can resynchronize the replicate databases in your replication environment if, for example, there is replication latency between primary and replicate databases such that to recover a database using replication alone is not feasible.

See also

- *Create Coordinated Dumps* on page 316
- *Replicate Database Resynchronization for Adaptive Server* on page 350

How to Use Recovery Procedures

When using recovery procedures, 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.

For each failure condition, there are corresponding failure symptoms and recovery procedures.

Warning! Use the recovery procedures only for the failure condition specific to the procedure. Do not use recovery procedures for replication system problems such as failure to replicate data. Attempting to use recovery procedures on conditions other than those specified can complicate your problem and require more drastic recovery actions.

See the *Replication Server Troubleshooting Guide* for help in diagnosing and correcting problems.

See also

- *Recovery from Partition Loss or Failure* on page 317
- *Recovery from Truncated Primary Database Logs* on page 321

- *Recovery from Primary Database Failures* on page 323
- *Recovery from RSSD Failure* on page 326
- *Replicate Database Resynchronization for Adaptive Server* on page 350

Configure the Replication System to Support Sybase Failover

Learn how Replication Server version 12.0 and later supports Sybase Failover available in Adaptive Server Enterprise version 12.0 and later.

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 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 companion 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 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 companion machine, reestablishing network connections.

See *Adaptive Server Enterprise > Using Sybase Failover in a High Availability System*.

See also

- *High Availability on Sun Cluster 2.2* on page 379

Enable 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 companion 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 and disk segment 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 for Sybase Failover Support

There are several requirements for Failover support.

- 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

Edit the configuration file to enable Failover support for an RSSD connection after you have installed the Replication Server.

You can also use **rs_init** to enable Failover support when you install a new Replication Server. See *Replication Server Configuration Guide > Configure Replication Server and Add Databases Using rs_init*.

1. 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.
2. Find the line `RSSD_ha_failover=no` and change it to `RSSD_ha_failover=yes`.

   You can disable Failover support for an RSSD connection by setting `RSSD_ha_failover=no`

   These changes take effect immediately; that is, you do not have to restart Replication Server to enable Failover support.

Enabling Failover Support for Non-RSSD Database Connections

Use **configure replication server** with **ha_failover** to enable Failover support for new database connections from the Replication Server to Adaptive Servers.

See *Adaptive Server Enterprise > Using Sybase Failover in a High Availability System*.

1. If necessary, start the Replication Server.

   See *Replication Server Administration Guide Volume 1 > Manage a Replication System > Starting Replication Server*.
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. Set **ha_failover** on.

   ```
   configure replication server
   set ha_failover to 'on'
   ```

Configure the Replication System to Prevent Data Loss

Learn the 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.

Routes Between Replication Servers

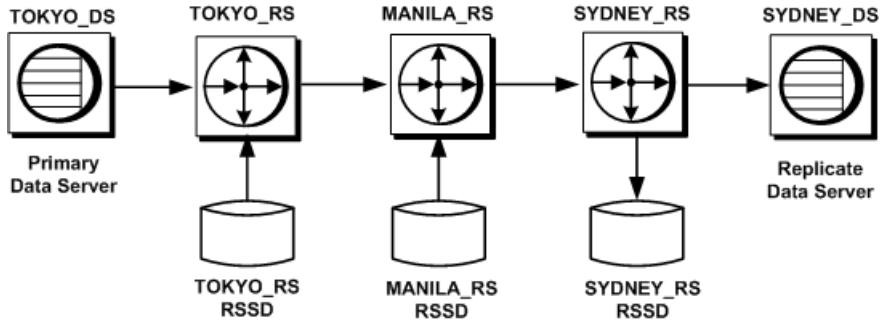
You can set save intervals for routes between Replication Servers for recovery of messages.

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* parameter 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 this figure, Replication Server TOKYO_RS maintains a direct route to MANILA_RS, and MANILA_RS maintains a direct route to SYDNEY_RS.

Figure 24: Save Interval Example



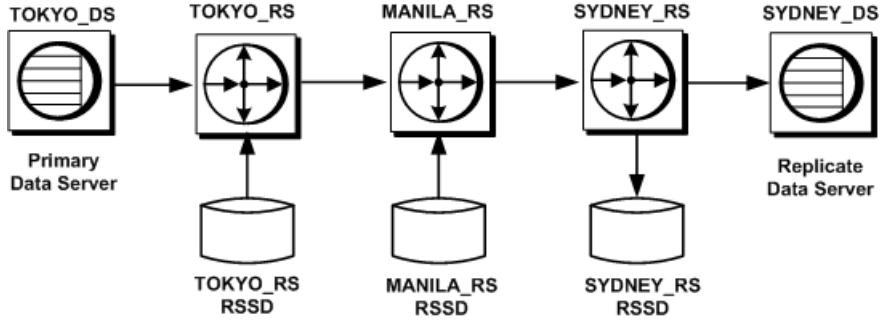
TOKYO_RS retains messages for a period of time after MANILA_RS has received them. If MANILA_RS experiences a partition failure, it requires that TOKYO_RS to resend the backlogged messages. MANILA_RS can also retain messages to allow SYDNEY_RS to recover from failures.

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.

Set the Save Interval for Routes

Execute the **alter route** command with the *save_interval* parameter at the source Replication Server to set the save interval for a route.

Figure 25: Save Interval Example



For example, to set Replication Server TOKYO_RS to save for one hour any messages destined for MANILA_RS, enter:

```
alter route to MANILA_RS
    set save_interval to '60'
```

By default, *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.

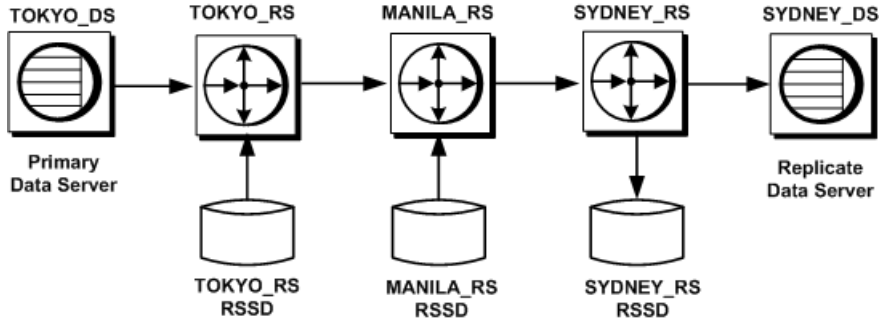
Connections Between Replication Servers and Data Servers

You can set save intervals for connections between Replication Servers for recovery of messages

When you set *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 **sysadmin restore_dsi_saved_segments**. See *Replication Server Reference Manual > Replication Server Commands >* ***sysadmin restore_dsi_saved_segments***.

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.

Figure 26: Save Interval Example



For example, in this figure, 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 *save_interval* for setting up a warm standby of a database that holds some replicate data or one that receives applied functions.

Set the Save Interval for Connections

Execute the **alter connection** command with the *save_interval* parameter at the Replication Server to set the save interval for a database connection.

For example, to set Replication Server SYDNEY_RS to save for one hour any messages destined for its replicate data server SYDNEY_DS, enter:

```
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 also

- *Configure Logical Connection Save Intervals* on page 104

Back up the RSSDs

Perform a dump of your RSSDs following any replication DDL, such as changing routes or adding subscriptions.

If you cannot recover the most recent state of an RSSD, recovery can be complex. The procedure you use depends on how much RSSD activity there has been since the last dump.

See also

- *Procedures to Recover an RSSD from Dumps* on page 327

Create 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.
- 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.

See also

- *Manage Function-String Classes* on page 26

Recovery 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: You can set save intervals for recovery.

See also

- *Save Interval for Recovery* on page 313

Symptoms of and Relevant Recovery Procedures for Partition Loss or Failure

Learn when to use and where to locate the appropriate recovery procedure for partition loss or failure.

Table 29. Symptoms of and Relevant Recovery Procedures for Partition Loss or Failure

Symptom	Use this procedure
Replication Server detects lost, damaged, or failed stable queue.	Recovering from partition loss or failure.
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.
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 the truncated message recovery from the database log to recover the database log. Then use message recovery from off-line database logs to rebuild the stable queues and recover lost messages.

See also

- *Recovering from Partition Loss or Failure* on page 318
- *Recovering Messages from Off-line Database Logs* on page 319
- *Recovering Messages from Truncated Primary Database Logs* on page 322

Recovering from Partition Loss or Failure

Recover from Replication Server partition loss or failure when Replication Server detects a lost, damaged,or failed stable queue.

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 is in use. If the partition is undamaged, Replication Server drops it only after all of the messages it holds are delivered and deleted. See *Replication Server Reference Manual > Replication Server Commands > **drop partition***.
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]
```

See *Replication Server Reference Manual > Replication Server Commands >* ***create partition***.

3. Since the partition is damaged, you must rebuild the stable queues:

   ```
   rebuild queues
   ```

   When all stable queues on the partition are removed, Replication Server drops the failed partition from the system and rebuilds the queues online using the remaining partitions.
4. After rebuilding the queues, check the Replication Server logs for loss detection messages.
5. If Replication Server detected message loss, do one of:

   - Perform message recovery from off-line database logs
   - 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.

Next

If you specify that Replication Server ignore message losses and you have rebuilt the queues of a Replication Server that is part a route, re-create subscriptions at the destination or use the **rs_subcmp** program with the **-r** flag to reconcile primary and replicate data.

See also

- *Rebuild Queues Online* on page 340
- *Loss Detection After Rebuilding Stable Queues* on page 342
- *Recovering Messages from Off-line Database Logs* on page 319

Recovering Messages from Off-line Database Logs

Recover messages from off-line logs after a partition failure.

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.

1. Restart Replication Server in standalone mode, using the **-M** flag.
2. Rebuild the stable queues. Log in to the Replication Server, and enter:

   ```
   rebuild queues
   ```
3. Inspect the Replication Server logs at each site for “Checking Loss” messages and use the date and time in the error log messages to determine which dumps to load.
4. Enable RepAgent for a temporary recovery database:

   ```
   sp_config_rep_agent temp_dbname, 'enable', \
   'rs_name', 'rs_user_name', 'rs_password'
   ```

See *Replication Server Administration Guide Volume 1 > Manage RepAgent and Support Adaptive Server > Set up RepAgent*.

5. Load the database dump and the first transaction log dump in to a temporary recovery database.
6. 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.
7. 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:

     ```
     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.
8. 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.
9. 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**.
10. 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.

11. Restart the Replication Server in normal mode.
12. Restart RepAgent for the original primary data server and database in normal mode.

See also

- *Rebuild Queues Online* on page 340
- *Determine Which Dumps to Load* on page 347
- *Determine Database Generation Numbers* on page 348
- *Loss Detection After Rebuilding Stable Queues* on page 342

Recovering Messages from the Online Database Log

Recover messages that are still in the online log at the primary database.

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.

Recovery from Truncated Primary Database Logs

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 cannot 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), recover messages from the online database log.
- If messages have been truncated from the online database log, recover the truncated messages 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.

See also

- *Recovering Messages from the Online Database Log* on page 321

Recovering Messages from Truncated Primary Database Logs

Recover truncated messages from the primary database log by replaying off-line transaction logs.

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.

   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 by doing either one of:

   - 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.
   - 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. You may need to execute **rs_zeroltm** to clear the locator information:

    ```
    rs_zeroltm data_server, database
    dbcc settrunc('ltm', 'valid')
    ```

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 also

- *Set Log Recovery for Databases* on page 346
- *Determine Database Generation Numbers* on page 348

Recovery from Primary Database Failures

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.

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.

Here are two possible scenarios for recovering from primary database failures:

- 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.
- 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.

Loading a Primary Database from Dumps

If you are loading only a primary database in a replication system, load the database to a previous state and resolve any inconsistencies with replicate databases.

1. Log in to the primary Replication Server to get the database generation number for the primary database:
   ```
   admin get_generation, data_server, database
   ```
   Make note of this number, which you will need for a later step.
2. Shut down the RepAgent for the primary database:
   ```
   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. Dump the transaction log.

   Enter:
   ```
   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 :
   ```
   use database
   go
   dbcc settrunc('ltm', 'gen_id', 1)
   ```
8. Clear the locator information.

   Enter:

```
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 *Replication Server Administration Guide Volume 1 > Manage Subscriptions > Obtain Subscription Information > Verify Subscription Consistency > Use* ***rs_subcmp*** *To Locate and Correct Inconsistencies*, and *Replication Server Reference Manual > Executable Programs >* ***rs_subcmp***.

Loading from Coordinated Dumps

Use this procedure only if you have coordinated dumps of both primary and replicate databases.The procedure loads a primary database and all replicate databases to the same state.

1. Perform steps 1 through 10 from the procedure for loading a primary database from dumps.
2. Suspend connections to the replicate databases that must be restored.
3. For each replicate database, log in to its managing Replication Server and suspend the connection to the database.

   Enter:

   ```
   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

   Enter:

   ```
   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 resume the connection to the database to restart the DSI for the database.

   Enter:

```
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.

Next

If any subscriptions were materializing when the failure occurred, drop them and re-create them.

See also

- *Loading a Primary Database from Dumps* on page 324

Recovery from RSSD Failure

If you cannot recover the most recent database state of the RSSD, 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 commands such as, cross-platform **dump** and **load,** or **bcp**. To migrate, you must 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 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 also

- *Procedures to Recover an RSSD from Dumps* on page 327

Procedures to Recover 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.

Table 30. Procedures to Recover from RSSD Failures

Activity since last RSSD dump	Use this procedure
No DDL activity	Basic RSSD Recovery Procedure.
DDL activity, but no new routes or subscriptions created	Subscription Comparison Procedure
DDL activity, but no new routes created	Subscription Comparison Procedure
New routes created	Deintegration/reintegration Procedure

See also

- *Using the Basic RSSD Recovery Procedure* on page 327
- *Using the Subscription Comparison Procedure* on page 330
- *Using the Subscription Re-Creation Procedure* on page 336
- *Using the Deintegration and Reintegration Procedure* on page 339

Using the Basic RSSD Recovery Procedure

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.

Warning! Do not execute any DDL commands until you have completed this recovery procedure.

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.

   Enter:

   ```
   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.

   Enter:

   ```
   rebuild queues
   ```

7. Start all RepAgents (except the RSSD RepAgent) that connect to the current Replication Server in recovery mode.

   Enter:

   ```
   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 using the procedure to recover from truncated primary database logs.

9. Shut down RepAgents for all primary databases managed by the current Replication Server.

   Enter:

   ```
   sp_stop_rep_agent dbname
   ```

10. Shut down Replication Server.

11. Move up the secondary truncation point.

    Execute the **dbcc settrunc** command at the Adaptive Server for the restored RSSD:

    ```
    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.

12. Clear the locator information.

    Enter:

    ```
    rs_zeroltm rssd_server, rssd_name
    go
    ```

13. 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.

    Enter:

    ```
    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.

14. 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.

15. 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.

See also

- *Rebuild Queues Online* on page 340
- *Recovery from Truncated Primary Database Logs* on page 321
- *Loss Detection After Rebuilding Stable Queues* on page 342

Using the 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.

1. To prepare the failed RSSD for recovery, perform steps 1 through 4 of the basic RSSD recovery procedure.
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 *Replication Server Administration Guide Volume 1 > Manage Replication Server Security* 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.

  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 *Replication Server Administration Guide Volume 1 > Manage Replication Server Security* 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.

     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 `rs_idnames` system table if they are missing.
c) Delete from the recovering RSSD `rs_idnames` system table any IDs of databases or Replication Servers that are no longer part of the replication system.
d) Ensure that the `rs_ids` system table is consistent.

   Execute in the RSSD of the current Replication Server this stored procedure:

   ```
   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 14 of the basic RSSD recovery procedure.
9. To complete RSSD recovery, re-execute any DDL commands executed at the current Replication Server since the last transaction dump.

See also

- *select Statements to Use for rs_subcmp on Replicated RSSD System Tables* on page 332
- *Using the Basic RSSD Recovery Procedure* on page 327

select Statements to Use for 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.

In the **select** statements listed , *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. See *Replication Server Reference Manual > Replication Server System Tables* for more information on the tables.

Note: These are the general form of the **select** statements. You may need to adjust these **select** statements in a mixed-version environment.

Table 31. General Form of select Statements for rs_subcmp Procedure

RSSD table name	select statement
rs_articles	`select * from rs_articles,rs_objects where rs_objects.prsid in sub_select and rs_articles.objid = rsobjects.objid order by articleid`
rs_classes	`select * from rs_classes where prsid in sub_select order by classid`
rs_columns	`select * from rs_columns where prsid in sub_select and rowtype = 1 order by objid, basecolnum, colname, colnum, version`
rs_databases	`select * from rs_databases where prsid in sub_select and rowtype = 1 order by dbid, dbname, dsname, ldbid, ltype, ptype`
rs_erroractions	`select * from rs_erroractions where prsid in sub_select order by ds_errorid, errorclassid`
rs_funcstrings	`select * from rs_funcstrings where prsid in sub_select and rowtype = 1 order by fstringid`
rs_functions	`select * from rs_functions where prsid in sub_select and rowtype = 1 order by funcid, funcname, objid`
rs_objects	`select * from rs_objects where prsid in sub_select and rowtype = 1 order by active_inbound, dbid, has_baserepdef, objid, objname, objtype, phys_tablename, phys_objowner, version`

RSSD table name	select statement
rs_publications	`select * from rs_publications where prsid in sub_select order by pubid`
rs_systext	`select * from rs_systext where prsid in sub_select and texttype in ('O', 'S') order by parentid, texttype, sequence`
rs_whereclauses	`select * from rs_whereclauses,rs_articles,rs_objects where rs_objects.prsid in sub_select and rs_articles.objid = rsobjects.objid and rs_whereclauses.articleid = rs_artilces.articleid order by wclauseid`

Classes and System Tables

Learn the impact of the subscription comparison procedure on classes and system tables and how to bring the RSSDs to a consistent state.

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 finds 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 this **select** statement for the `rs_classes` table:

```
select * from rs_classes where prsid = 0
    order by primary_keys
```

Example

Bring RSSDs to a consistent state.

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

Reconcile system tables with upstream RSSDs.

1. Run **rs_subcmp** against the 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:

   ```
   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

Reconcile system tables with downstream RSSDs.

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:

```
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
```

See *Replication Server Reference Manual > Executable Programs >* ***rs_subcmp***, and see *Replication Server Reference Manual > Replication Server System Tables*.

Using the Subscription Re-Creation Procedure

Restore an RSSD that requires that lost subscriptions be re-created 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.

This task 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. You also either delete or re-create subscriptions that are in limbo due to the loss of the RSSD.

In this procedure, however,

If DDL commands have been executed at the current Replication Server since the last transaction dump, you may have to reexecute them.

See *Replication Server Reference Manual > Executable Programs >* ***rs_subcmp***, and see *Replication Server Reference Manual > Replication Server System Tables*.

1. To prepare the failed RSSD for recovery, perform steps 1 through 4 of the basic RSSD recovery procedure.
2. To prepare the RSSDs of all upstream and downstream Replication Servers for recovery, perform step 2 through 3 of the subscription comparison procedure.
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 the subscription comparison procedure.

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 the subscription comparison procedure.
7. If the recovering Replication Server is an ID Server, to restore the IDs in its RSSD, perform step 6 of the subscription comparison procedure.
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 the subscription comparison procedure.
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 the basic RSSD recovery procedure.
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.

See also

- *Using the Basic RSSD Recovery Procedure* on page 327
- *Using the Subscription Comparison Procedure* on page 330

Using the Deintegration and Reintegration Procedure

If you created routes since the last time the RSSD was dumped, you are required to follow the deintegration and reintegration procedure.

1. Remove the current Replication Server from the replication system.

   See *Replication Server Administration Guide Volume 1 > Remove a Replication Server*.
2. Reinstall the Replication Server.

   Refer to the Replication Server installation and configuration guides for your platform for complete information on reinstalling Replication Server.
3. Re-create Replication Server routes and subscriptions.

   See *Replication Server Administration Guide Volume 1 > Manage Routes* and *Replication Server Administration Guide Volume 1 > Manage Subscriptions*.

Recovery Support Tasks

Standard recovery tasks let you manipulate and identify critical data in the replication system.

Use recovery tasks only for the procedure to which they apply.

Recovery support tasks include:

- Rebuilding stable queues
- Checking for Replication Server loss detection messages after rebuilding stable queues
- Putting Replication Server in log recovery mode and setting log recovery for databases
- Checking for Replication Server loss detection messages after setting log recovery for databases
- Determining which dumps and logs to load
- Adjusting database generation numbers

See also

- *Rebuild Stable Queues* on page 340
- *Loss Detection After Rebuilding Stable Queues* on page 342
- *Set Log Recovery for Databases* on page 346
- *Loss Detection After Setting Log Recovery* on page 347
- *Determine Which Dumps to Load* on page 347
- *Adjust Database Generation Numbers* on page 348

Rebuild 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.

See *Replication Server Reference Manual > Replication Server Commands > **rebuild queues***.

Rebuild 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 also

- *Rebuild Queues from Offline Database Logs* on page 340
- *Loss Detection After Rebuilding Stable Queues* on page 342

Rebuild Queues from Offline Database Logs

You can recover data from off-line database logs.

You use the **rebuild queues** command only after you have restarted the Replication Server in standalone mode. Executing **rebuild queues** in standalone mode puts Replication Server in recovery mode.

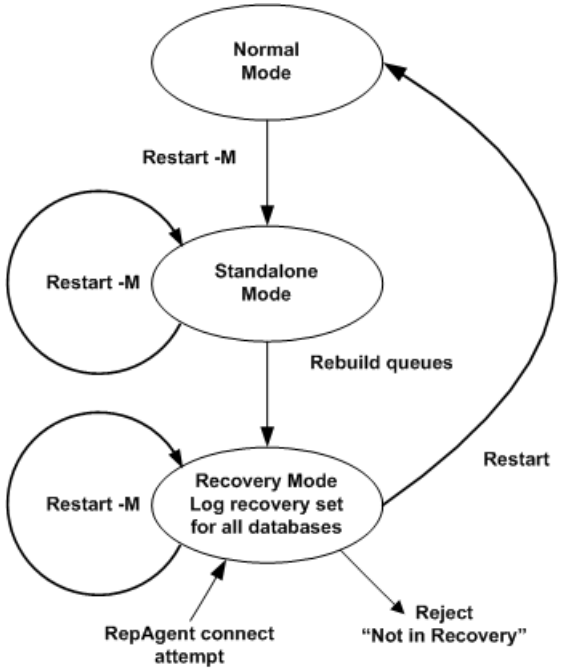
In recovery mode, the Replication Server allows only RepAgents in recovery mode to connect. If a RepAgent that is not in recovery mode attempts to connect, Replication Server rejects it with following error message:

```
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.

This figure illustrates the progression from normal mode to standalone mode to recovery mode using the **rebuild queues** command.

Figure 27: Enter Recovery Mode with the Rebuild Queues Command



See also

- *Replication Server Standalone Mode* on page 341

Replication Server Standalone Mode

Standalone mode allows you to review the contents of the stable queues because no messages are being written to or read from the queues.

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 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

Replication Server performs loss detection to determine if any messages could not be recovered after the stable queues were rebuilt.

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 – data lost between two Replication Servers, detected at the next downstream site
- DSI loss – data lost between a Replication Server and a replicate database that the Replication Server manages

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

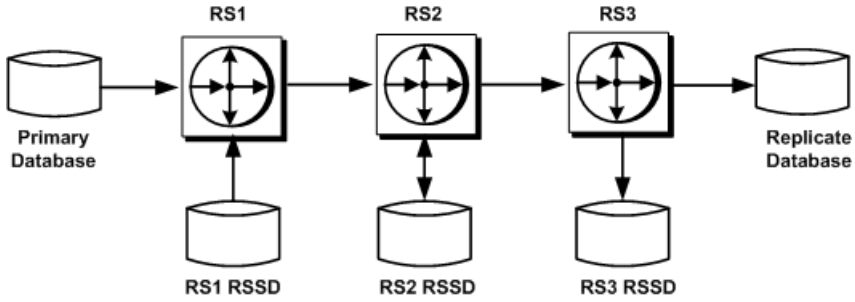
Learn how Replication Server detects data loss between two Replication Servers.

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 the figure, 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 28: 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

Learn how Replication Server detects data loss between between a Replication Server and a replicate database that the Replication Server manages.

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 of 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.

Recover a Loss

To recover the loss, you need to choose from one of three options.

You can choose to:

- Ignore the loss and continue, even though some messages may be lost. You can use the subscription comparison procedure that includes the **rs_subcmp** program with the **-r** flag to reconcile primary and replicate data.

  See also *Replication Server Administration Guide Volume 1 > Manage Subscriptions*, and *Replication Server Reference Manual > Executable Programs > **rs_subcmp***.
- 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.

See also

- *Using the Subscription Comparison Procedure* on page 330

Ignore a Loss

You must execute an **ignore loss** command in certain situations.

Execute **ignore loss**:

- 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 at Replication Server 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.

Set Log Recovery for Databases

Manually setting log recovery is part of recovering from truncated primary database logs off-line, or restoring primary and replicate databases from dumps.

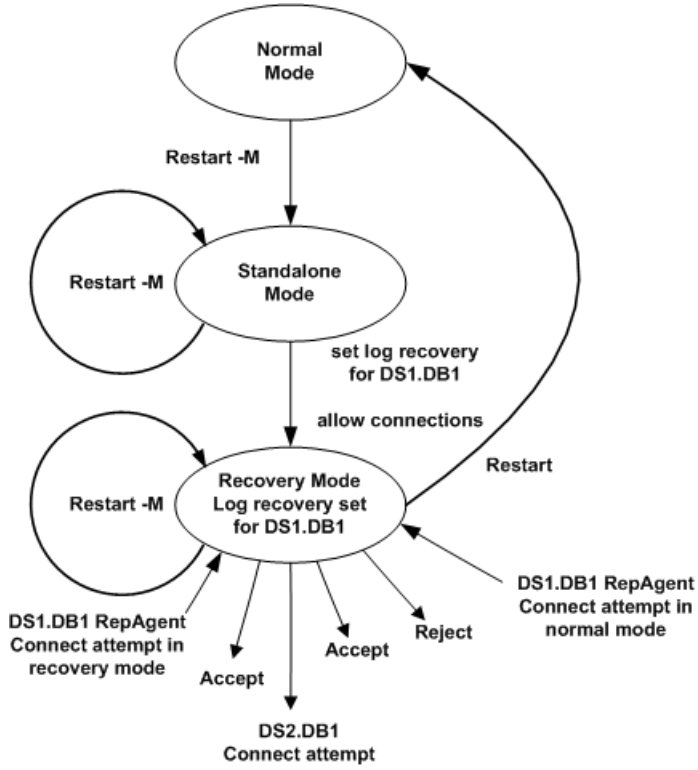
While the procedure to rebuild queues off-line automatically sets log recovery for all databases, manually setting log recovery allows you to recover each database without reconstructing the stable queue.

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.

This figure 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 29: 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. You must correctly determine which dumps to load.

See also

- *Determine Which Dumps to Load* on page 347

Determine 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, load the dumps taken just before November 1, 2011 at 10:58 p.m.:

```
Checking Loss for DS3.RDB from DS1.PDB
date=Nov-01-2011 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.

Adjust Database Generation Numbers

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.

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.

The maximum value for the database generation number is 65,535. Sybase recommends that you avoid incrementing the number to high values unless absolutely necessary.

If you want to reset the database generation number, you must rebuild the replication environment. Rebuilding the environment includes deleting the connection to the primary database where you want to reset the database generation number, recreating the connection, and then rebuilding the replication configuration of the primary database.

Determine Database Generation Numbers

Learn when to adjust 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 loading a primary database from dumps or loading from coordinated dumps.

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.

See also

- *Loading a Primary Database from Dumps* on page 324
- *Loading from Coordinated Dumps* on page 325

Dumps and Database Generation Numbers

Learn how and when to adjust database generation numbers after reloading dumps.

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 32. 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.

Resetting Primary Database Generation Number

Learn how to reset database generation numbers.

In this procedure, the primary database refers to the primary database where you want to reset the database generation number.

1. At the replicate Replication Server, drop all subscriptions that reference replication definitions and publications defined for the connection to the primary database.
2. Drop all publications referenced by the subscriptions you dropped in step 1.
3. Drop all articles referenced by the publications you dropped in step 2.
4. In the primary Replication Server, drop all replication definitions for the primary database connection.
5. In the primary Replication Server, drop the connection to the primary database, and all connections to replicate databases that subscribe to the primary database.
6. Set the database generation number to 0 on the primary database:
   - In Adaptive Server:
     ```
     dbcc settrunc('ltm', 'gen_id', 0)
     ```
   - In Replication Agent for IBM DB2 UDB on UNIX and Windows, Microsoft SQL Server, and Oracle:
     ```
     pdb_gen_id 0
     ```
7. In the primary Replication Server, create a new connection to the primary database, and create connections to the replicate databases.
8. Re-create all the replication definitions, publications, articles, and subscriptions you dropped. See *Replication Server Administration Guide Volume 1 > Manage Replication Environment with Sybase Central > Set up a Replication Environment*.

Support for Non-ASE Databases

You can reset database generation numbers for all supported non-ASE databases acting as the primary database.

See the *Replication Server Heterogeneous Replication Guide* for supported primary databases.

Replicate Database Resynchronization for Adaptive Server

Replication Server allows you to resynchronize and materialize the replicate database, and resume further replication without losing or risking inconsistency of data, and without forcing a quiesce of your primary database.

Database resynchronization is based on obtaining a dump of data from a trusted source and applying the dump to the target database you want to resynchronize.

To resynchronize Oracle databases, see *Replication Server Heterogeneous Replication Guide > Oracle Replicate Databases Resynchronization*.

Configuring Database Resynchronization

Use commands and parameters from both Replication Server and RepAgent to configure database resynchronization.

1. Stop replication processing by suspending RepAgent.
2. Place Replication Server in resync mode.
   In resync mode, Replication Server skips transactions and purges replication data from replication queues in anticipation of the replicate database being repopulated from a dump taken from the primary database or trusted source.
3. Restart RepAgent and send a resync database marker to Replication Server to indicate that a resynchronization effort is in progress.
4. Verify that DSI receives the resync database marker.
5. Obtain a dump from the primary database.

   When Replication Server detects a dump marker that indicates the completion of the primary database dump, Replication Server stops skipping transactions and can determine which transactions to apply to the replicate database.
6. Verify that DSI receives the dump database marker.

   Note: Sending a dump database marker does not apply in cases where you send the resync marker with the **init** instruction.
7. Apply the dump to the replicate database.
8. Resume replication.

Instructing Replication Server to Skip Transactions

Use the **skip to resync** parameter with the **resume connection** command to instruct Replication Server to skip transactions in the DSI outbound queue for the specified replicate database until Replication Server receives and acknowledges a dump database marker sent by RepAgent.

Replication Server does not process records in the outbound queue, since the data in the replicate database is expected to be replaced with the dump contents.

See *Replication Server Reference Manual* > *Replication Server Commands* > ***resume connection***.

Run:

```
resume connection to data_server.database skip to resync marker
```

Warning! If you execute **resume connection** with the **skip to resync marker** option on the wrong connection, data on the replicate database becomes unsynchronized.

When you set **skip to resync marker**, Replication Server does not log the transactions that are skipped in the Replication Server log or in the database exceptions log. Replication Server logs transactions that are skipped when you set **skip [n] transaction**.

Send the Resync Database Marker to Replication Server

Instruct RepAgent to send a resync database marker to Replication Server to indicate that a resynchronization effort is in progress.

When you restart RepAgent in resync mode, RepAgent sends the resync database marker to Replication Server as the first message before it sends any SQL data definition language (DDL) or data manipulation language (DML) transactions. Multiple replicate databases for the same primary database all receive the same resync marker since they each have a DSI outbound queue.

For each DSI that resumes with the **skip to resync marker** parameter, the DSI outbound queue records in the Replication Server system log that DSI has received the resync marker and also records that from that point forward, DSI rejects committed transactions until it receives the dump database marker.

In Adaptive Server, use **sp_start_rep_agent** with the **resync**, **resync purge,** or **resync init** parameters to support the corresponding options for sending the resync database marker.

Sending a Resync Marker Without Any Option

Send a resync marker using **sp_start_rep_agent** without any option when there is no change to the truncation point and the expectation is that the RepAgent should continue processing the transaction log from the last point that it processed.

Syntax: **sp_start_rep_agent** *database_name*, **'resync'**

Each outbound DSI thread and queue receives and processes the resync database marker. DSI reports to the Replication Server system log when a resync marker has been received, satisfying the skip to resync marker request of DSI. Subsequently, DSI rejects committed transactions while it waits for a dump database marker. With this message and the change of behavior to one of waiting for the dump database marker, you can apply any dump to the replicate database.

Sending a Resync Marker with a purge Instruction

Send a resync marker using **sp_start_rep_agent** with the **purge** option to instruct Replication Server to purge all open transactions from the inbound queue, and reset duplicates detection, before receiving any new inbound transactions.

Syntax: **sp_start_rep_agent** *database_name*, **'resync purge'**

Use the **purge** option when the truncation point of the primary database has been moved, which occurs when you:

- Manually change the truncation point.
- Disable RepAgent.
- Execute Adaptive Server commands such as, **dbcc dbrepair**.

Since the truncation point has changed, open transactions in the Replication Server inbound queue must be purged because these transactions do not match new activity sent from the new

secondary truncation point. Replication Server resets checking for duplicates since the changed truncation point could send a record with a previous origin queue ID (OQID). Since the prior data is purged from the queues, Replication Server does not treat any new activity from the RepAgent as duplicate activity, and consequently does not reject the new activity. The purge option does not change DSI processing because Replication Server continues to reject outbound queue commands while waiting for the dump database marker.

Sending a Resync Marker with the init Command

Send a resync marker with an **init** command using **sp_start_rep_agent** with the init option to instruct Replication Server to purge all open transactions from the inbound queue, reset duplicate detection, and suspend the outbound DSI.

Syntax: **sp_start_rep_agent** *database_name*, **'resync init'**

Use this option to reload the primary database from the same dump as the replicate database. Since there is no dump taken from the primary database, RepAgent does not send a dump database marker. Instead of waiting for a dump database marker after the resync marker, the **init** option suspends the DSI connection immediately after Replication Server processes the resync marker.

After DSI is suspended, all subsequent activity through DSI consists of new transactions. You can resume DSI once you reload the replicate database from the same dump you used for the primary.

Obtain a Dump of the Database

Use the **dump database** Adaptive Server command.

See *Adaptive Server Enterprise > System Administration Guide: Volume 2 > Developing a Backup and Recovery Plan > Using the* ***dump*** *and* ***load*** *Commands*.

Send the Dump Database Marker to Replication Server

RepAgent automatically generates and sends a dump database marker to Replication Server when you obtain a dump of the primary database.

Note: Sending a dump database marker does not apply when you send the resync marker with the **init** instruction.

You can manually resume DSI after you apply the dump to the replicate database.Transactions that commit after the dump point, which is indicated by the dump database marker, are replicated.

Monitor DSI Thread Information

Use the **admin who** command to provide information on DSI during database resynchronization.

State	Description
SkipUntil Resync	DSI resumes after you execute **skip to resync**. This state remains until DSI receives a resync database marker.
SkipUntil Dump	DSI has received a resync database marker. This state remains until DSI has processed a subsequent dump database marker.

Apply the Dump to a Database to be Resynchronized

You can load the primary database dump to the replicate database only after you see the relevant messages in the system log.

- When Replication Server receives the resync database marker with or without the **purge** option, and the dump database marker:

  ```
  DSI for data_server.database received and processed
  Resync Database Marker. Waiting for Dump Marker.
  ```

  ```
  DSI for data_server.database received and processed
  Dump Marker. DSI is now suspended. Resume after database has been
  reloaded.
  ```

- When Replication Server receives the resync database with **init** marker:

  ```
  DSI for data_server.database received and processed
  Resync Database Marker. DSI is now suspended. Resume after
  database has been reloaded.
  ```

See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***load database*** for instructions about loading the dump to the database you want to resynchronize.

Database Resynchronization Scenarios

Follow the procedure to resynchronize databases in different scenarios. After completing a procedure, the primary and replicate databases are transactionally consistent.

To execute a procedure, you must:

- Be a replication system administrator
- Have an existing replication environment that is running successfully
- Have methods and processes available to copy data from the primary database to the replicate database

For commands and syntax for RepAgent for Adaptive Server and Replication Server, see the *Replication Server Reference Manual* and *Replication Server Administration Guide Volume 1 > Manage RepAgent and Support Adaptive Server*.

Resynchronize One or More Replicate Databases Directly from a Primary Database

Resynchronize one or multiple replicate databases from a single primary database.

This procedure with minor variations, allows you to:

- Repopulate the replicate database when the replication latency between primary and replicate databases is such that to recover a database using replication is not feasible, and reporting based on the replicate data may not be practical because of the latency.
- Repopulate the replicate database with trusted data from the primary database.
- Coordinate resynchronization when the primary database is the source for multiple replicate databases.
- Coordinate resynchronization if the primary site is a logical database that consists of a warm standby pair of databases that you want to resynchronize with one or more replicate databases. In a warm standby pair, the active database acts as the primary database, and the standby acts as the replicate database. Therefore, the active database of a warm standby pair at a primary site also appears as a primary database to one or multiple replicate databases.

Resynchronizing Directly from a Primary Database

Resynchronize a replicate database from a primary database.

1. Stop replication processing by RepAgent. In Adaptive Server, execute:
   ```
   sp_stop_rep_agent database
   ```
2. Suspend the Replication Server DSI connection to the replicate database:
   ```
   suspend connection to dataserver.database
   ```
3. Instruct Replication Server to remove data from the replicate database outbound queue and wait for a resync marker from the primary database RepAgent:
   ```
   resume connection to data_server.database skip to
   resync marker
   ```
4. Instruct RepAgent to start in resync mode and send a resync marker to Replication Server:
   - If the truncation point has not been moved from its original position, in Adaptive Server execute:
     ```
     sp_start_rep_agent database, 'resync'
     ```
   - If the truncation point has been moved from its original position, in Adaptive Server execute:
     ```
     sp_start_rep_agent database, 'resync purge'
     ```
5. In the Replication Server system log, verify that DSI has received and accepted the resync marker from RepAgent by looking for this message:
   ```
   DSI for data_server.database received and processed
   Resync Database Marker. Waiting for Dump Marker.
   ```

Note: If you are resynchronizing multiple databases, verify that the DSI connection for each of the databases you want to resynchronize has accepted the resync marker.

6. Obtain a dump of the primary database contents. See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***dump database***. Adaptive Server automatically generates a dump database marker.
7. Verify that Replication Server has processed the dump database marker by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Dump Marker. DSI is now suspended. Resume after database has been
   reloaded.
   ```

   When Replication Server receives the dump marker, the DSI connection automatically suspends.
8. Apply the dump of the primary database to the replicate database. See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***load database***.
9. After you apply the dump to the replicate database, resume DSI:

   ```
   resume connection to data_server.database
   ```

Resynchronizing Using a Third-Party Dump Utility

Coordinate resynchronization after you dump the primary database using a third-party dump utility, such as a disk snapshot.

Third-party tools do not interact as closely with the primary database as native database dump utilities. If your third-party tool does not record anything in the primary database transaction log that RepAgent can use to generate a dump database marker, generate your own dump database markers to complete the resynchronization process. See your third-party tool documentation.

1. Stop replication processing by RepAgent. In Adaptive Server, execute:

   ```
   sp_stop_rep_agent database
   ```
2. Suspend the Replication Server DSI connection to the replicate database:

   ```
   suspend connection to dataserver.database
   ```
3. Instruct Replication Server to remove data from the replicate database outbound queue and wait for a resync marker from the primary database RepAgent:

   ```
   resume connection to data_server.database skip to
   resync marker
   ```
4. Obtain a dump of the primary database contents using the third-party dump utility.
5. Determine the dump point based on information from the primary database when you took the dump, or information from the third-party tool. With a third-party tool, you are responsible for determining the dump point. For example, if you are using a disk replication tool, you can temporarily halt activity at the primary database to eliminate transactions in progress from the disk snapshot, and then use the “end of transaction log” point as the dump database marker.

6. Execute the **rs_marker** stored procedure on the primary database for RepAgent to mark the end of the dump position that you obtained in step 5:

   ```
   rs_marker “dump database database_name 'current date' oqid"
   ```

   where *current date* is any value in `datetime` format and *oqid* is any valid hexadecimal value. See *Replication Server Reference Manual > Topics > Datatypes > Date/time, and Date and Time Datatypes > Entry Format for Date/Time Values*.

   For example, you can mark the end of the dump position on the `rdb1` database with a date and time value of "20110915 14:10:10" and a value of 0x0003 for *oqid*:

   ```
   rs_marker “dump database rdb1 '20110915 14:10:10' 0x0003"
   ```

   RepAgent automatically generates a dump database marker for the point you marked in step 6, and sends the dump database marker to Replication Server.

7. Instruct RepAgent to start in resync mode and send a resync marker to Replication Server:

   - If the truncation point has not been moved from its original position, execute this command in Adaptive Server:

     ```
     sp_start_rep_agent database, 'resync'
     ```

   - If the truncation point has been moved from its original position, execute this command in Adaptive Server:

     ```
     sp_start_rep_agent database, 'resync purge'
     ```

8. Verify that DSI has received and accepted the resync marker from Replication Agent by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Resync Database Marker. Waiting for Dump Marker.
   ```

9. Verify that Replication Server has processed the dump database marker by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Dump Marker. DSI is now suspended. Resume after
   database has been reloaded.
   ```

   When Replication Server receives the dump marker, the DSI connection automatically suspends.

10. Apply the dump of the primary database from the third-party tool to the replicate database. See your Adaptive Server and third-party tool documentation.

11. After you apply the dump to the replicate database, resume DSI:

    ```
    resume connection to data_server.database
    ```

Resynchronizing if There is No Support for the Resync Database Marker

Coordinate resynchronization if the RepAgent or the primary database have not been updated to support automatic generation of a resync marker.

Note: You can use this procedure for Adaptive Server only.

1. Suspend the Replication Server DSI connection to the replicate database:

   ```
   suspend connection to dataserver.database
   ```

2. Instruct Replication Server to remove data from the replicate database outbound queue and wait for a resync marker from the primary database RepAgent:

   ```
   resume connection to data_server.database skip to
   resync marker
   ```

3. Ensure that there are no open transactions in system log, and then in the primary database, manually generate the **resync marker**:

   ```
   execute rs_marker 'resync database'
   ```

4. In the Replication Server system log, verify that DSI has received and accepted the resync marker from RepAgent by looking for this message:

   ```
   DSI for data_server.database received and processed
   Resync Database Marker. Waiting for Dump Marker.
   ```

5. Obtain a dump of the primary database contents.

   Adaptive Server automatically generates a dump database marker.See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***dump database***.

6. Verify that Replication Server has processed the dump database marker by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Dump Marker. DSI is now suspended. Resume after database has been
   reloaded.
   ```

   When Replication Server receives the dump marker, the DSI connection automatically suspends.

7. Apply the dump of the primary database to the replicate database. See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***load database***.

8. After you apply the dump to the replicate database, resume DSI:

   ```
   resume connection to data_server.database
   ```

Resynchronizing Both the Primary and Replicate Databases from the Same Dump

Coordinate resynchronization to reload both the primary database and replicate database from the same dump or copy of data. No dump database marker is needed, since you are not obtaining a dump from the primary database.

1. Stop replication processing by RepAgent. Do not alter the truncation point.

   In Adaptive Server, execute:

   ```
   sp_stop_rep_agent database
   ```

2. Suspend the Replication Server DSI connection to the replicate database:

   ```
   suspend connection to data_server.database
   ```

3. Instruct Replication Server to remove data from the replicate database outbound queue and wait for a resync marker from the primary database RepAgent:

```
resume connection to data_server.database skip to
resync marker
```

4. Obtain the RepAgent settings before you apply the dump.

   Note: Adaptive Server stores, within the database, the connectivity settings and other configurations that RepAgent uses. If you load the primary database from a dump that you took from a different database, RepAgent loses its configuration settings, or the settings change to match the settings of the database from which you took the dump.

5. Apply the dump of the data from the external source to the primary database.

   After you apply the dump, reset the RepAgent configurations to the settings that existed before you applied the dump.

6. Make sure that the last primary database transaction log page does not contain any operation that can affect replicate database tables by executing at the primary Adaptive Server database:

   ```
   rs_update_lastcommit 0, 0, 0, ""
   go 100
   ```

7. Move the truncation point to the end of the transaction log for the primary database. In Adaptive Server, execute:

   ```
   dbcc settrunc('ltm', 'end')
   go
   ```

8. Instruct RepAgent to start in resync mode with the **init** instruction. In Adaptive Server, execute:

   ```
   sp_start_rep_agent database, 'resync init'
   ```

9. Verify that DSI has received and accepted the resync marker from the RepAgent by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Resync Database Marker. DSI is now suspended. Resume
   after database has been reloaded.
   ```

   When Replication Server receives and processes the resync database with **init** marker, the DSI connection suspends.

10. Apply the dump of the data from the external source to the replicate database.

11. After you apply the dump to the replicate database, resume DSI to the replicate database to allow Replication Server to apply transactions from the primary database:

    ```
    resume connection to data_server.database
    ```

Resynchronizing the Active and Standby Databases in a Warm Standby Application

Resynchronize the active and standby databases in a warm standby environment, when the warm standby pair is the replicate site for a single primary database.

In this scenario, the replicate site is a warm standby pair that consists of the active and standby databases that act as a single logical database.

```
Primary ---> Replication ---> Replicate logical database
database     Server           [Active+Standby warm standby
                               pair]
```

The resynchronization scenario procedure is a two-step process—resynchronize the replicate active database of the warm standby pair with a dump from the primary database, and then resynchronize the replicate standby database of the warm standby pair with a dump from the active database or the existing dump from the primary database.

1. Stop replication processing by both the primary database RepAgent and the warm standby active database RepAgent.

   In Adaptive Server, execute:

   ```
   sp_stop_rep_agent database
   ```

2. Suspend the Replication Server DSI connection to the active and standby databases:

   ```
   suspend connection to dataserver.database
   ```

3. Instruct Replication Server to remove data from the outbound queue of the active and standby databases, and wait for a resync marker from the primary database RepAgent:

   ```
   resume connection to data_server.database skip to
   resync marker
   ```

4. Instruct the primary database RepAgent to start in resync mode and send a resync marker to Replication Server.

   - If the truncation point has not been moved from its original position, execute this command in Adaptive Server:

     ```
     sp_start_rep_agent database, 'resync'
     ```

   - If the truncation point has been moved from its original position, execute this command in Adaptive Server:

     ```
     sp_start_rep_agent database, 'resync purge'
     ```

5. Verify that DSI for the active database has received and accepted the resync marker from the primary database RepAgent by looking for this message in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Resync Database Marker. Waiting for Dump Marker.
   ```

6. Obtain a dump of the primary database contents. See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***dump database***. Adaptive Server automatically generates a dump database marker.

7. Obtain the RepAgent settings before you apply the dump.

Note: Adaptive Server stores, within the database, the connectivity settings and other configurations that RepAgent uses. If you load the primary database from a dump that you took from a different database, RepAgent loses its configuration settings, or the settings change to match the settings of the database that you took the dump from.

8. Verify that the Replication Server DSI for the active database has processed the dump database marker by looking for this message from the active database in the Replication Server system log:

   ```
   DSI for data_server.database received and processed
   Dump Marker. DSI is now suspended. Resume after database has been
   reloaded.
   ```

9. Apply the dump of the primary database to the active database. See *Adaptive Server Enterprise Reference Manual: Commands > Commands >* ***load database***.

   After you apply the dump, reset the RepAgent configurations to the settings that existed before you applied the dump.

10. Make sure that the last primary database transaction log page does not contain any operation that can affect replicate database tables by executing at the primary Adaptive Server database:

    ```
    rs_update_lastcommit 0, 0, 0, ""
    go 100
    ```

11. Move the truncation point to the end of the transaction log for the active database. In Adaptive Server, execute:

    ```
    dbcc settrunc('ltm', 'end')
    go
    ```

12. Instruct RepAgent to start in resync mode with the **init** instruction. In Adaptive Server, execute:

    ```
    sp_start_rep_agent database, 'resync init'
    ```

13. Verify that DSI for the standby database has received and accepted the resync marker from the active database RepAgent by looking for this message in the Replication Server system log:

    ```
    DSI for data_server.database received and processed
    Resync Database Marker. DSI is now suspended. Resume
    after database has been reloaded.
    ```

    When Replication Server receives and processes the resync database with **init** marker, the DSI connection suspends.

14. Obtain a dump of the active database contents and apply the dump to the standby database. You can also apply the dump of the primary database from step 6 if the dump does not include database configuration information.

15. Resume DSI to the active and standby databases:

    ```
    resume connection to data_server.database
    ```

Asynchronous Procedures

Learn about asynchronous stored procedures, and the method for replicating stored procedures that are associated with table replication definitions. This method is supported for applications that require it.

See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions* for information about replicated stored procedures that are associated with function replication definitions.

See *Replication Server Design Guide* for information about replication system design issues relating to replicated stored procedures.

Introduction to Asynchronous Procedure Delivery

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.

Replicated Stored Procedures Logging by Adaptive Server

Learn how Adaptive Server determines the database in which a replicated stored procedure execution is logged.

Stored procedure are logged in the database in which the enclosing transaction was started.

- If the user does not explicitly begin a transaction, Adaptive Server begins 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 is still ogged 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.

Restriction to Logging of Replicated Stored Procedures

Replicated Adaptive Server stored procedures cannot contain parameters with the `text`, `unitext`, or `image` datatypes.

See the *Adaptive Server Reference Manual*

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 may occur when 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. There are several warning conditions and corresponding actions that Replication Server takes if it fails to deliver the applied stored procedure.

See also

- *Warning Conditions* on page 368

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

There are several prerequisites for implementing applied or request stored procedures.

- Understand how you will use asynchronous procedure delivery to meet your application needs. See the *Replication Server Design Guide*.
- Set up a RepAgent for the stored procedure, even if the database contains no primary data (such as when using request functions). See the Replication Server installation and configuration guides for your platform.
- 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.

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 tasks in this section 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.

See also

- *Function Strings and Function-string Classes* on page 32

Implementing an Applied Stored Procedure

Learn the steps to implement an applied stored procedure.

Prerequisites

Verify that you have completed the asynchronous stored procedure prerequisites.

Task

See *Replication Server Reference Manual > RSSD Stored Procedures* for information about stored procedures used to query the RSSD for system information.

1. Set up replicate databases that contain replicate tables. These tables may or may not match the replication definition for the primary table.
2. 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 *Replication Server Administration Guide Volume 1 > Manage Routes*.

3. Locate or create a replication definition on the primary Replication Server that identifies the table to be modified.

   See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables*.

4. In the primary database, mark the table for replication using either the **sp_setreplicate** or **sp_setreptable** system procedure.

   For example, for a table named `employee`, enter one of:

   - `sp_setreplicate employee, 'true'`

     Follow the guidelines when specifying stored procedures and tables and for replication.
   - `sp_setreptable employee, 'true'`

     For **sp_setreptable**, the single quotes are optional. See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables > Mark Tables for Replication > Use the sp_setreptable System Procedure*.

5. 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. Check the warning conditions. 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.

6. In the primary database, mark the stored procedure for replication using either **sp_setreplicate** or **sp_setrepproc**.

   For example, enter one of:

   - `sp_setreplicate upd_emp, 'true'`

     Follow the guidelines when specifying stored procedures and tables and for replication.
   - `sp_setrepproc upd_emp, 'table'`

     See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions > Mark Stored Procedures for Replication*.

7. At the replicate Replication Servers, create subscriptions to a replication definition for the table that the stored procedure at the primary database updates.

See *Replication Server Administration Guide Volume 1 > Manage 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.

8. 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
   ```

9. Grant **execute** permission on the stored procedure to the maintenance user.

   For example:

   ```
   grant execute on upd_emp to maint_user
   ```

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)
    ```

    Only one user-defined function is shared by all replication definitions for the same table. You can specify the name of any of these replication definitions.

11. Verify that all Replication Server and database objects in all the steps exist at the appropriate locations.

See also

- *Asynchronous Stored Procedure Prerequisites* on page 366
- *Specify Stored Procedures and Tables for Replication* on page 372

Warning Conditions

Replication Server warning conditions occur when an applied stored procedure is not delivered at a replicate database.

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 are 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

Warning Conditions and Replication Server Actions

- 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.

Implementing a Request Stored Procedure

Learn the steps to implement an request stored procedure.

Prerequisites

Verify that you have completed the asynchronous stored procedure prerequisites.

Task

See *Replication Server Reference Manual > RSSD Stored Procedures* for information about stored procedures used to query the RSSD for system information.

1. 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 *Replication Server Administration Guide Volume 1 > Manage Routes*.
2. Create a login name and password at the primary Replication Server for the user at the replicate Replication Server.

   See *Replication Server Administration Guide Volume 1 > Manage Replication Server Security*.
3. At the replicate Replication Server, create the necessary permissions for this user to execute the stored procedure at the primary Replication Server.

   See *Replication Server Administration Guide Volume 1 > Manage Replication Server Security*.
4. At the primary Replication Server, locate or create a replication definition that identifies the table to be modified.

   See *Replication Server Administration Guide Volume 1 > Manage Replicated Tables* for information on creating replication definitions.

   The replicate Replication Server may have subscriptions on the replication definition.
5. 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, follow the guidelines for specifying a nonunique name for a user-defined function.

6. In the replicate database, use the **sp_setreplicate** or **sp_setrepproc** system procedure to mark the stored procedure for replication.

   For example, enter one of:

   ```
   sp_setreplicate upd_emp, 'true'
   ```

   Follow the guidelines when specifying stored procedures and tables and for replication.

   or

   ```
   sp_setrepproc upd_emp, 'table'
   ```

   See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions > Mark Stored Procedures for Replication*.

7. 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.

   You can map the function to a different stored procedure name.

8. 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
   ```

9. 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)
   ```

10. Verify that all Replication Server and database objects in all the exist at the appropriate locations.

See also

- *Asynchronous Stored Procedure Prerequisites* on page 366
- *Specify a Nonunique Name for a User-defined Function* on page 376
- *Specify Stored Procedures and Tables for Replication* on page 372
- *Map a Function to a Different Stored Procedure Name* on page 375

Specify 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.

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.

Manage User-Defined Functions

Learn the commands for managing user-defined functions.

You can customize database operations by altering function strings for user-defined functions and you can display function-related information.

Also see *Replication Server Administration Guide Volume 1 > Manage Replication Server Security* for a list of permissions that are required to use the commands.

See also

- *Customize Database Operations* on page 13

Create 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 which are reserved.
- 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.

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.

See also

- *Create Function Strings* on page 38
- *Summary of System Functions* on page 17

Adding Parameters to a User-Defined Function

Use the **alter function** command to tell Replication Server about new parameters you add to a replicated stored procedure.

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 *Replication Server Administration Guide Volume 1 > Manage a Replication System > Quiesce Replication Server*.
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.

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
```

See also

- *Alter Function Strings* on page 40

Drop 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.

   Optionally, use the **sp_setreplicate** or **sp_setrepproc** system procedure and specify **'false'** to disable replication for the stored procedure.

Follow the guidelines when specifying stored procedures and tables and for replication. See *Replication Server Administration Guide Volume 1 > Manage Replicated Functions > Mark Stored Procedures for Replication* 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 *Replication Server Administration Guide Volume 1 > Manage a Replication System > Quiesce Replication Server*.

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 previously:

```
drop function Items_rd.Stock_receipt
```

See also

- *Specify Stored Procedures and Tables for Replication* on page 372

Map a Function to a Different Stored Procedure Name

Learn how to map a user-defined function 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.

Specify 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

Learn the background and procedures for configuring Sybase Replication Server for high availability (HA) on Sun Cluster 2.2.

Introduction to Sybase Replication for Sun Cluster HA

There are several assumptions if you want to use Sybase Replication for Sun Cluster HA.

The assumptions are:

- You are familiar with Sybase Replication Server.
- You are familiar with Sun Cluster HA.
- You have a two-node cluster hardware system 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*)

Terminology

Learn the terms discussed in Sybase Replication Server for Sun Cluster HA.

The terms used are:

- 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 solution that 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 has stopped responding, 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 is restarted on the second node.

To Replication Server clients, it appears as though the original Replication Server has restarted. 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.

Configuration of Replication Server for High Availability

Learn the tasks required to configure a Replication Server for HA on Sun Cluster (assuming a two-node cluster machine).

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 several tasks in addition to the tasks described in the Replication Server installation guide for your platform.

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

      For example enter:

      ```
      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.

      Enter:

      ```
      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 several 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/repserver_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

   Enter: **hareg -y** *repserver_name*

Administration of Replication Server as a Data Service

Learn how to start and shut down Replication Server as a data service, and learn about useful logs for monitoring and troubleshooting.

Data Service Start and Shutdown

Learn the commands to start and shut down Replication Server once a Replication Server is registered as data service.

To start Replication Server if it is not already running, and also start the fault monitor for Replication Server, enter:

```
hareg -y repserver_name
```

To shut down Replication Server, enter:

```
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 for Sun Cluster for HA

There are several logs you can use for debugging.

You can use:

- 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`.

Implement a Reference Replication Environment

You can quickly set up an Adaptive Server-to-Adaptive Server or Oracle-to-Oracle reference replication environment using the products available in your environment.

Reference Replication Environment Implementation

Replication Server includes a toolset for quickly setting up a reference implementation of Adaptive Server-to-Adaptive Server or Oracle-to-Oracle replication using the products available in your environment.

You can implement a replication environment to demonstrate Replication Server features and functionalities. Use the toolset to:

1. Build a reference environment containing Replication Server and the primary and replicate databases.
2. Configure the replication environment.
3. Perform simple transactions on the primary database and replicate the changes by database level replication.
4. Collect statistics and monitors counters from the replication processing in step 3.
5. Clean up the reference replication environment.

The reference implementation toolset consists of scripts that are in `$SYBASE/refimp`.

Note: The reference implementation builds a replication environment containing a single Replication Server, primary database server, and replicate database server. You cannot configure the reference environment topology for multiple replication system components.

Platform Support

You can implement a reference environment on all platforms that Replication Server supports except for Linux on IBM p-Series (Linux on Power) 64-bit. You must use Cygwin to run the reference implementation scripts to set up the reference environment on any Microsoft Windows platform that Replication Server supports.

See the Cygwin Web site: *http://www.cygwin.com*.

Components for Reference Implementation

You must have supported versions of the components of a replication environment before you can implement a reference environment.

Adaptive Server

You can build a reference implementation environment for Adaptive Server-to-Adaptive Server replication with the supported versions of Replication Server and Adaptive Server.

Table 33. Supported Product Component Versions for Adaptive Server Reference Implementation

Replication Server	Adaptive Server
15.5	15.0.3, 15.5

For example, you can build an Adaptive Server reference environment with Replication Server 15.5 and Adaptive Server version 15.0.3 or 15.5.

Oracle

You can also build a reference implementation environment for Oracle-to-Oracle replication with the supported versions of Replication Server, Oracle, Replication Agent for Oracle, and ECDA Option for Oracle.

Table 34. Supported Product Component Versions for Oracle Reference Implementation

Replication Server	Oracle	Replication Agent for Oracle	ECDA Option for Oracle
15.5	10.2	15.2	15.0 ESD #3

For example, you can build a reference implementation environment for Oracle with Replication Server 15.5, Oracle 10.2, Replication Agent 15.2, and ECDA Option for Oracle 15.0 ESD #3.

Prerequisites for the Reference Environment

There are several prerequisites and some information you must be aware of before you build the reference environment.

1. For Oracle, verify that you have **execute** permission in the Oracle release directory. For example, verify whether you can manually create an instance.

2. Verify that the environment variable settings in the SYBASE.sh file in the Replication Server or Adaptive Server release directory is correct. If you cannot verify this, remove or rename the file.
3. Verify that you have the UNIX **grep**, **kill**, **awk**, and **ps** commands available in your bash shell.

The reference implementation procedure uses the `interfaces` file in the Replication Server release directory. If the file exists before you run the reference implementation procedure, the procedure backs up the existing file by incrementing the file name extension.

For Oracle, the reference implementation procedure renames the existing `tnsname.ora`, `listener.ora` files and creates new files for the Oracle reference implementation.

Build the Reference Environment

Execute the **buildenv** script to automatically create a Replication Server, and the primary and replicate data servers and databases.

Enter:

```
buildenv -f config_file
```

Use *config_file* to specify the name and location of the build configuration file that contains the parameters you can specify in the file.

If **buildenv** executes successfully, you see:

```
Environment setup successfully completed.
```

Reference Implementation Configuration Files

Sybase provides configuration file templates for Adaptive Server-to-Adaptive Server, and Oracle-to-Oracle replication on supported UNIX and Microsoft Windows platforms, that you can use to create a configuration file for your environment.

The files are located in `$SYBASE/REP-15_5/REFIMP-01_0`.

Table 35. Reference Implementation Configuration Files

Primary to replicate data server and platform	Configuration file
ASE-to-ASE on UNIX	`ase_unix_refimp.cfg`
ASE-to-ASE on Windows	`ase_win_refimp.cfg`
Oracle-to-Oracle on UNIX	`ora_unix_refimp.cfg`
Oracle-to-Oracle on Windows	`ora_win_refimp.cfg`

Example of ase_unix_refimp.cfg Template File

Provide values such as directory locations and host names according to your environment.

```
#########################################################################
#######
# --- Part 1. release directory of repserver/ase/oracle/refimp ----#
#########################################################################
#######
#
# --- PLATFORM('unix': UNIX/Linux platform, 'win': Windows) ---#
#
os_platform=unix
# --- DATABASE ('ase': Adaptive Server Enterprise, 'ora': ORACLE) ---
#
#
db_type=ase
#
# --- RS RELEASE DIRECTORY --- #
#
rs release directory=/remote/repeng4/users/xiel/repserver
#
# --- RS RELEASE SUBDIRECTORY --- #
#
rs_sub_directory=REP-15_2
#
# --- ASE RELEASE DIRECTORY --- #
#
ase_release=/remote/repeng4/users/xiel/ase
#
# --- ASE/ORACLE RELEASE SUBDIRECTORY --- #
#
ase_subdir=ASE-15_0
#
# --- REFERENCE IMPLEMENTATION RELEASE DIRECTORY --- #
#
refimp_release_dir=/calm/repl/svr/refimp
#
#
#
# --- REFERENCE IMPLEMENTATION WORK DIRECTORY ---
#
refimp_work_dir=/remote/repeng4/users/xiel/test
#
# --- OCS RELEASE DIRECTORY --- #
#
ocs_release_directory=OCS-15_0
#
# --- PDS DEVICE NAME WITH FULL PATH --- #
#
pds_device_file=/remote/repeng4/users/xiel/pds
#
# --- RDS DEVICE NAME WITH FULL PATH --- #
#
rds_device_file=/remote/repeng4/users/xiel/rds
```

```
#
# --- rs_init RELEASE DIRECTORY --- #
#
rsinit_release=/remote/repeng4/users/xiel/repserver
#
#
# --- interface FILE NAME ---
#
ini_filename=interfaces
#
# --- HOST NAME ---
#
host_name=newgarlic
#########################################################################
#######
# --- Part 2. login information of replication server and data server
---#
#########################################################################
#######
#
# --- RS NAME --- #
#
rs_name=SAMPLE_RS
#
# --- RS USER NAME --- #
#
rs_username=sa
#
# --- RS PASSWORD --- #
#
rs_password=
#
#
#
# --- ERSSD NAME --- #
#
rssd_name=SAMPLE_RS_ERSSD
#
# --- ERSSD USER NAME --- #
#
rssd_username=rssd
#
# --- ERSSD PASSWORD --- #
#
rssd_password=rssd_pwd
#
# --- PDS NAME --- #
#
primary_ds=PDS
#
# --- PDB NAME ---
#primary_db=pdb
#
# --- PDB USER NAME ---
#
pdb_username=sa
```

```
#
# --- PDB PASSWORD ---
#
pdb_password=
#
# --- RDS NAME ---
#
replicate_ds=RDS
#
# --- RDB NAME ---
#
replicate_db=rdb
#
# --- RDB USER NAME ---
#
rdb_username=sa
#
# --- RDB PASSWORD ---
#
rdb_password=
#
# --- PORT FOR RS ---
#
rs_port=11754
#
# --- PORT FOR RSSD ---
#
rssd_port=11755
#
# --- PORT FOR PDS ---
#
pds_port=20000
#
# --- PORT FOR RDS ---
#
rds_port=20001
#
#########################################################################
#######
# --- Part 3. transaction profile configuration parameters  --- #
#########################################################################
#######
#
# --- number of transactions to be executed --- #
#
tran_number=100
#
# --- what kind of transction will be executed --- #
#    1."Tran_Profile_1(insert--48% delete--4% update 48%)"
#    2."Tran_Profile_2(insert--30% delete--5% update 65%)"
#    3."Tran_Profile_3(insert--61% delete--2% update 37%)"
#    4."Tran_Profile_LargeTran"
#
tran_option=1
#
#########################################################################
```

```
#######
# --- Part 4. system settings --- #
#########################################################################
#######
#
# --- WAIT TIME FOR CONNECTING SERVERS, SPECIFIED BY SECOND(S) ---
#
wait_time=10
```

Configure the Reference Environment

After you build the reference replication environment, execute the **refimp** script with the **config** parameter and a configuration file to create tables and stored procedures on the reference primary and replicate databases, and create a database replication definition and a subscription on the reference Replication Server.

Enter:

```
refimp config -f config_file
```

Use *config_file* to specify the name and location of the configuration file that contains the parameters you can specify in the file.

You must use the same configuration file information you specified for **buildenv** in the build process.

If **refimp config** executes successfully, you see:

```
Task succeeded: configuring database replication environment
completed.
```

See also

- *Objects Created for the Reference Environment* on page 396

Run Performance Tests on the Reference Environment

Execute the **refimp** script with the **run** parameter to automatically insert, update, and delete data on the primary data server using database level replication.

Enter:

```
refimp run -f config_file
```

Use *config_file* to specify the same configuration file that you use for **refimp config**.

If **refimp run** executes successfully, you see:

```
Task succeeded: insert data into primary database completed.
```

Obtain Tests Results from the Reference Environment

Execute the **refimp** script with the **analyze** parameter to collect statistics and performance information.

Enter:

```
refimp analyze -f config_file
```

Use *config_file* to specify the same configuration file that you use for **refimp config**.

If **refimp analyze** executes successfully, you see:

```
Task succeeded: fetch performance data completed.
```

Obtain the rs_ticket_history report, and the monitors and counters report from `$refimp_work_dir/report` where *refimp_work_dir* is the location you specified in the configuration file.

rs_ticket_history Report

The rs_ticket_history report describes the time that the ticket data took to pass through each Replication Server module from the time stamp reported by the ticket at each module.

The report is generated by the **rs_ticket** stored procedure. See *Replication Server Reference Manual > RSSD Stored Procedures >* ***rs_ticket***.

You can calculate the total replication duration from the times reported by a ticket at the primary and replicate databases. In the report, the columns are:

- cnt – the ticket sequence number.
- pdb_t – the time the **rs_ticket** stored procedure was executed at the primary database.
- rdb_t – the time the ticket arrived at the replicate database.
- ticket – information about the ticket, including the time that it passed through each module.

Sample rs_ticket_history Report

```
cnt       pdb_t                          rdb_t
---       -------------------            -------------------
  1       Jan 19 2010  2:17AM            Jan 19 2010  2:17AM

ticket
------------------------------------------------------------
V=2;H1=profile1;H2=start;PDB(pdb)=01/19/10 02:17:19.406;
EXEC(40)=01/19/10 02:17:19.423;B(40)=1332;
DIST(26)=01/19/10 02:17:19.669;
DSI(35)=01/19/10 02:17:19.916;
DSI_T=1;DSI_C=3;RRS=SAMPLE_RS_XIEL


cnt       pdb_t                          rdb_t
---       -------------------            -------------------
```

```
  2       Jan 19 2010  2:20AM           Jan 19 2010  2:20AM

ticket
------------------------------------------------------------

V=2;H1=profile1;H2=end;PDB(pdb)=01/19/10 02:20:32.206;
EXEC(40)=01/19/10 02:20:32.211;B(40)=5044893;
DIST(26)=01/19/10 02:20:32.249;DSI(35)=01/19/10 02:20:32.524;
DSI_T=5410;DSI_C=18297;RRS=SAMPLE_RS_XIEL
```

Monitors and Counters Report

The monitors and counters report describes the performance figures reported by Replication Server counters that monitor the commands you execute during the reporting period,

Sample Monitors and Counters Report
This is a long report; only one counter is shown.

Note: Comments to the right of the output are included to explain the example. They are not part of the output.

```
Comment: refimp
Jan 19 2010 02:17:39:606AM                          *Start time stamp*
Jan 19 2010 02:20:22:576AM                          *End time stamp*
9                                                 *No of obs intervals*
0                                                *No of min between obs*
16384                                              *SQM bytes per block*
64                                               *SQM blocks per segment*
AOBJ                                                *Module name*
10305                                               *Instance ID*
11                                                  *Instance value*
AOBJ dbo.district                                   *Module name*
AOBJ: Insert command                              *Counter external name*
AOBJInsertCommand                                  *Counter display name*
65000, , 10305, 11                                 *Counter ID, instance
ID,
                                                           instance value*
ENDOFDATA                                          *EOD for counter*

AOBJ: Update command                              *Counter external name*
AOBJUpdateCommand                                  *Counter display name*
65000, , 10305, 11                                 *Counter ID, instance
ID,
                                                           instance value*
Jan 19 2010 02:17:39:606AM, 50, 50,  1,  1       *Dump ts, obs, total,
                                                          last, max*
....
ENDOFDATA                                            *EOD for counter*
```

See also

- *Monitor Performance Using Counters* on page 273

Shut Down the Reference Implementation Servers

Execute the **cleanenv** script to shut down Replication Server and the data servers after you have cleaned up the environment.

Enter:

```
cleanenv -f config_file
```

Use *config_file* to specify the same configuration file that you use for **refimp config**.

If **cleanenv** executes successfully, you see:

```
Task succeeded: shut down all the servers.
```

Clean Up the Reference Environment

Execute the **refimp** script with the **cleanup** parameter to delete test data, and drop replication definitions, subscriptions, tables, and stored procedures to prepare for the next test.

Enter:

```
refimp cleanup -f config_file
```

Use *config_file* to specify the same configuration file that you use for **refimp config**.

If **refimp cleanup** executes successfully, you see:

```
Task succeeded: clean up database replication environment completed.
```

Objects Created for the Reference Environment

The reference implementation toolset creates stored procedure, replication definition, subscription, and table objects in the reference replication environment.

Table 36. Stored Procedures Created for Reference Implementation

Stored procedure	Location
sp_load_warehouse_data	Primary and replicate databases
sp_load_district_data	Primary and replicate databases
sp_load_customer_data	Primary and replicate databases
sp_load_history_data	Primary and replicate databases
sp_load_item_data	Primary and replicate databases
sp_load_stock_data	Primary and replicate databases

Stored procedure	Location
sp_load_order_orderline_data	Primary and replicate databases
sp_load_neworder_data	Primary and replicate databases
sp_load_data_multi_tran	Primary and replicate databases
sp_gen_neworder_data	Primary database
sp_gen_payment_data	Primary database
sp_gen_delivery_data	Primary database
sp_gen_neworder_data_large_tran	Primary database
sp_gen_payment_data_large_tran	Primary database
sp_gen_delivery_data_large_tran	Primary database
sp_generator_data_1	Primary database
sp_generator_data_2	Primary database
sp_generator_data_3	Primary database
sp_generator_data_4	Primary database

Table 37. Replication Definition and Subscription Created for Reference Implementation

Stored procedure	Subscribing for
Replication definition: **pdbrepdefforrdb**	
Subscription: **rdbsubforpdb**	**pdbrepdefforrdb**

Table 38. Tables Created for Reference Implementation

Table	Location
WAREHOUSE	Primary and replicate databases
DISTRICT	Primary and replicate databases
CUSTOMER	Primary and replicate databases
HISTORY	Primary and replicate databases
NEW_ORDER	Primary and replicate databases
ORDER	Primary and replicate databases
ORDER_LINE	Primary and replicate databases

Table	Location
ITEM	Primary and replicate databases
Stock	Primary and replicate databases

Table Schema

Table schema for the tables created for reference implementation.

Table 39. WAREHOUSE

Field name	Field definition	Comments
W_ID	2*W unique IDs	W is the warehouse number
W_NAME	Variable text, size 10	
W_STREET1	Variable text, size 20	
W_STREET2	Variable text, size 20	
W_CITY	Variable text, size 20	
W_STATE	Fixed text, size 2	
W_ZIP	Fixed text, size 9	
W_TAX	Numeric, 4 digits	Sales tax
W_YTD	Numeric, 12 digits	Year to date balance

Keys:

- Primary key: (W_ID)

Table 40. DISTRICT

Field name	Field definition	Comments
D_ID	20 unique IDs	10 are populated per warehouse
D_W_ID	2*W unique IDs	
D_NAME	Variable text, size 10	
D_STREET1	Variable text, size 20	
D_STREET2	Variable text, size 20	
D_CITY	Variable text, size 20	
D_STATE	Fixed text, size 2	

Field name	Field definition	Comments
D_ZIP	Fixed text, size 9	
D_TAX	Numeric, 4 digits	Sales tax
D_YTD	Numeric, 12 digits	Year to date balance
D_NEXT_O_ID	10, 000 unique IDs	Unique IDs for next available order number

Keys:

- Primary key (D_W_ID, D_ID)
- Foreign key (D_W_ID) references (W_ID)

Table 41. CUSTOMER

Field name	Field definition	Comments
C_ID	96, 000 unique IDs	3, 000 are populated per warehouse
C_D_ID	20 unique IDs	
C_W_ID	2*W unique IDs	
C_FIRST	Variable text, size 16	
C_MIDDLE	Fixed text, size 2	
C_LAST	Variable text, size 16	
C_STREET1	Variable text, size 20	
C_STREET2	Variable text, size 20	
C_CITY	Variable text, size 20	
C_STATE	Fixed text, size 2	
C_ZIP	Fixed text, size 9	
C_PHONE	Fixed text, size 16	
C_SINCE	Date and time	The date of registration
C_CREDIT	Fixed text, size 2	Credit: "GC"=good credit, "BC"=bad credit
C_CREDIT_LIM	Numeric, 12 digits	

Field name	Field definition	Comments
C_DISCOUNT	Numeric, 4 digits	
C_BALANCE	Signed numeric, 12 digits	
C_YTD_PAYMENT	Numeric, 12 digits	
C_PAYMENT_CNT	Numeric, 4 digits	
C_DELIVERY_CNT	Numeric, 4 digits	
C_DATA	Variable text, size 500	For remarks

Keys:

- Primary key (C_W_ID, C_D_ID, C_ID)
- Foreign key (C_W_ID, C_D_ID) references (D_W_ID, D_ID)

Table 42. HISTORY

Field name	Field definition	Comments
H_C_ID	96, 000 unique IDs	
H_C_D_ID	20 unique IDs	
H_C_W_ID	2*W unique IDs	
H_D_ID	20 unique IDs	
H_W_ID	2*W unique IDs	
H_DATE	Date and time	
H_AMOUNT	Numeric, 6 digits	
H_DATA	Variable text, size 24	

Keys:

- Primary key: None
- Foreign key (H_C_W_ID, H_C_D_ID, H_C_ID) references (C_W_ID, C_D_ID, C_ID)
- Foreign key (H_W_ID, H_D_ID) references (D_W_ID, D_ID)

Table 43. NEW_ORDER

Field name	Field definition	Comments
N_O_ID	10, 000, 000 unique IDs	

Field name	Field definition	Comments
N_D_ID	20 unique IDs	
NO_W_ID	2*W unique IDs	

Keys:

- Primary key (NO_W_ID, NO_D_ID, NO_O_ID)
- Foreign key (NO_W_ID, NO_D_ID, NO_O_ID) references (O_W_ID, O_D_ID, O_ID)

Table 44. ORDER

Field name	Field definition	Comments
O_ID	10, 000, 000 unique IDss	
O_D_ID	20 unique IDs	
O_W_ID	2*W unique IDs	
O_C_ID	96, 000 unique IDs	
O_ENTRY_D	Date and time	
O_CARRIER_ID	10 unique IDs, or null	
O_OL_CNT	From 5 to 15	
O_ALL_LOCAL	Numeric, 1digit	

Keys:

- Primary key (O_W_ID, O_D_ID, O_ID)
- Foreign key (O_W_ID, O_D_ID, O_C_ID) references (C_W_ID, C_D_ID, C_ID)

Table 45. ORDER_LINE

Field name	Field definition	Comments
OL_O_ID	10, 000, 000 unique IDs	
OL_D_ID	20 unique IDs	
OL_W_ID	2*W unique IDs	
OL_NUMBER	15 unique IDs	
OL_I_ID	200,000 unique IDs	
OL_SUPPLY_W_ID	2*W unique IDs	

Field name	Field definition	Comments
OL_DELIVERY_D	Date and time, or null	
OL_QUANTITY	Numeric, 2 digits	
OL_AMOUNT	Numeric, 6 digits	
OL_DIST_INFO	Fixed text, size 24	

Keys:

- Primary key (OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER)
- Foreign key (OL_W_ID, OL_D_ID, OL_O_ID) references (O_W_ID, O_D_ID, D_ID)
- Foreign key (OL_SUPPLY_W_ID, OL_I_ID) references (S_W_ID, S_I_ID)

Table 46. ITEM

Field name	Field definition	Comments
I_ID	200, 000 unique IDs	
I_IM_ID	200, 000 unique IDs	
I_NAME	Variable text, size 50	
I_PRICE	Numeric, 5 digits	
I_DATA	Variable text, size 50	

Keys:

- Primary key (I_ID)

Table 47. STOCK

Field name	Field definition	Comments
S_I_ID	200, 000 unique IDs	
S_W_ID	2*W unique IDs	
S_QUANTITY	Numeric, 4 digits	
S_DIST_01	Fixed text, size 24	
S_DIST_02	Fixed text, size 24	
S_DIST_03	Fixed text, size 24	
S_DIST_04	Fixed text, size 24	

Field name	Field definition	Comments
S_DIST_05	Fixed text, size 24	
S_DIST_06	Fixed text, size 24	
S_DIST_07	Fixed text, size 24	
S_DIST_08	Fixed text, size 24	
S_DIST_09	Fixed text, size 24	
S_DIST_10	Fixed text, size 24	
S_YTD	Numeric, 8 digits	
S_ORDER_CNT	Numeric, 4 digits	
S_REMOTE_CNT	Numeric, 4 digits	
S_DATA	Variable text, size 50	

Keys:

- Primary key (S_W_ID, S_I_ID)
- Foreign key (S_W_ID) references (W_ID)
- Foreign key (S_I_ID) references (I_ID)

Glossary

Glossary of terms used in replication systems.

- **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 copy-in –** A feature that improves Replication Server performance when replicating large batches of **insert** statements on the same table in Adaptive Server® Enterprise 12.0 and later. Replication Server implements bulk copy-in in Data Server Interface (DSI), the Replication Server module responsible for sending transactions to replicate databases, using the Open Client™ Open Server™ Bulk-Library.

  Bulk copy-in also improves the performance of subscription materialization. When **dsi_bulk_copy** is on, Replication Server uses bulk copy-in to materialize the subscriptions if the number of **insert** commands in each transaction exceeds **dsi_bulk_threshold**.
- **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*.
- **connection profiles –** Connection profiles allow you to configure your connection with a pre-defined set of properties.

- **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 SQL Anywhere (SA) 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)*.
- **Enterprise Connect Data Access (ECDA) –** An integrated set of software applications and connectivity tools that allow access to data within a heterogeneous database environment, such as a variety of LAN-based, non-ASE data sources, and mainframe data sources.
- **ExpressConnect for Oracle –** A set of libraries that can be used to provides direct communication between Replication Server and an Oracle database.

- **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.
- **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).
- **gateway –** Connectivity software that allows two or more computer systems with different network architectures to communicate.
- **generation number –** See *database generation number*.
- **heterogeneous data servers –** Multi-vendor data servers used together in a distributed database system.
- **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.
- **high availability (HA) –** Very low downtime. Computer systems that provide HA usually provide 99.999% availability, or roughly five minutes unscheduled downtime per year.
- **high volume adaptive replication (HVAR) –** Compilation of a group of **insert**, **delete**, and **update** operations to produce a net result and the subsequent bulk application of the net result to the replicate database.
- **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.
- **Log Transfer Manager (LTM) –** The Replication Agent program for Sybase SQL Server. See also *Replication Agent* and *RepAgent thread*.
- **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.
- **profiles –** Profiles allow you to configure your connection with a pre-defined set of properties.
- **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.
- **quoted identifiers –** Object names that contain special characters such as spaces and non-alphanumeric characters, start with a character other than an alphabet, or that correspond to a reserved word, need to be enclosed in double quote characters to be parsed correctly.
- **real time loading (RTL) –** High volume adaptive replication (HVAR) to a Sybase IQ database. Uses relevant commands and processes to apply HVAR changes to a Sybase IQ replicate database. See *high volume adaptive replication*.
- **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 SQL Anywhere (SA) 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*.
- **resync marker –** When you restart Replication Agent in resync mode, Replication Agent sends the resync database marker to Replication Server to indicate that a resynchronization effort is in progress. The resync marker is the first message Replication Agent sends before sending any SQL data definition language (DDL) or data manipulation language (DML) transactions.
- **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.
- **SQL statement replication –** In SQL statement replication, the Replication Server receives the SQL statement that modified the primary data, rather than the individual row changes from the transaction log. Replication Server applies the SQL statement to the replicated site. RepAgent sends both the SQL Data Manipulation Language (DML) and

individual row changes. Depending on your configuration, Replication Server chooses either individual row change log replication or SQL statement replication.

- **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 –** *mixed-version system*

  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.

Obtaining Help and Additional Information

Use the Sybase Getting Started CD, Product Documentation site, and online help to learn more about this product release.

- The Getting Started CD (or download) – contains release bulletins and installation guides in PDF format, and may contain other documents or updated information.
- Product Documentation at *http://sybooks.sybase.com/* – is an online version of Sybase documentation that you can access using a standard Web browser. You can browse documents online, or download them as PDFs. In addition to product documentation, the Web site also has links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, Community Forums/Newsgroups, and other resources.
- Online help in the product, if available.

To read or print PDF documents, you need Adobe Acrobat Reader, which is available as a free download from the *Adobe* Web site.

Note: A more recent release bulletin, with critical product or document information added after the product release, may be available from the Product Documentation Web site.

Technical Support

Get support for Sybase products.

If your organization has purchased a support contract for this product, then one or more of your colleagues is designated as an authorized support contact. If you have any questions, or if you need assistance during the installation process, ask a designated person to contact Sybase Technical Support or the Sybase subsidiary in your area.

Downloading Sybase EBFs and Maintenance Reports

Get EBFs and maintenance reports from the Sybase Web site.

1. Point your Web browser to *http://www.sybase.com/support*.
2. From the menu bar or the slide-out menu, under **Support**, choose **EBFs/Maintenance**.
3. If prompted, enter your MySybase user name and password.
4. (Optional) Select a filter from the **Display** drop-down list, select a time frame, and click **Go**.
5. Select a product.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as an authorized support contact. If

you have not registered, but have valid information provided by your Sybase representative or through your support contract, click **My Account** to add the "Technical Support Contact" role to your MySybase profile.

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

Sybase Product and Component Certifications

Certification reports verify Sybase product performance on a particular platform.

To find the latest information about certifications:

- For partner product certifications, go to *http://www.sybase.com/detail_list?id=9784*
- For platform certifications, go to *http://certification.sybase.com/ucr/search.do*

Creating a MySybase Profile

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

1. Go to *http://www.sybase.com/mysybase*.
2. Click **Register Now**.

Accessibility Features

Accessibility ensures access to electronic information for all users, including those with disabilities.

Documentation for Sybase products is available in an HTML version that is designed for accessibility.

Vision impaired users can navigate through the online document with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Sybase HTML documentation has been tested for compliance with accessibility requirements of Section 508 of the U.S Rehabilitation Act. 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 may 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 the Sybase Accessibility site: *http://www.sybase.com/products/accessibility*. The site includes links to information about Section 508 and W3C standards.

You may find additional information about accessibility features in the product documentation.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U