

SYBASE®

Design Guide

Replication Server®

15.2

DOCUMENT ID: DC32580-01-1520-01

LAST REVISED: February 2009

Copyright © 2009 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

Contents

About This Book	ix
CHAPTER 1	
Introduction	1
Centralized and distributed database systems.....	1
Advantages of replicating data	2
Improved performance	2
Greater data availability.....	3
Data distribution with Replication Server.....	3
Publish-and-subscribe model	4
Replicated functions	4
Transaction management.....	5
Replication system components.....	7
Replication system domain	8
Replication Server	8
ID Server	10
Replication environment.....	10
Replication Manager	11
Replication Monitoring Services	11
Data servers	11
Replication Agent	11
Client applications	12
Replication management solutions	12
Two-tier management solution	12
Three-tier management solution	13
Connecting replication system components.....	13
Interfaces file	13
Routes and connections.....	14
Master database replication	17
Non-ASE data server support	18
Enterprise Connect Data Access (ECDA)	18
Replication Agents	19
Processing data server errors	19
Functions, function strings, and function-string classes	20
Replication Server security.....	21

	Login names.....	21
	Permissions.....	22
	Network-based security.....	23
	Advanced Security option.....	24
	Summary.....	24
CHAPTER 2	Application Architecture for Replication Systems	25
	Application types	25
	Decision-support applications	26
	Distributed OLTP applications.....	28
	Remote OLTP using request functions	30
	Standby applications	31
	Effects of loose consistency on applications.....	32
	Controlling risks in high-value transactions	32
	Measuring lag time.....	33
	Methods for updating primary data	33
	Centralized primary maintenance.....	34
	Primary maintenance via network connections.....	34
	Managing update conflicts for multiple primaries	35
CHAPTER 3	Implementation Strategies.....	39
	Overview of models and strategies.....	39
	Basic primary copy model	40
	Using table replication definitions.....	41
	Using applied functions	43
	Distributed primary fragments model	47
	Replication definitions	50
	Subscriptions.....	51
	Corporate rollup	52
	Replication definitions	54
	Subscriptions.....	55
	Redistributed corporate rollup.....	56
	Warm standby applications	58
	Setting up a warm standby application.....	59
	Switching to the standby database.....	61
	Model variations and strategies	63
	Multiple replication definitions	64
	Publications.....	66
	Request functions.....	71
	Implementing master/detail relationships.....	76
CHAPTER 4	Planning for Backup and Recovery	89

	Protecting against data loss	89
	Preventive measures	90
	Standby applications	90
	Save interval.....	93
	Coordinated dumps	94
	Recovery measures	94
	Re-creating subscriptions.....	94
	Subscription reconciliation utility (rs_subcmp)	94
	Database recovery	95
	Restoring coordinated dumps	95
CHAPTER 5	Introduction to Replication Agents	97
	Replication Agent overview.....	97
	Replication Agent transaction logs.....	98
	Replication Agent products	99
	Replication Agent for DB2	99
	Sybase Replication Agent	101
CHAPTER 6	Replicating Data into Non-Adaptive Server Data Servers.....	103
	Interfacing with non-ASE data servers.....	103
	Sybase database gateway products	104
	Maintenance user.....	105
	Function-string class	105
	Creating function-string classes using inheritance	106
	Creating distinct function-string classes	106
	Error class	107
	rs_lastcommit table	108
	rs_get_lastcommit function	110
CHAPTER 7	International Replication Design Considerations	111
	Designing an international replication system.....	111
	Message language	112
	Character sets	113
	Character-set conversion	113
	Unicode UTF-8 and UTF-16 support.....	114
	Guidelines for using character sets	115
	Sort order	116
	Subscriptions.....	116
	Unicode sort order.....	120
	Changing the character set and sort order.....	121
	When changing the character set changes the character width	124
	Summary.....	125

APPENDIX A	Capacity Planning	127
	Overview of requirements	127
	Replication Server requirements	127
	Replication Server requirements for primary databases	128
	Replication Server requirements for replicate databases.....	129
	Replication Server requirement for routes.....	129
	Data volume (queue disk space requirements).....	129
	Overview of disk queue size calculation.....	130
	Message sizes.....	131
	Change rate (number of messages).....	134
	Change volume (number of bytes).....	134
	Calculating table volume	135
	Overall queue disk usage.....	141
	Additional considerations	141
	Example queue usage calculations.....	142
	Message size example calculations	143
	Change rate.....	143
	Table volume example calculations	144
	Inbound database volume	144
	Inbound queue size example calculation	145
	Other disk space requirements	147
	Stable queues	148
	RSSD	148
	ERSSD	148
	Logs.....	148
	Memory usage	149
	Replication Server memory requirements	149
	RepAgent memory requirements	149
	CPU usage.....	151
	Enabling SMP.....	152
	Network requirements	152
APPENDIX B	Log Transfer Language.....	153
	Log Transfer Language overview.....	153
	connect source.....	154
	Keywords.....	155
	Upgrade locator.....	156
	Example of connect source	156
	get maintenance user.....	157
	get truncation	158
	Format of the origin queue ID.....	158
	distribute.....	159
	Command tags	160
	Transaction-control subcommands	161

applied subcommand	163
execute subcommand	171
sqlddl append subcommand	173
dump subcommand.....	174
purge subcommand.....	175
Sample RepAgent session.....	175
Index	177

About This Book

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

This book introduces distributed database systems built upon replication technology and helps you design a replication system.

Audience

The *Replication Server Design Guide* is for everyone who uses Replication Server. If you are new to Replication Server, begin with this book for an introduction to Replication Server and the applications that use replicated data.

If you are designing a new application for Replication Server, you should read this book before you install the Replication Server software. Use the information in this book to plan your replication system so that you will know where to install the software components that make up your replication system.

How to use this book

The information in this book is organized as follows:

- Chapter 1, “Introduction” introduces Replication Server and its features.
- Chapter 2, “Application Architecture for Replication Systems” discusses replication system design issues.
- Chapter 3, “Implementation Strategies” describes models for implementing your replication system design.
- Chapter 4, “Planning for Backup and Recovery” describes the preventive and corrective measures you can use to recover from replication system failures.
- Chapter 5, “Introduction to Replication Agents” describes Sybase® Replication Agent™ products that you can use to replicate data from a database that is not an Adaptive Server® Enterprise database.

-
- Chapter 6, “Replicating Data into Non-Adaptive Server Data Servers” describes the replication system components that you need to replicate data into a data server other than Adaptive Server.
 - Chapter 7, “International Replication Design Considerations” describes how to configure languages, character sets, and sort orders for an international environment.
 - Appendix A, “Capacity Planning” explains how to estimate the amount of disk space needed for Replication Server partitions.
 - Appendix B, “Log Transfer Language” describes the Log Transfer Language (LTL) used by Replication Agents to send transaction operation and stored procedure invocation data to a Replication Server.

Related documents

The Sybase Replication Server documentation set consists of:

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

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

- *Installation Guide* for your platform – describes installation and upgrade procedures for all Replication Server and related products.
- *Configuration Guide* for your platform – describes configuration procedures for all Replication Server and related products, and explains how to use the `rs_init` configuration utility.
- *Getting Started with Replication Server* – provides step-by-step instructions for installing and setting up a simple replication system.
- *New Features Guide* – describes the new features in Replication Server.
- *Administration Guide* – contains an introduction to replication systems. This manual includes information and guidelines for creating and managing a replication system, setting up security, recovering from system failures, and improving performance.
- *Design Guide* (this book) – contains information about designing a replication system and integrating heterogeneous data servers into a replication system.
- *Heterogeneous Replication Guide* and the Replication Server Options documentation set – describes how to use Replication Server to replicate data between databases supplied by different vendors.

- *Reference Manual* – contains the syntax and detailed descriptions of Replication Server commands in the Replication Command Language (RCL); Replication Server system functions; Sybase Adaptive Server® commands, system procedures, and stored procedures used with Replication Server; Replication Server executable programs; and Replication Server system tables.
- *Troubleshooting Guide* – contains information to aid in diagnosing and correcting problems in the replication system.
- *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Available only in print version.
- Replication Manager plug-in help, which contains information about using Sybase Central™ to manage Replication Server.

Other sources of information

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

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

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

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

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

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

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

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

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click Certification Report.
- 3 In the Certification Report filter select a product, platform, and timeframe and then click Go.
- 4 Click a Certification Report title to display the report.

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

- 1 Point your Web browser to Availability and Certification Reports at <http://certification.sybase.com/>.
- 2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
- 3 Select Search to display the availability and certification report for the selection.

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

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

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

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

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
- 3 Select a product.
- 4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

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

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

Conventions

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

Style conventions Syntax statements (displaying the syntax and options for a command) are printed as follows:

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

See “Syntax conventions” on page xiv for more information.

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

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

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

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

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

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

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

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

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

`rs_default_function_class` is a default function-string class.

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

Table 1: Syntax formatting conventions

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

Obligatory choices

- Curly braces and vertical bars – choose only one option.
`{red | yellow | blue}`
- Curly braces and commas – choose one or more options. If you choose more than one, separate your choices with commas.
`{cash, check, credit}`

Optional choices

- One item in square brackets – choose it or omit it.
`[anchovies]`
- Square brackets and vertical bars – choose none or only one.
`[beans | rice | sweet_potatoes]`
- Square brackets and commas – choose none, one, or more options. If you choose more than one, separate your choices with commas.
`[extra_cheese, avocados, sour_cream]`

Repeating elements

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

```
alter function replication definition function_rep_def
{deliver as 'proc_name' |
```

```

add @parameter datatype [, @parameter
datatype]... |
add searchable parameters @parameter
[, @parameter]... |
send standby {all | replication definition}
parameters)

```

RCL command formatting

RCL commands are similar to Transact-SQL® commands. The following sections present the formatting rules.

Command format and command batches

- You can break a line anywhere except in the middle of a keyword or identifier. You can continue a character string on the next line by typing a backslash (\) at the end of the line.
- Extra space characters on a line are ignored, except after a backslash. Do not enter any spaces after a backslash.
- You can enter more than one command in a batch, unless otherwise noted.
- RCL commands are not transactional. Replication Server executes each command in a batch without regard for the completion status of other commands in the batch. Syntax errors in a command prevent Replication Server from parsing subsequent commands in a batch.

Case sensitivity

- Keywords in RCL commands are not case-sensitive. You can enter them with any combination of uppercase or lowercase letters.
- 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.

Identifiers

Identifiers are names you give to servers, databases, variables, parameters, database objects, and replication objects. Database object names include names for tables, columns, and views. Replication object names include names for replication definitions, subscriptions, functions, and publications.

- Identifiers can be 1 – 255 bytes long (equivalent to 1 – 255 single-byte characters) and must begin with a letter, the @ sign, or the _ character. See the *Replication Server Reference Manual* for a list of identifiers that have been extended to 255 bytes.

Parameters in function strings

- Replication Server function parameters are the only identifiers that can begin with the @ character. Function parameter names can include 255 characters *after* the @ character.
- After the first character, identifiers can include letters, digits, and the #, \$, or _ characters. Spaces are not allowed.
- Parameters in function strings have the same rules as identifiers, except:
 - They are enclosed in question marks (?), allowing Replication Server to locate them in the function string. Use two consecutive question marks (??) to represent a literal question mark in a function string.
 - The exclamation point (!) introduces a parameter modifier that indicates the source of the data that will be substituted for a parameter at runtime. Refer to the *Replication Server Reference Manual* for a complete list of modifiers.

Data support Replication Server supports all Adaptive Server datatypes.



User-defined datatypes are not supported. The double precision, nchar, and nvarchar datatypes are indirectly supported by mapping them to other datatypes.




For more information about the supported datatypes, including how to format them, see the *Replication Server Reference Manual*.

Replication Server supports a set of datatype definitions for non-Sybase data servers that lets you replicate column values of one datatype to a column of a different datatype in the replicate database. See the *Replication Server Administration Guide Volume 1* for more information about heterogeneous datatype support (HDS).

Icons

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

	Description
	This icon represents Replication Server, the Sybase server program maintains replicated data on a local-area network (LAN) and processes data transactions received from other Replication Servers on wide-area network (WAN).
	This icon represents Adaptive Server, the Sybase data server. Data servers manage databases containing primary or replicated data. Replication Server also works with heterogeneous data servers, so, unless otherwise noted, this icon can represent any data server in a replication system.

	Description
	<p>This icon represents Replication Agent, a replication system process or module that transfers transaction log information for primary database to a Replication Server. The Replication Agent for Adaptive Server is RepAgent. Sybase provides Replication Agent products for Adaptive Server™ Anywhere, DB2, Microsoft SQL Server, and Oracle data servers.</p> <p>Except for RepAgent, which is an Adaptive Server thread, all Replication Agents are separate processes. In general, this icon only appears when representing a Replication Agent that is a separate process.</p>
	<p>This icon represents client application. A client application is a user process or application connected to a data server. It may be a front-end application program executed by a user or a program that executes as an extension of the system.</p>
	<p>This icon represents the Sybase Central Replication Manager plug-in (RM), a management utility that lets a replication system administrator develop, manage, and monitor a Sybase Replication Server environment.</p>

Accessibility features

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

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

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

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

If you need help

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

Introduction

This chapter introduces Replication Server and describes how it is used to create and maintain distributed data applications.

Topic	Page
Centralized and distributed database systems	1
Advantages of replicating data	2
Data distribution with Replication Server	3
Replication system components	7
Replication management solutions	12
Connecting replication system components	13
Master database replication	17
Non-ASE data server support	18
Replication Server security	21
Summary	24

Centralized and distributed database systems

In the traditional enterprise computing model, an Information Systems department maintains control of a centralized corporate database system. Mainframe computers, usually located at corporate headquarters, provide the required performance levels. Remote sites access the corporate database through wide-area networks (WANs) using applications provided by the Information Systems department.

Changes in the corporate environment toward decentralized operations have prompted organizations to move toward distributed database systems that complement the new decentralized organization.

Today's global enterprise may have many local-area networks (LANs) joined with a WAN, as well as additional data servers and applications on the LANs. Client applications at the sites need to access data locally through the LAN or remotely through the WAN. For example, a client in Tokyo might locally access a table stored on the Tokyo data server or remotely access a table stored on the New York data server.

In a distributed database environment, mainframe computers may be needed at corporate or regional headquarters to maintain sensitive corporate data, while clients at remote sites use minicomputers and server-class workstations for local processing.

Both centralized and distributed database systems must deal with the problems associated with remote access:

- Network response slows when WAN traffic is heavy. For example, a mission-critical transaction-processing application may be adversely affected when a decision-support application requests a large number of rows.
- A centralized data server can become a bottleneck as a large user community contends for data server access.
- Data is unavailable when a failure occurs on the network.

Advantages of replicating data

The performance and availability problems associated with remote database access can be solved by replicating the data from its source database to a local database. Replication Server provides a cost-effective, fault-tolerant system for replicating data.

Replication Server keeps data up to date in multiple databases so that clients can access local data instead of remote, centralized databases. Compared to a centralized data system, a replication system provides improved system performance and data availability and reduces communication overhead.

Because it transfers transactions, not rows, Replication Server maintains the integrity of replicated data across the system, while also increasing data availability. Replication Server also allows you to replicate stored procedure invocations, further enhancing performance.

Improved performance

In a distributed replication system, data requests are completed on the local data server without the client having to access the WAN. Performance for local clients is improved because:

- LAN data transfer rates are faster than WAN data transfer rates.

- Local clients share local data server resources instead of competing for central data server resources.
- Traffic and contention for locks are reduced considerably because local decision-support applications are separated from centralized OLTP applications.

Greater data availability

In a distributed replication system, data is replicated at local and remote sites, so clients can continue to work regardless of what happens at the primary data source or over the WAN.

- When a failure occurs at a remote site, clients can continue to use local copies of replicated data.
- When a WAN failure occurs, clients can continue to use local replicated data.
- When the local data server fails, clients can switch to replicated data at another site.

When WAN communications fail, Replication Servers at other sites store transactions in stable queues (disk storage) so that replicated tables at the unavailable site can be brought up to date when communications resume. When a replicated function is initiated in a source database, it is stored in stable queues until it can be delivered to the destination site.

Data distribution with Replication Server

Replication Server works to distribute data over a network by:

- Providing application developers and system administrators with a flexible publish-and-subscribe model for marking data and stored procedures to be replicated
- Managing replicated transactions while retaining transaction integrity across the network

Because Replication Server replicates transactions—incremental changes instead of data copies—and stored procedure invocations, not the stored procedures themselves, it provides a high-performance distributed data environment while maintaining data integrity.

Publish-and-subscribe model

In a functioning Replication Server system, transactions occurring in a source database are detected by a Replication Agent and transferred to the local Replication Server, which distributes the information across LANs and WANs to Replication Servers at destination sites. These Replication Servers in turn update the target database according to the requirements of the remote client. If a network or system component fails, data in the process of being delivered is temporarily stored in queues. When the failed component returns to operation, the replication system resynchronizes copies of the data and normal replication resumes.

The primary data is the source of the data that Replication Server replicates in other databases. You “publish” data at primary sites to which Replication Servers at other sites “subscribe.” You first create a replication definition to designate the location of the primary data. The replication definition describes the structure of the table and names the database that contains the primary copy of the table. For easier management, you may collect replication definitions into publications.

The creation of a replication definition or publication does not, by itself, cause Replication Server to replicate data. You must create a subscription against the replication definition (or the publication) to instruct Replication Server to replicate the data in another database. A subscription resembles a SQL select statement. It can include a where clause to specify which rows of a table you want to replicate in the local database, allowing you to replicate only the necessary data.

Beginning with the 11.5 version of Replication Server, you can have multiple replication definitions for a primary table. Replicate tables can subscribe to different replication definitions to obtain different views of the data.

Once you have created subscriptions to replication definitions or publications, Replication Server replicates transactions to databases with subscriptions for the data.

Replicated functions

Replication Server lets you replicate Adaptive Server stored procedure invocations asynchronously between databases. This method can improve performance over normal data replication by encapsulating many changes in a single replicated function. Because they are not associated with table replication definitions, replicated functions can execute stored procedures that may or may not modify data directly.

You can replicate stored procedure invocations from a primary database to a replicate database, or from a replicate database to a primary database. See “Using applied functions” on page 43 and “Request functions” on page 71 for details.

With replicated functions, you can execute a stored procedure in another database. A replicated function allows you to:

- Replicate the execution of an Adaptive Server stored procedure to subscribing sites
- Improve performance by replicating only the name and parameters of the stored procedure rather than the actual changes

Like tables, replicated stored procedures may have replication definitions, which are called *function replication definitions*, and subscriptions. When a replicated stored procedure executes, the Replication Server passes its name and execution parameters to subscribing sites, where the corresponding stored procedure executes.

You create function replication definitions at the primary data site. Replication Server supports applied functions and request functions:

- An *applied function* is replicated from a primary to a replicate database. You create subscriptions at replicate sites for the function replication definition and mark the stored procedure for replication in the primary database. The applied function is applied at replicate database by `maint_user`.
- A *request function* is replicated from a primary to a replicate database. You create subscriptions at replicate sites for the function replication definition and mark the stored procedure for replication in the primary database. The request function is applied at replicate database by the same user who executes the stored procedure at the primary database.

Transaction management

Replication Server depends on data servers to provide the transaction processing services needed to protect their stored data. To guarantee the integrity of distributed data, data servers must comply with such transaction-processing conventions as atomicity and consistency.

Data servers that store primary data provide most of the concurrency control needed for the distributed database system. If a transaction fails to update a table with primary data, Replication Server does not distribute the transaction to other sites. When a transaction does update primary data, Replication Server distributes the changes and, unless a failure occurs, the update succeeds at all sites that have subscribed to the data.

Replication Server uses *optimistic concurrency control* to maintain replicated data consistency. This method differs from a *pessimistic distributed concurrency control* method—such as the two-phase commit—because it processes failures after they occur.

Optimistic concurrency control has these advantages in a replication system:

- It promotes high availability of data because it does not lock the data for the duration of the distributed transaction.
- It requires fewer system resources to process a transaction.
- It does not require data servers to have special distributed transaction processing features in order to participate in a distributed transaction.

Failed replicated transactions

A modification to primary data may fail to update a replicate copy of the data at another site. The primary version is the “official” copy, and updates that succeed at the primary database are expected to succeed at sites with replicate copies.

Some reasons for the failure of an update to a replicated table are:

- The data server’s maintenance user login name does not have the permissions needed to update the replicate data.
- The replicate and primary versions of the data are inconsistent after a system recovery.
- A client updates replicate data directly rather than updating the primary version.
- The data server storing the replicate table has constraints that are not enforced by the data server storing the primary version.
- The data server storing the replicated copy of the table rejects the transaction due to a system failure, such as lack of space in the database.

When a transaction fails, Replication Server receives an error from the data server. Data server errors are mapped to Replication Server error actions. The default action for a failed transaction is to write a message in the Replication Server error log (including the message returned by the data server) and then suspend the database connection. After you correct the cause of the failure, you can resume the database connection and Replication Server will retry the failed transaction.

You also can have Replication Server record a failed transaction in the exceptions log (a set of three tables in the RSSD) and continue processing the next transaction. Refer to “Replication Server” on page 8 for a description of the RSSD.

If you use the exceptions log, you must manually resolve the transactions that are saved there to make the replicate data consistent with the primary data. In some cases, the process can be automatic by encapsulating the logic for handling the rejected transactions in an intelligent application program.

Transactions that modify data in multiple data servers and databases

A transaction that modifies primary data in more than one data server may require additional concurrency control. According to the transaction processing requirements, either all of the operations in the transaction are performed or none of them are performed. If a transaction fails on one data server, it must be rolled back on all other data servers updated in the transaction.

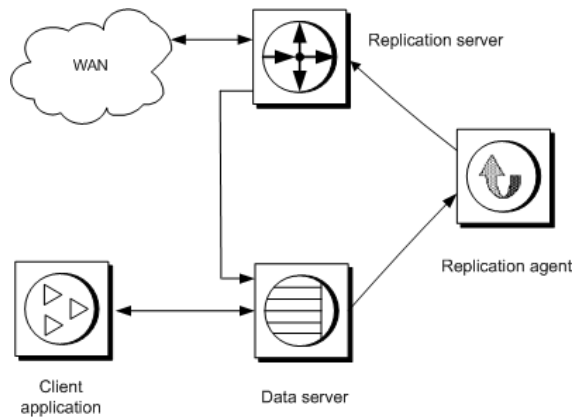
Normally, there is exactly one Replication Agent for each primary database. If a single transaction updates multiple primary databases, that transaction is replicated as multiple independent transactions, one for each primary database. Or, you can select to encapsulate such transactions in a single stored procedure, which then flows as an atomic unit to subscribing sites.

Replication system components

Replication Server has an open architecture that allows you to build a replication system from existing systems and applications and add to it as your organization grows and changes.

Figure 1-1 is a simplified depiction of one replication system site in a WAN-based, distributed database system that uses Replication Server. The sections that follow describe each component.

Figure 1-1: Replication system



Replication system domain

Replication system domain refers to all replication system components that use the same ID Server. You can set up multiple replication system domains, with the following restrictions:

- Replication Servers in different domains cannot exchange data. Each domain must be treated as a separate replication system with no cross-communication between them. You cannot create a route between Replication Servers in different domains.
- A database can be managed by only one Replication Server in one domain. Any given database is in one, and only one, ID Server's domain. This means you cannot create multiple connections to the same database from different domains.

Replication Server

A Replication Server at each site coordinates the data replication activities for the local data servers, and exchanges data with Replication Servers at other sites.

A Replication Server:

- Receives primary data transactions from databases via Replication Agents and distributes them to sites with subscriptions for the data
- Receives transactions from other Replication Servers and applies them to local databases

Replication Server system tables store information needed to accomplish these tasks. The system tables include descriptions of the replicated data and replication objects such as replication definitions and subscriptions, security records for Replication Server users, routing information for other sites, access methods for the local databases, and other administrative information.

Replication Server system tables are stored in an Adaptive Server database called the Replication Server System Database (RSSD), or an SQL Anywhere® (SA) database called the Embedded Replication Server System Database (ERSSD). An RSSD or ERSSD is assigned to each Replication Server. An Adaptive Server data server with an RSSD can also store application databases. For more information, see the *Replication Server Administration Guide Volume 1*.

Use Replication Command Language (RCL) or the Replication Manager plug-in of Sybase Central to manage information in Replication Server. You can execute RCL commands, which resemble SQL commands, on Replication Server using isql, the Sybase interactive SQL utility. The *Replication Server Reference Manual* is the complete reference for RCL. You can find information about Replication Manager and Replication Monitoring Services in the *Replication Server Administration Guide Volume 1* and in the online help for Replication Manager.

Partitions and stable queues

Replication Server stores messages on disk to make sure that they can be delivered following a failure. When you install a Replication Server, you allocate an initial *disk partition* that Replication Server uses for its disk storage. You can add additional partitions when you have finished installing the Replication Server.

The partition is either a raw disk device or operating system file. Because UNIX operating systems buffer file I/O, you may not be able to completely recover data following a failure. On such a system, use operating system files for partitions only in a test environment. Use raw disk partitions for production environments. See the *Replication Server Reference Manual* for more information about adding partitions.

Replication Server allocates *stable queues* from its disk partitions for the routes and connections it serves. Messages are saved in the stable queues at least until the messages are confirmed as received at their destination.

The amount of disk space you should allocate for Replication Server partitions depends on the size of transactions and the transaction rate for your application. Stable queues act as buffers for data as it flows through your replication system. If a remote site's Replication Server is unreachable during a network failure, the primary Replication Server stores transactions in a stable queue until communication is restored. The more space allocated for disk partitions, the longer the Replication Server can queue data without interrupting operations in the primary database.

Appendix A, "Capacity Planning" explains in detail how to calculate partition space for a Replication Server.

ID Server

The ID Server is a Replication Server that registers all Replication Servers and databases in the replication system. The ID Server must be running each time a:

- Replication Server is installed
- Route is created
- Database connection is created or dropped

Because of this requirement, *the ID Server is the first Replication Server that you start when you install a replication system.*

The ID Server must have a login name for Replication Servers to use when they connect to the ID Server. The login name is recorded in the configuration files of all Replication Servers in the replication system by the `rs_init` configuration program.

Replication environment

A replication environment consists of a set of servers that participate in replication. This includes the data servers, Replication Agents, Replication Servers, and DirectConnect™ servers. A replication environment does not have to contain all servers in a replication system domain.

Replication Manager

The Replication Manager (RM) is installed as a plug-in to Sybase Central. RM is a management utility for developing, managing, and monitoring replication environments. See the *Replication Server Administration Guide Volume 1* for more information on using RM.

Replication Monitoring Services

The Replication Monitoring Services (RMS) acts as the middle tier in a three-tier management solution for a replication environment. RMS monitors the health of the servers and components in the replication environment and provides information to troubleshoot problems and commands to fix the problems. See the *Replication Server Administration Guide Volume 1* for more information on using RMS.

Data servers

Data servers manage databases containing primary or replicated data. Clients use them to store and retrieve data and to process queries and transactions. Replication Server maintains replicated data in data servers by logging in as a database user.

Replication Server supports heterogeneous data servers through an open interface. Any system for storing data can be used as a data server if it supports a set of required data operations and transaction processing directives.

See “Non-ASE data server support” on page 18 for more information on data server requirements.

Replication Agent

A Replication Agent transfers transaction log information, which represents changes made to primary data, from a data server to a Replication Server for distribution to other databases. The Replication Agent for Adaptive Server is RepAgent, which is an Adaptive Server thread.

The Replication Agent reads the database transaction log and transfers log records for replicated tables and replicated stored procedures to the Replication Server that manages the database. The Replication Server reconstructs the transaction and forwards it to the sites that have subscriptions for the data.

A Replication Agent is needed for each database that contains primary data or for each database where replicated stored procedures are executed. A database that contains only copies of replicated data and has no replicated stored procedures does not require a Replication Agent.

Because RepAgent is an Adaptive Server thread, most system diagrams in this book do not include a Replication Agent icon. In Chapter 5, “Introduction to Replication Agents,” however, the Replication Agent described is a separate process and, for clarity, system diagrams contain the Replication Agent icon.

Client applications

A client application is a user program that accesses a data server. When the data server is an Adaptive Server, applications can be programs created with Open Client/Server™, Embedded SQL™, PowerBuilder®, or any other front-end development tool compatible with Sybase Client/Server Interfaces™ (C/SI).

Client applications should update only the primary data. Replication Server distributes the changes to the other sites. Client applications that do not modify data do not need to distinguish between primary and replicated data.

Replication management solutions

Replication Server offers two management solutions to support different replication environments

Two-tier management solution

In a two-tier management solution, RM manages the replication environment by connecting directly to servers in the environment without communicating through a management layer.

This two-tier management solution lets you manage small, simple replication environments with fewer than ten servers. You can create, alter, and delete components in the replication environment. In addition to managing the replication environment, RM also lets you monitor the status of the servers and replication components in the replication environment.

Three-tier management solution

In a three-tier management solution, RM can manage larger and complex replication environments with the help of RMS. RM connects to the servers in the environment through RMS.

RMS provides the monitoring capabilities for the replication environment. In this three-tier management solution, RMS monitors the status of the servers and other components in the replication environment, and RM provides the client interface that displays the information provided by RMS.

Connecting replication system components

Replication Server, Replication Agent, and Adaptive Server use C/SI to communicate over a network. In addition, Replication Server uses routes and connections to send messages to other Replication Servers and databases. The following sections describe the interfaces file, routes, and connections.

Interfaces file

Server programs such as data servers, Replication Servers, and Replication Agents are registered in an interfaces file (*sql.ini* in Windows and *interfaces* in UNIX) or a Lightweight Directory Access Protocol (LDAP) server so that client applications and other server programs can locate them.

Generally, one interfaces file at each site contains entries for all of the local and remote Replication Servers and data servers. The entry for each server includes its unique name and the network information that other servers and client programs need to connect to it.

Use a text editor to maintain your interfaces file. For information about LDAP servers, see the *Replication Server Administration Guide Volume 1*.

Note If you are using network-based security, available with Replication Server version 12.0 and later, use the directory services of your network security mechanism (rather than the interfaces file) to register Replication Servers, Adaptive Servers, and gateway software. Refer to the documentation that comes with your network-based security mechanism for details.

Routes and connections

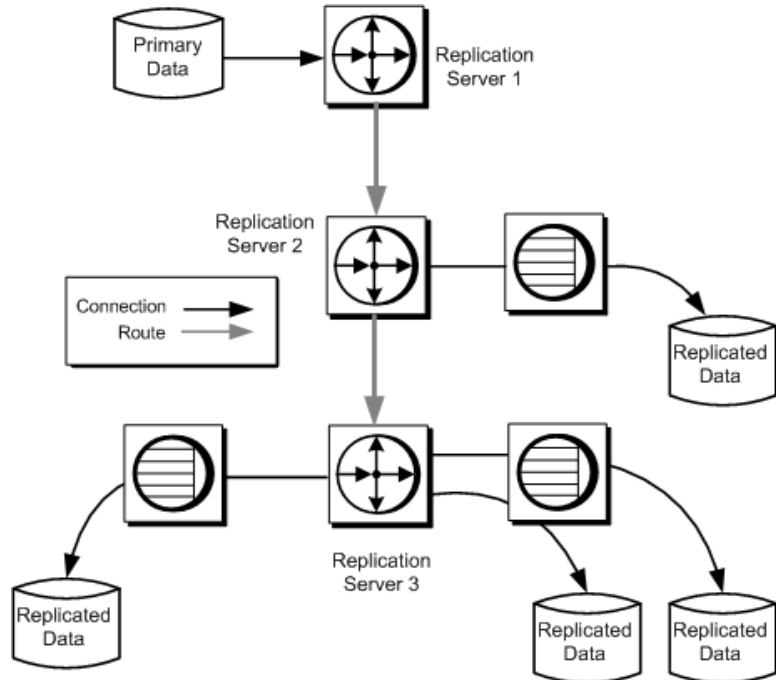
Routes and connections allow Replication Servers to send messages to each other and to send commands to databases. A *route* is a one-way message stream that sends requests from one Replication Server to another Replication Server. A *connection* is a message stream from a Replication Server to a database. Replication Server uses a *logical connection* to represent the active and standby databases in a warm standby application.

To replicate data from one database into another, you must first establish the routes and connections that allow Replication Server to move the data from its source to its destination.

When you add a database to your replication system, Sybase Central or `rs_init` creates the connection for you. You never have to create a connection directly unless you are replicating data into a database that is not an Adaptive Server.

If you have more than one Replication Server in your replication system, you must create routes between them. If you have only one Replication Server, you do not need to create routes.

Figure 1-2 illustrates connections and routes between three Replication Servers, one database storing primary data, and four databases storing replicated data.

Figure 1-2: Routes and connections

When you create a route from a primary Replication Server to a replicate Replication Server, transactions flow from the primary server to the replicate server.

If you plan to execute replicated stored procedures in a replicate database to update a primary database, you must also create a route from the replicate Replication Server to the primary Replication Server.

Direct and indirect routes

In a replication system with one primary Replication Server and many replicate Replication Servers, you can use indirect routes to reduce the load on the primary Replication Server. Indirect routes allow Replication Server to send messages to multiple destinations through a single intermediate Replication Server.

Routes with intermediate sites have important advantages:

- Reduced WAN volume

Replication Server distributes one copy of a message to each intermediate site. Replication Servers at the intermediate sites duplicate the messages for each of their outgoing queues.

- Reduced Replication Server load

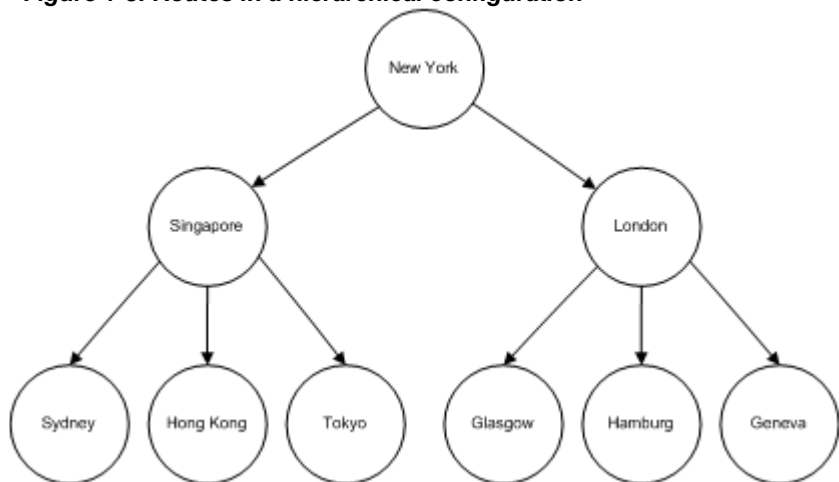
Additional Replication Servers running on separate computers share the processing load, reducing the amount of processing required of Replication Servers at primary sites.

- Fault tolerance

Messages stored at intermediate sites can be used to recover from partition failures at remote sites. See the *Replication Server Administration Guide Volume 1* for details.

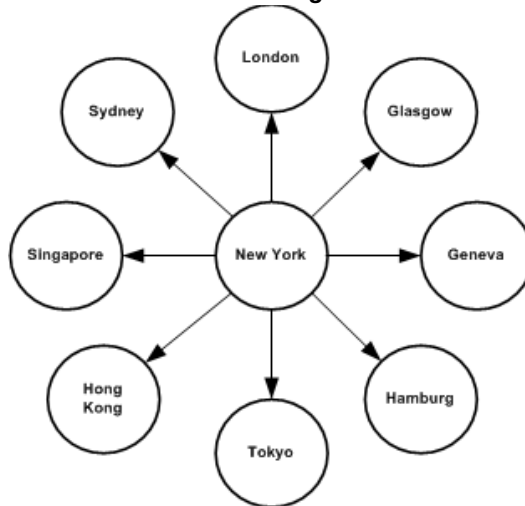
Figure 1-3 shows how message distribution is handled using intermediate sites. The message follows a direct route to the intermediate sites. From the intermediate site, it follows a direct route to the local site. With this routing arrangement, the primary site sends two messages rather than eight.

Figure 1-3: Routes in a hierarchical configuration



Intermediate sites reduce primary site message volume, but they increase the time between updates at the primary and replicate servers. Plan your routes carefully; use only the number of intermediate sites required.

If intermediate sites are not used, routes are set up in a star configuration, as Figure 1-4 illustrates.

Figure 1-4: Direct routes in a star configuration

When a row is updated at a primary site, the primary Replication Server sends messages through the WAN to each remote site that has a subscription for the row. For example, in Figure 1-4, New York can send identical data through eight different routes. If there are many sites, the network is quickly overloaded with redundant messages.

Creating routes in a hierarchical arrangement allows load balancing by reducing the number of connections and messages distributed from the primary site. Additional Replication Servers running on separate computers share the processing load, reducing the amount of processing required of Replication Servers at primary sites.

Master database replication

The master database controls the operation of Adaptive Server and stores information about every user database and associated database devices. You can replicate the master database, although only the DDL and system commands used to manage logins and roles are replicated. Master database replication does not replicate data from system tables, data or procedures from any other user tables in the master database.

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

For a list of supported DDL and system procedures that apply to master database, see the *Replication Server Administration Guide Volume 2*.

Replication Server 12.0 and later supports master database replication with warm standby, and with MSA in Replication Server 12.6 and later. The primary or active Adaptive Server must be Adaptive Server 15.0 ESD #2 and later.

See the *Replication Server Administration Guide Volume 1* for information about master database replication in MSA, and the *Replication Server Administration Guide Volume 2* for information about master database replication in a warm standby environment.

Non-ASE data server support

Support for Adaptive Server is built into Replication Server. The open architecture of Replication Server also supports non-ASE data servers in replication systems. This section contains a brief overview of open architecture components. See the *Replication Server Heterogeneous Replication Guide* and Chapter 6, “Replicating Data into Non-Adaptive Server Data Servers” for more information.

The open architecture includes:

- Sybase Enterprise Connect™ Data Access
- Replication Agents
- Error classes and error processing actions
- Functions, function strings, and function-string classes
- User defined datatypes (UDD) and datatype translations
- Connection profiles

Enterprise Connect Data Access (ECDA)

ECDA is an integrated set of software applications and connectivity tools that allows you to access data within a heterogeneous database environment. ECDA gives you the ability to access a variety of LAN-based, non-Sybase data sources, as well as mainframe data sources. It consists of components that provide transparent data access within an enterprise. You require a specific ECDA component for each actively supported non-ASE database. See the *Replication Server Options Overview Guide*.

Replication Agents

A Replication Agent is required for every database that stores primary data or initiates replicated functions. A Replication Agent reads the data server transaction log to detect changes to primary data and executions of replicated stored procedures. The transaction log provides a reliable source of information about primary data modifications because it contains records of committed, recoverable transactions.

Chapter 5, “Introduction to Replication Agents” describes the Replication Agents for data servers other than Adaptive Server. Replication Agents for some non-ASE data servers—such as DB2, Microsoft SQL Server, and Oracle—are available from Sybase.

Processing data server errors

Replication Server processes the errors and status codes returned by data servers according to your instructions. Each vendor’s data server has a different set of error codes. Replication Command Language (RCL) commands allow you to:

- Create an error class to group together the error code mappings for a database.
- Assign error actions, such as `warn`, `retry_log`, and `stop_replication`, to data server error codes.
- Associate an error class with a database.

Note Replication Server 15.2 and later includes for actively supported databases, pre-loaded error classes with associated error actions. See “Connection profiles,” in Chapter 7, “Managing Database Connections” in the *Replication Server Administration Guide Volume 1*.

See “Error class” on page 107 for more information.

Functions, function strings, and function-string classes

In order to operate in a heterogeneous database environment, Replication Server differentiates database commands from the functions it uses to distribute data server requests to other sites. A *function* is a Replication Server object that represents a data server operation such as insert, delete, and begin transaction. Replication Server uses function strings to convert functions into data-server-specific commands. A *function string* is a template that Replication Server uses to generate a command the data server can interpret as a transaction-control directive or data-modification instruction.

A *function-string class* is the set of all function strings used with a database. Function-string classes are provided for Adaptive Server and DB2 data servers. Function strings for transaction control directives are defined just once for each function-string class. Function strings to insert a row, delete a row, or update a row are defined once for each replicated table in a database.

A function string can contain variables—identifiers enclosed in question marks (?)—that represent the values of columns, procedure parameters, system-defined information, and user-defined variables. Replication Server replaces the variables with actual values before sending the function strings to the data server.

Function strings can be used to generate either RPC or database commands such as SQL statements, depending on their format. An RPC-formatted function string contains a remote procedure call followed by a list of data parameters. Embedded variables can be used to assign runtime values to the parameters. Replication Server interprets RPC function strings, builds a remote procedure call, and replaces the variables with runtime data values. An RPC can execute a registered procedure in an Open Server™ gateway to a data server or a stored procedure in an Adaptive Server.

A language-formatted function string passes a database command to the data server. Replication Server does not attempt to interpret the string, except to replace embedded variables with runtime data values. For example, several relational database servers use the SQL database language. A SQL command should be represented as a language function string.

RPC function strings can be more efficient than language function strings because the network packets sent from Replication Server are more compact.

Note Replication Server 15.2 and later includes function string classes pre-loaded with function strings for actively supported databases. See “Connection profiles,” in Chapter 7, “Managing Database Connections” in the *Replication Server Administration Guide Volume 1*.

Replication Server security

Replication Server security includes password-protected login names and a permission system based on the grant and revoke commands. Replication Server 12.0 and later supports third-party security services that ensure secure message transmission over the network and that enable user authentication. Replication Server 12.5 and later supports secure socket layer (SSL) session-based security through the Advanced Security option.

Login names

Each Replication Server uses login names that are distinct from data server login names. Many clients do not need a Replication Server login name because they accomplish their work through data server applications.

Replication Server login names

When you install a Replication Server, rs_init creates Replication Server login names that other Replication Servers and Replication Agents use to log in to Replication Server.

A replication system administrator creates and manages the Replication Server login names and passwords used to manage replicated data or replication system functions such as the addition of new users or a route change. Passwords can be encrypted.

Data server login names

Data server login names are used with a client application to connect to a data server. The application uses data stored by the data server, including data replicated by Replication Server. A Database Administrator creates and manages data server login accounts.

Client access to replicated copies of tables is also managed by the Database Administrator. Since Sybase recommends that replicated tables be read-only, clients may be permitted to view replicated data but should be prevented from inserting, deleting, or updating rows.

To modify replicated tables, a client must modify the primary data so that the Replication Server can distribute the changes to replicate databases that have subscriptions for the data. To modify a table, a client must have a login name on the data server where the primary data is stored as well as the permissions necessary to update the primary data.

Data server maintenance user login name

Replication Server uses a maintenance user login name for each local data server database that contains replicated tables. It is called the *maintenance user* login name because Replication Server uses it to maintain replicated tables. The Database Administrator must make sure that the maintenance user login name has the permissions needed to update the replicated tables in the database.

Normally, transactions applied by the maintenance user are filtered by the Replication Agent so they are not replicated out of a database. In certain applications, however, these transactions must be replicated. See Chapter 3, “Implementation Strategies” for more information on these application types.

Permissions

The grant and revoke commands are used to grant and revoke Replication Server permissions for clients. Table 1-1 lists the permissions that can be granted to clients.

Table 1-1: Replication Server permissions

Permission	Capabilities
sa	Gives recipient System Administrator capabilities. Clients with sa permission can perform any Replication Server command.

Permission	Capabilities
create object	Allows recipient to create, alter, or drop Replication Server objects, including replication definitions and subscriptions.
primary subscribe	Gives the recipient permission to create a subscription in a primary database, but not to create other objects. To create a subscription at a remote site, a client needs create object permission in the replicate database and create object or primary subscribe permission in the primary database.
connect source	This permission is required for login names used by the Replication Agent. It allows the recipient to execute the RCL commands that are reserved for Replication Agents.

Network-based security

With a third-party network-based security mechanism, users are authenticated by the security system at login. Authentication is the process of verifying that users are who they say they are. Users receive a credential that can be presented to remote servers in lieu of a password. As a result, users have seamless access to the components of the replication system through a single login.

Network-based security mechanisms also provide a variety of data-protection services, such as message confidentiality and out-of-sequence checking. Replication Server requests the service, prepares the data, and sends it to the network server for encryption or validation. Once the service is performed, data is returned to the requesting Replication Server for distribution.

Once a secure pathway has been established, data can move in both directions. Both ends of the pathway must support the same security mechanism and be configured the same. The security mechanism must be installed on all machines that make use of network security, and Replication Server 12.0 or later must be installed on all participating machines.

See the *Replication Server Reference Manual* for information about network-based security in the replication system.

Advanced Security option

Replication Server's Advanced Security option provides secure socket layer (SSL), session-based security. SSL is the standard for securing the transmission of sensitive information, such as credit card numbers and stock trades, over the Internet.

SSL provides a lightweight, easy-to-administer security mechanism with several encryption algorithms. It is intended for use over those database connections and routes where heightened security is required.

See the *Replication Server Administration Guide Volume 1* for information about using the Advanced Security option.

Summary

- Replication Server maintains copies of tables in different databases on a network. The replicated copies provide two advantages to users at the sites: faster response and greater availability.
- A replication system built on Replication Server uses the Sybase ECDA to connect components—data servers, Replication Servers, Replication Agents, and client applications.
- Replication Server is designed with an open interface that allows non-Sybase data servers to be included in a replication system.
- One copy of a table is the primary version. All others are replicated copies.
- Using subscriptions, a replicated copy of a table may contain a subset of the rows in a table.
- Replication Server security consists of login names, passwords, and permissions. Replication Server also supports third-party network-based security mechanisms.
- Replication Server uses optimistic concurrency control that processes failures when they occur. Compared to other methods, optimistic concurrency control provides greater data availability, uses fewer resources, and works well with heterogeneous data servers.

Application Architecture for Replication Systems

This chapter discusses several topics that are important for you, as an application designer and user, to explore before you implement Replication Server.

Topic	Page
Application types	25
Effects of loose consistency on applications	32
Methods for updating primary data	33

Application types

Determining the type of Replication Server application you build will in part determine the type of replication strategy you use to implement the application. Chapter 3, “Implementation Strategies” covers various replication scenarios.

Replication Server supports the following basic application types:

- Decision support
- Distributed online transaction processing (OLTP)
- Remote OLTP using request functions
- Warm standby

Each of these application types differs in the way it updates primary data and in the way it distributes primary and replicated data within the replication system.

Decision-support applications

Decision-support clients and production online transaction processing (OLTP) clients use data in different ways. Decision-support clients execute lengthy queries that hold locks on tables to guarantee serial consistency. OLTP clients, on the other hand, execute transactions that must complete quickly and cannot accept the delays caused by decision-support clients' data locks. The two types of clients do not interfere with each other if they maintain separate copies of replicated tables.

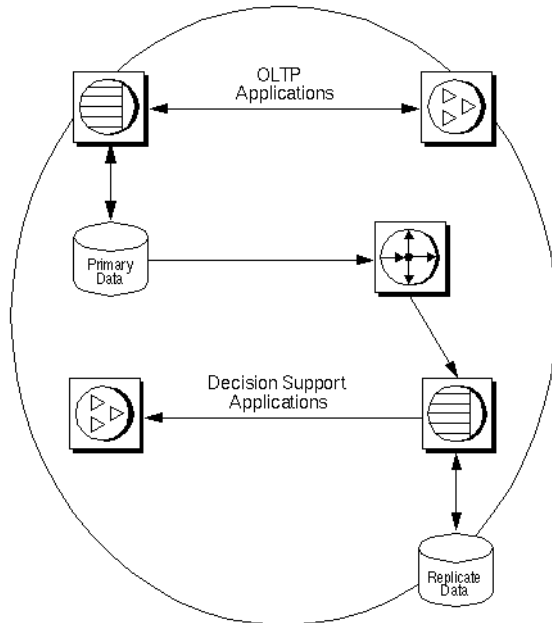
Replication Server off-loads processing associated with decision-support applications from a centralized online transaction processing application onto local servers. The primary database manages the transaction processing, and the replicate databases at local sites handle requests for information from decision-support clients. Providing a separate, reference-only copy of the data allows OLTP systems to continue unobstructed.

Multiple copies of tables containing primary data used in decision-support applications can be maintained at a single site or at multiple sites over the network.

Multiple copies at a single site

Sometimes it is useful to maintain multiple replicate copies of a table at a single site. A subscription specifies the database where Replication Server maintains the replicated data. You can create subscriptions for multiple copies of a table by creating the table in different databases at the same site and then creating subscriptions for each one.

If OLTP and decision-support clients are on the same LAN, one Replication Server can manage both the primary data and the replicate data. Figure 2-1 illustrates such an arrangement.

Figure 2-1: Single LAN decision support replicate

For best performance, the databases are usually maintained by different data servers. The subscriptions can request different subsets of the data to be maintained in each database, so the replicated copies do not have to be identical.

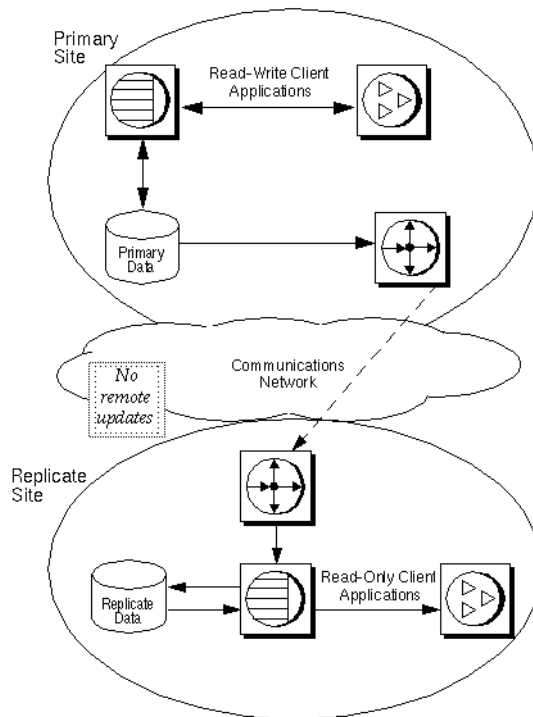
If you must have two copies of a table in the same database, you can use multiple replication definitions for a primary table. One replication definition could have publishers as the replicate table name, and the other publishers2. Multiple replication definitions are also useful if you want different replicates to receive different column subsets. See “Multiple replication definitions” on page 64.

Another way to update multiple tables in an Adaptive Server database is to use stored procedures. Code the multiple updates in the stored procedures and write Replication Server function strings to execute the stored procedures. You can also use replicated functions and stored procedures to update multiple tables.

Multiple copies distributed over a network

When copies of tables are distributed over a WAN in a decision-support application, all updates are performed by applications executing at the primary site and are distributed to the remote sites that have subscriptions for the data. Figure 2-2 illustrates this arrangement.

Figure 2-2: Multiple LAN decision support replicate



This type of system uses the centralized primary maintenance method of updating primary data. Clients at remote sites subscribe to replication definitions or publications of primary data. They do not update primary data. See “Centralized primary maintenance” on page 34 for information on this method.

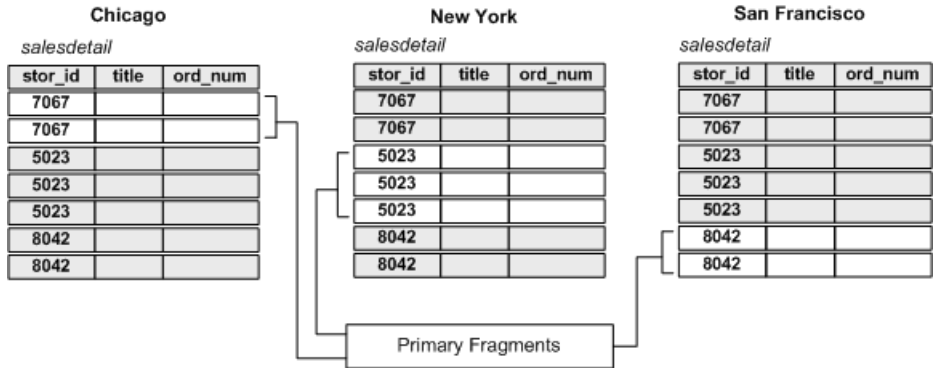
Distributed OLTP applications

Although some distributed transaction-processing applications maintain centralized primary data, others fragment primary data among replicate sites.

A **primary fragment** is a horizontal segment of a table that holds the primary version of a set of rows. Updates are applied to the primary version first and are then distributed to sites that have replicated copies of the data.

Sites that are responsible for, or own, portions of a table by definition have multiple primary fragments. For example, the salesdetail table in Figure 2-3 has primary fragments in Chicago, New York, and San Francisco:

Figure 2-3: Table with multiple primary fragments



A key constructed from one or more columns identifies the primary fragment where a row belongs. The key for the salesdetail table is the stor_id column.

- Rows with “7067” in the stor_id column belong to the primary fragment at the Chicago site.
- Rows with “5023” in the stor_id column belong to the primary fragment at the New York site.
- Rows with “8042” in the stor_id column belong to the primary fragment at the San Francisco site.

There are three application models based on multiple primary fragments:

- Distributed primary fragments – in this model, tables at each site contain both primary and replicated data. Updates to the primary version are distributed to other sites. Updates to non-primary data are received from the primary site.
- Corporate rollup – in this model, multiple primary fragments maintained at remote sites are consolidated into a single aggregate replicate table at a central site.
- Redistributed corporate rollup – this model is the same as the corporate rollup model, except that the consolidated table is redistributed.

More information about these models can be found in Chapter 3, “Implementation Strategies”

Remote OLTP using request functions

Replicated functions can be used to execute transactions remotely. Client applications at remote sites can update primary data asynchronously with request functions. The client application does not require a network connection to the primary site, and the request can be accepted by the Replication Server even when the primary site is not available.

Once the request function executes the stored procedure in the primary database, Replication Server may replicate some or all of the data changes made in the primary database. These changes can be propagated to replicate databases as data rows or as applied functions.

Local update applications

A local update application allows clients at a remote site to see the updates they have entered before the replication system returns them from the primary site. For example, if a customer account is updated at a remote site, clients at the site can see the results of the transaction even if the primary site is not accessible.

Local updates can be performed by using a pending updates table. For each replicated table, a corresponding local table contains provisional updates—updates that have been submitted to the primary site, but that have not been returned through the replication system. Client applications update the pending transactions table and, at the same time, send a request function to the primary site. See “An example using a local pending table” on page 72 for information on implementing this type of application.

When the update succeeds against the primary copy, it is distributed to remote sites, including the site where the transaction originated. You can create a function string or replicated stored procedure to update the replicated table and delete local updates from the pending table. This makes it possible for client applications to know which transactions have been confirmed and which are pending.

Standby applications

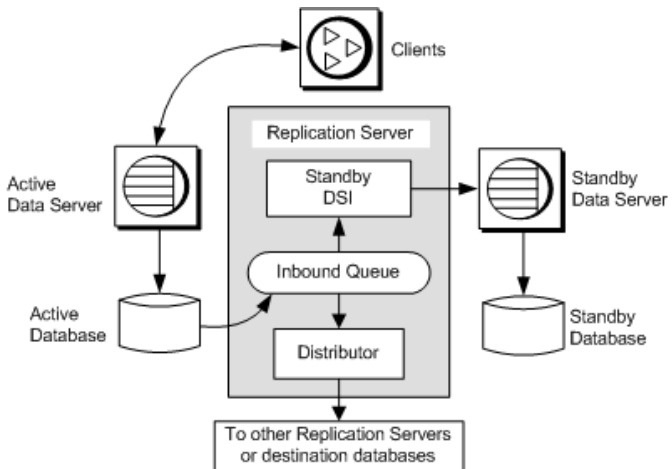
A warm standby application is a Replication Server application that maintains a pair of Adaptive Server databases, one of which functions as a standby copy of the other.

Client applications generally update the active database, while Replication Server maintains the standby database as a copy of the active database. Replication Server keeps the standby database consistent with the active database by replicating transactions retrieved from the active database transaction log.

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

Figure 2-4 illustrates a warm standby system.

Figure 2-4: A warm standby system



The two databases in a warm standby application appear as a single logical database in the replication system. Depending on your application, this logical database may not participate in replication, or it may be a primary database or a replicate database with respect to other databases in the replication system.

Several Replication Server and RepAgent features explicitly support warm standby applications. See the *Replication Server Administration Guide Volume 2* for more detailed information about warm standby applications.

Effects of loose consistency on applications

Data in a replicate database is “loosely consistent” with the data in the primary database. Replicate data lags behind primary data by the amount of time it takes to distribute updates from the primary database to another part of the replication environment. This *latency* can be measured in seconds (or less) when the system is working properly. If a component fails—if, for example, a network connection is temporarily lost—updates can be delayed for minutes, hours, or days. Thus, latency information can be used to monitor the performance and health of the replication environment.

Although replicate data may lag behind primary data, it is transactionally consistent with the primary data. Replication Server delivers transactions to replicate databases in the order they are committed in the primary database. This ensures that the replicate data goes through the same series of states as the primary data.

The importance of loose consistency varies by application and even within an application. Some applications tolerate average system lag time and occasional, longer delays with no special provisions. Some require special handling when the lag time becomes too great, and some require special handling for certain types of transactions.

Controlling risks in high-value transactions

The delay introduced by data replication adds risk to some business decisions. For example, a banking application that approves cash withdrawals uses the most current account information available to verify that a customer’s balance is sufficient to cover the withdrawal. If withdrawals processed in a primary database have not reached the replicate database, an application using the replicate database risks approving a withdrawal that exceeds the funds available in the customer’s account.

To limit risk, the banking application can distinguish between high-value transactions and low-value transactions. For example, it might approve a \$100 withdrawal based on the account balance in the local replicate database, but it would log in to the primary database to check the account balance before approving a \$1000 withdrawal.

Measuring lag time

A measure of the latency for a replicate site can be used to limit risks for some transactions. A small lag time indicates that the primary and replicate data are nearly consistent. An extensive lag time indicates a greater potential difference between the primary and replicate data.

An application can use a measure of lag time to:

- Limit risk by restricting the transactions clients can execute as the lag time increases. For example, the banking application described in the previous section could include the lag time in its approval formula. It might allow withdrawals of up to \$1000 based on the balance in the local replicated table when the latency is less than a minute. If the lag time is more than a minute, however, the application would log in to the primary database to approve withdrawals of more than \$500.
- Provide clients with a “performance meter” for data replication. Clients can use an estimate of lag time as an advisory. For example, a decision-support user, noting that the lag time is high, might wait for the local data to catch up with the primary data, and for the lag time to decrease, before running an analysis based on replicate data.

See Chapter 4, “Performance Tuning” in the *Replication Server Administration Guide Volume 2* for information on measuring latency.

Methods for updating primary data

In a replication system, the primary copy of a data row is the definitive copy. An update committed in the primary database is authoritative and is distributed to all databases with subscriptions for the data.

Replication Server distributes transactions after they are committed in the primary database. Because changes made to replicate data are not distributed, make the data in replicate databases read-only for clients and route all client transactions to the primary database.

There are four ways to update primary data in a replication system based on Replication Server:

- Primary data maintenance is centralized at the primary site. Clients cannot update primary data from remote sites.
- Clients at remote sites update primary data through network connections.

- Clients at remote sites update primary data using request functions.
- Primary data maintenance is distributed at multiple primaries. Any resulting conflicts must be avoided or resolved.

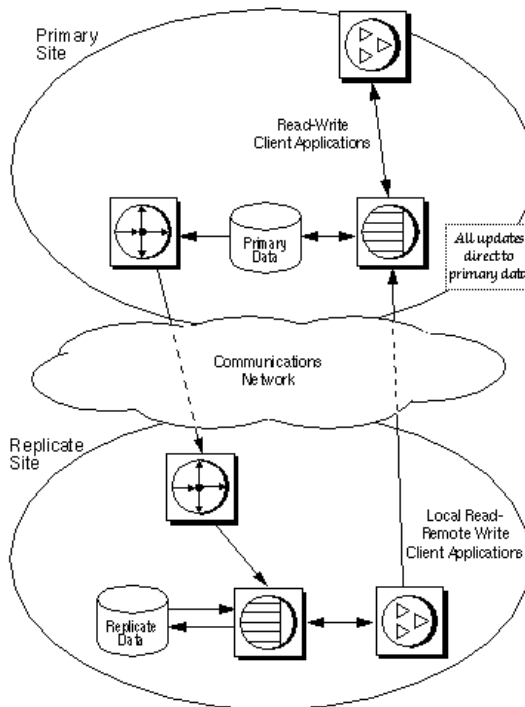
Centralized primary maintenance

This method is the simplest, and the most restrictive, for remote clients. Client applications at remote sites use replicate data for reference only. This architecture can be used to create a copy of a production OLTP system that allows decision support applications to run separately from the OLTP system. See “Decision-support applications” on page 26 for information on applications that use this method for updating primary data.

Primary maintenance via network connections

For some applications, clients at remote sites must update primary data. The easiest way to do this is for the client to connect directly to the primary data server through the network. Replication Server distributes updates from the primary to the remote sites in the usual way.

Figure 2-5 illustrates this design.

Figure 2-5: Primary data maintenance via network

This architecture uses client connections through the WAN. It is useful for applications with large amounts of data and low update rates. Updates are executed directly against the primary data so that they can be distributed to all sites that have subscriptions for the data. At remote sites, local replicated copies of the data can be used for decision-support applications to reduce the network load.

Managing update conflicts for multiple primaries

When there are multiple primaries, remote updates should be designed so that they do not introduce errors resulting from simultaneous requests to update the same information from multiple remote sites.

If updates from two different sites conflict when they are applied at a primary site, one of the updates will be rejected, and the replicated data at the site where the update was rejected will be inconsistent with primary data.

To handle inter-site concurrency conflicts when there are multiple primaries, follow these guidelines:

- *When each row has an owner* – design the application so that inter-site conflicts are impossible. For example, restrict the updates performed at one site to rows that cannot be updated by clients at other sites. This guarantees that updates will not conflict at the primary site.
- *When there is no segmentation of ownership* – add version control information to function strings to allow conflicts to be detected and handled.

Designing conflicts out of an application

One way to handle conflicting updates from different sites is to construct the application and the environment so that conflicts cannot happen. For example, an application with customer account information distributed to each of several branch offices could require that an account be updated only at the customer's home branch. This prevents two clients from updating the same account at the same time in different databases.

Another technique is to include a location key, such as a branch ID, in the primary key for replicated tables. If each site uses a unique location key for all of its transactions, inter-site conflicts cannot occur.

Version-controlled updates

You can use version-control updates to:

- Detect and resolve conflicting updates
- Resolve multiple primary conflicts when there is no single primary source
- Make multiple requests to a single primary

Updates are accepted or rejected based on a version column that changes each time the row is updated. The version column can be a number that increases with each update, a timestamp, or some other value from a set of unique values.

To update a row, an application must provide the current value of the version column at the primary site. Typically, the value is provided as a parameter to a replicated stored procedure. The stored procedure at the primary site checks the version parameter and takes appropriate action if it detects a conflict. If the application chooses to roll back the transaction, it is written to the exceptions log.

Note Managing update conflicts using version control requires careful planning and design. When there are multiple primaries, it is usually simpler and more effective to establish owners for each row of a table.

Implementation Strategies

This chapter describes several models and strategies you can use to implement your Replication Server application design. It includes procedures and sample scripts that you can adapt to your own application.

Topic	Page
Overview of models and strategies	39
Basic primary copy model	40
Distributed primary fragments model	47
Corporate rollup	52
Redistributed corporate rollup	56
Warm standby applications	58
Model variations and strategies	63

Overview of models and strategies

The models discussed in this chapter are:

- Basic primary copy model – centralized primary data, distributed replicate data
- Distributed primary fragments model – both primary and replicate data distributed through the replication system
- Corporate rollup – distributed primary data, centralized replicate data
- Redistributed corporate rollup – same as corporate rollup, but updates redistributed to replicate databases
- Warm standby applications – two databases, one serving as backup for the other, which together as a logical unit may participate in replication

In addition, this chapter describes model variations and other strategies you can use:

- Multiple replication definitions
- Publications

- Request functions
- Pending tables
- Master/detail relationships

These methods are described in detail in “Model variations and strategies” on page 63.

The type of application you are building, the way you update primary data, and the way you manage potential update conflicts determine the model you use to implement your replication application.

For instance, you can use the basic primary copy model to implement either a decision-support application or a low-volume distributed OLTP system. You might implement a decision-support application using either the basic primary copy model or the redistributed corporate rollup model, depending on whether your primary data is centralized or fragmented. A distributed OLTP application might be implemented using the distributed primary fragment model, with or without corporate rollup, depending on additional decision-support needs.

Basic primary copy model

The basic primary copy model allows you to replicate data from a primary database to destination databases. This model is well suited to decision-support applications, although low-volume transaction-processing applications can update primary data remotely, either directly over the WAN or through request functions (replicated stored procedures). Primary data that is updated from remote sites can then be replicated back to subscribing sites.

You can implement the basic primary copy model by using any or all of the following:

- Table replication definitions
- Applied functions
- Request functions

This section provides basic examples for using table replication definitions and applied functions. For examples of request functions and other, more advanced uses of the primary copy model, see “Model variations and strategies” on page 63.

Using table replication definitions

Using table replication definitions allows you to replicate data from a primary source as read-only copies.

You can create one or many replication definitions for a primary table although a particular replicate table can subscribe to only one of them. See “Multiple replication definitions” on page 64 for an example using multiple replication definitions.

You also can collect replication definitions in a publication and subscribe to all of them at one time with a publication subscription. See “Publications” on page 66 for an example using publications.

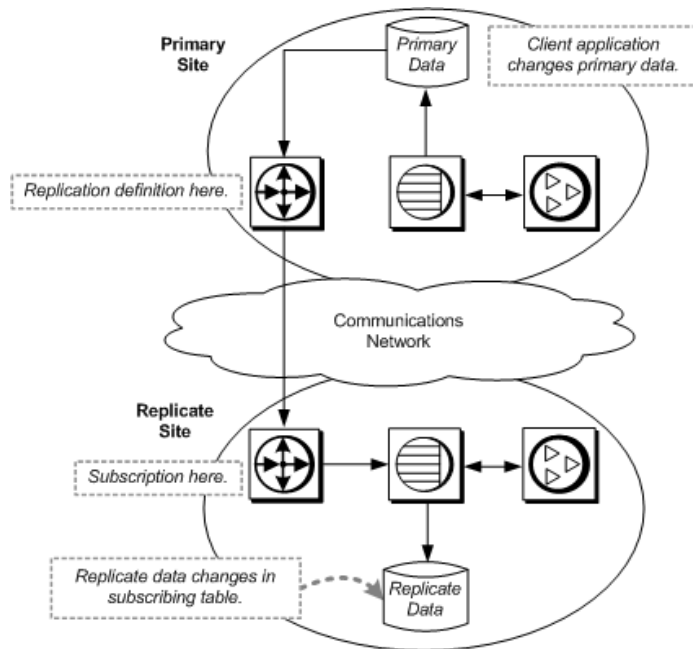
For each table you want to replicate according to the basic primary copy model, you need to:

- Set up routes and connections between Replication Servers.
 - Create the table you want to replicate in the primary database.
 - Create the table (or tables) to which you want to replicate in destination databases.
 - Create indexes and grant appropriate permissions on the tables.
- At the primary site:
- Mark the primary table for replication using the `sp_setreptable` system procedure.
 - Create one (or more) replication definitions for the table at the primary Replication Server.
- At the replicate sites:
- Create subscriptions for the table replication definitions at each replicate Replication Server.

See the *Replication Server Administration Guide Volume 1* for details on setting up the basic primary copy model.

In Figure 3-1, a client application at the primary (Tokyo) site makes changes to the publishers table in the primary database. At the replicate (Sydney) site, the publishers table subscribes to the primary publishers table—for those rows where `pub_id` is equal to or greater than 1000.

Figure 3-1: Basic primary copy model using table replication definitions



Marking the table for replication

This script marks the publishers table for replication.

```
-- Execute this script at Tokyo data server
-- Marks publishers for replication
sp_setreptable publishers, 'true'
go
/* end of script */
```

Replication definition

This script creates a table replication definition for the publishers table at the primary Replication Server.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition pubs_rep
create replication definition pubs_rep
```

```
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
 city varchar(20),
 state varchar(2))
primary key (pub_id)
go
/* end of script */
```

Subscription

This script creates a subscription for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription pubs_sub
Create subscription pubs_sub
for pubs_rep
with replicate at SYDNEY_DS.pubs2
where pub_id >= 1000
go
/* end of script */
```

Using applied functions

You can also use applied functions to replicate stored procedure invocations to remote sites with replicate data. If you use applied functions to replicate primary data, you can:

- Reduce network traffic over the WAN
- Increase throughput and decrease latency because applied functions execute more rapidly
- Enable a more modular system design

In the following example, a client application at the primary (Tokyo) site executes a user stored procedure, `upd_publishers_pubs2`, which makes changes to the `publishers` table in the primary database. Execution of `upd_publishers_pubs2` invokes function replication, which causes the corresponding stored procedure, also named `upd_publishers_pubs2`, to execute on the replicate data server.

To create an applied function for an application that implements the basic primary copy model, you need to:

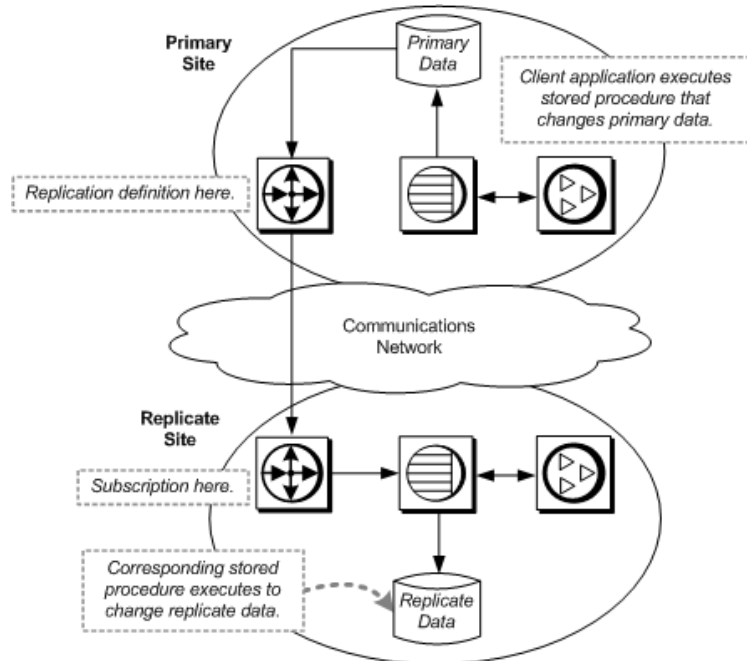
At the primary site:

- Create the user stored procedure in the primary database.
- Mark the user stored procedure for replicated function delivery using `sp_setreproc`.
- Grant the appropriate procedure permissions to the appropriate user.
- At the primary Replication Server, create the function replication definition for the stored procedure with parameters and datatypes that match those of the stored procedure. You can specify only the parameters you want to replicate.

At the replicate site:

- Create a stored procedure in the replicate database with the same parameters (or a subset of those parameters) and datatypes as those created in the primary database. Grant appropriate permissions to the procedure to the maintenance user.
- Create a subscription to the function replication definition in the replicate Replication Server.

Figure 3-2: Basic primary copy model using applied functions



Stored procedures

This script creates stored procedures for the publishers table at the primary and replicate sites.

```
-- Execute this script at Tokyo and Sydney data servers
-- Creates stored procedure upd_publishers_pubs2
create procedure upd_publishers_pubs2
(@pub_id char(4),
@pub_name varchar(40),
@city varchar(20),
@state char(2))
as
update publishers
set
    pub_name = @pub_name,
    city = @city,
    state = @state
where
    pub_id = @pub_id
```

```
go
/* end of script */
```

Function replication definition

This script creates an applied function replication definition for the publishers table at the primary Replication Server. The replication definition uses the same parameters and datatypes as the stored procedure in the primary database.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
_upd_publishers_pubs2_repdef
create applied function replication definition
    upd_publishers_pubs2_repdef
with primary at TOKYO_DS.pubs2
with all functions named upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
go
/* end of script */
```

Subscriptions

You can create a subscription for a function replication definition in one of two ways:

- Use the create subscription command and the no-materialization method.
Use this method if primary data is already loaded at the replicate, and updates are not in progress.
- Use the define subscription, activate subscription, and validate subscription commands and the bulk materialization method.
Use this method if you are coordinating loading data with updates.

Examples of both methods follow.

Using the no-materialization method

This script creates a subscription at the replicate Replication Server using the no-materialization method for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
```



```

-- Creates subscription using no-materialization
-- for upd_publishers_pubs2_repdef
create subscription upd_publishers_pubs2_sub
  for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
without materialization
go
/* end of script */

```

Using bulk materialization

This script defines, activates, and validates a subscription at the replicate Replication Server for the replication definition defined at the primary Replication Server.

```

-- Execute this script at Sydney Replication Server
-- Creates subscription using bulk materialization
-- for upd_publishers_pubs2_repdef
define subscription upd_publishers_pubs2_sub
  for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go

activate subscription upd_publishers_pubs2_sub
  for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go
/* Load data. If updates are in progress, use activate
subscription with the "with suspension" clause and
resume connection after the load. */

validate subscription upd_publishers_pubs2_sub
  for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go
/* end of script */

```

Distributed primary fragments model

In this model, tables at each site contain both primary and replicated data. However, each site functions as a primary site for a particular subset of rows called a *fragment*. Updates to the primary fragment are distributed to other sites. Updates to nonprimary data are received from the primary sites of other fragments.

Applications that use the distributed primary fragments model have distributed tables that contain primary and replicated data. The Replication Server at each site distributes modifications made to local primary data to other sites and applies modifications received from other sites to the data that is replicated locally.

The following tasks must be performed at each site to replicate a table in the distributed primary fragments model:

- Create the table in each database. The table should have the same structure in each database.
- Create indexes and grant appropriate permissions on the tables.
- Allow for replication on the tables using the `sp_setreptable` system procedure.
- Create a replication definition for the table at each site.
- At each site, create a subscription for the replication definition at the other sites. If n is the number of sites, create $n-1$ subscriptions.

Figure 3-3 diagrams the flow of data for distributed primary fragments:

Figure 3-3: Distributed primary fragments model

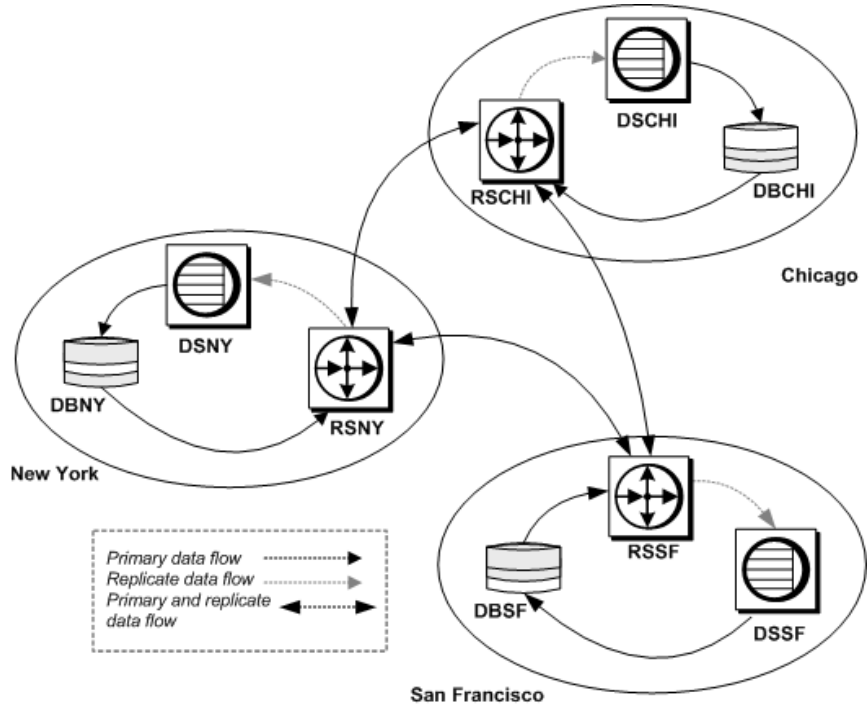
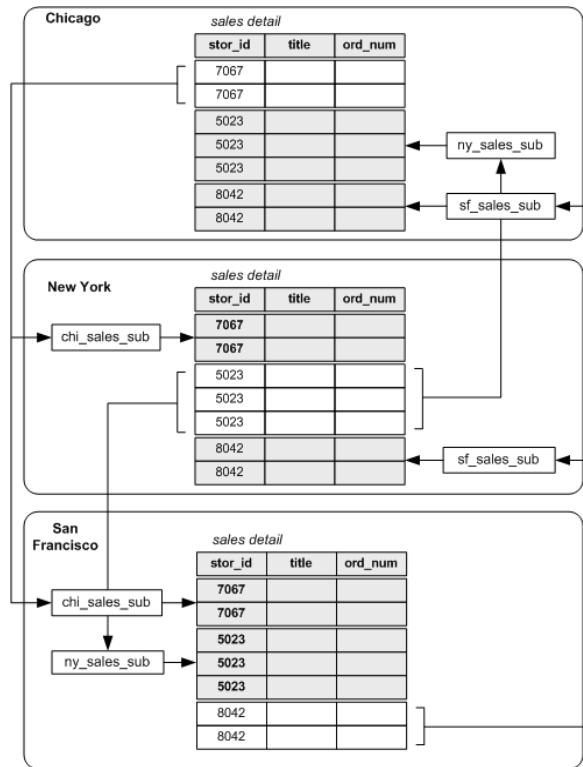


Figure 3-4 illustrates a salesdetail table set up with distributed primary fragments at three sites. Each site receives replicated data via two subscriptions.

Figure 3-4: Table with three distributed primary fragments



Replication definitions

These scripts create replication definitions for the salesdetail table at each site:

```
-- Execute this script at Chicago RSCHI.
-- Creates replication definition chi_sales.
create replication definition chi_sales_rep
with primary at DSCHI.DBCHI
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar(6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
```

```

/* end of script */
-- Execute this script at New York RSNY.
-- Creates replication definition ny_sales.
create replication definition ny_sales_rep
with primary at DSNY.DBNY
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
-- Execute this script at San Francisco RSSF.
-- Creates replication definition sf_sales.
create replication definition sf_sales_rep
with primary at DSSF.DBSF
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */

```

Subscriptions

Each site has a subscription to the replication definitions at the other two sites. These scripts create the subscriptions:

```

-- Execute this script at Chicago RSCHI.
-- Creates subscriptions to ny_sales and sf_sales.
create subscription ny_sales_sub
for ny_sales_rep
with replicate at DSCHI.DBCHI
where stor_id = '5023'
go
create subscription sf_sales_sub
for sf_sales_rep
with replicate at DSCHI.DBCHI

```

```
        where stor_id = '8042'
go
/* end of script */
-- Execute this script at New York RSNY.
-- Create subscriptions to chi_sales and sf_sales.
create subscription chi_sales_sub
    for chi_sales_sub
    with replicate at DSNY.DBNY
    where stor_id = '7067'
go
create subscription sf_sales_sub
    for sf_sales_rep
    with replicate at DSNY.DBNY
    where stor_id = '8042'
go
/* end of script */
-- Execute this script at San Francisco RSSF.
-- Creates subscriptions to chi_sales and ny_sales.
create subscription chi_sales_sub
    for chi_sales_rep
    with replicate at DSSF.DBSF
    where stor_id = '7067'
go
create subscription ny_sales_sub
    for ny_sales_rep
    with replicate at DSSF.DBSF
    where stor_id = '5023'
go
/* end of script */
```

Corporate rollup

In this model, multiple primary fragments maintained at remote sites are consolidated into a single aggregate replicate table at a central site.

The corporate rollup model has distributed primary fragments and a single, centralized consolidated replicate table. The table at each primary site contains only the data that is primary at that site. No data is replicated to these sites. The corporate rollup table is a “rollup” of the data at the primary sites.

The corporate rollup model requires distinct replication definitions at each of the primary sites. The site where the data is consolidated has a subscription for the replication definition at each primary site.

Replication Agents are required at the primary sites but not at the central site, since data will not be replicated from that site.

These tasks must be performed to create a corporate rollup from distributed primary fragments:

- Create the table in each primary database and in the database at the central site. The tables should have the same structure and the same name.
- Create indexes and grant appropriate permissions on the tables.
- In each remote database, allow for replication on the table with the `sp_setreptable` system procedure.
- Create a replication definition for the table at each remote site.
- At the headquarters site, where the data is to be consolidated, create subscriptions for the replication definitions at the remote sites.

Figure 3-5 illustrates the flow of data for a corporate rollup application model.

Figure 3-5: Corporate rollup model with distributed primary fragments

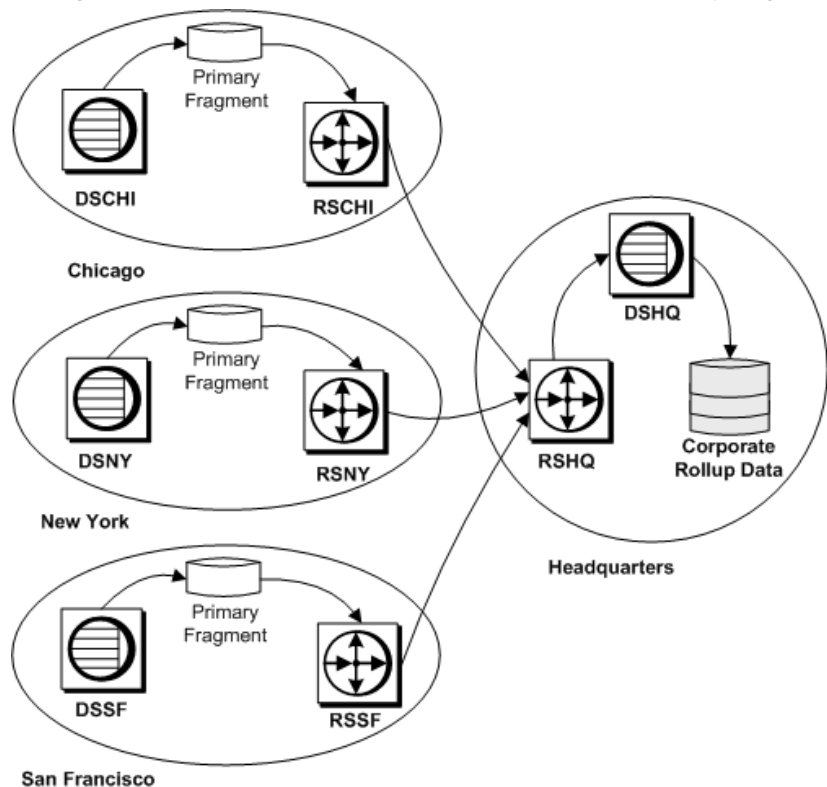
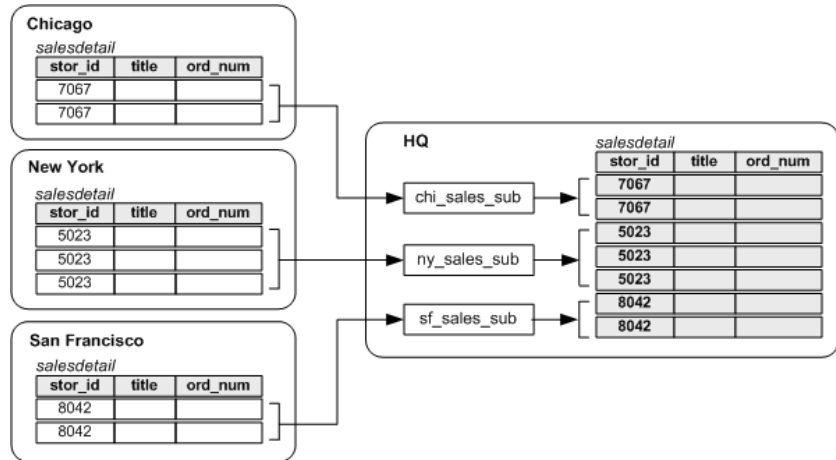


Figure 3-6 illustrates a salesdetail table with a corporate rollup at a headquarters site. The headquarters site receives data from the remote sites via three subscriptions.

Figure 3-6: Table with multiple primary fragments



Replication definitions

These scripts create replication definitions for the salesdetail table at each primary site:

```
-- Execute this script at Chicago RSCHI.
-- Creates replication definition chi_sales.
create replication definition chi_sales_rep
with primary at DSCHI.DBCHI
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar(6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```



```

-- Execute this script at New York RSNY.
-- Creates replication definition ny_sales.
create replication definition ny_sales_rep
with primary at DSNY.DBNY
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
-- Execute this script at San Francisco RSSF.
-- Creates replication definition sf_sales.
create replication definition sf_sales_rep
with primary at DSSF.DBSF
with all tables named 'salesdetail'
(stor_id char(4),
 ord_num varchar(20),
 title_id varchar6),
 qty smallint,
 discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */

```

Subscriptions

The headquarters site has subscriptions to the replication definitions at each of the three primary sites. The primary sites have no subscriptions. This script creates the subscriptions in the RSHQ Replication Server:

```

-- Execute this script at Headquarters RSHQ.
-- Creates subscriptions to chi_sales, ny_sales,
-- and sf_sales.
create subscription chi_sales_sub
for chi_sales_rep
with replicate at DSHQ.DBHQ
where stor_id = '7067'
go
create subscription ny_sales_sub
for ny_sales_rep

```

```
        with replicate at DSHQ.DBHQ
        where stor_id = '5023'
go
create subscription sf_sales_sub
    for sf_sales_rep
    with replicate at DSHQ.DBHQ
    where stor_id = '8042'
go
/* end of script */
```

Redistributed corporate rollup

The redistributed corporate rollup is the same as the corporate rollup model, except that the consolidated table is redistributed back to the remote sites.

Primary fragments distributed at remote sites are rolled up into a consolidated table at a central site. At the site where the fragments are consolidated, RepAgent processes the consolidated table as if it were primary data.

The consolidated table is described with a replication definition. Other sites can create subscriptions for this table.

Normally, RepAgent for Adaptive Server filters out updates made by the maintenance user. This ensures that replicated data is not redistributed as primary data.

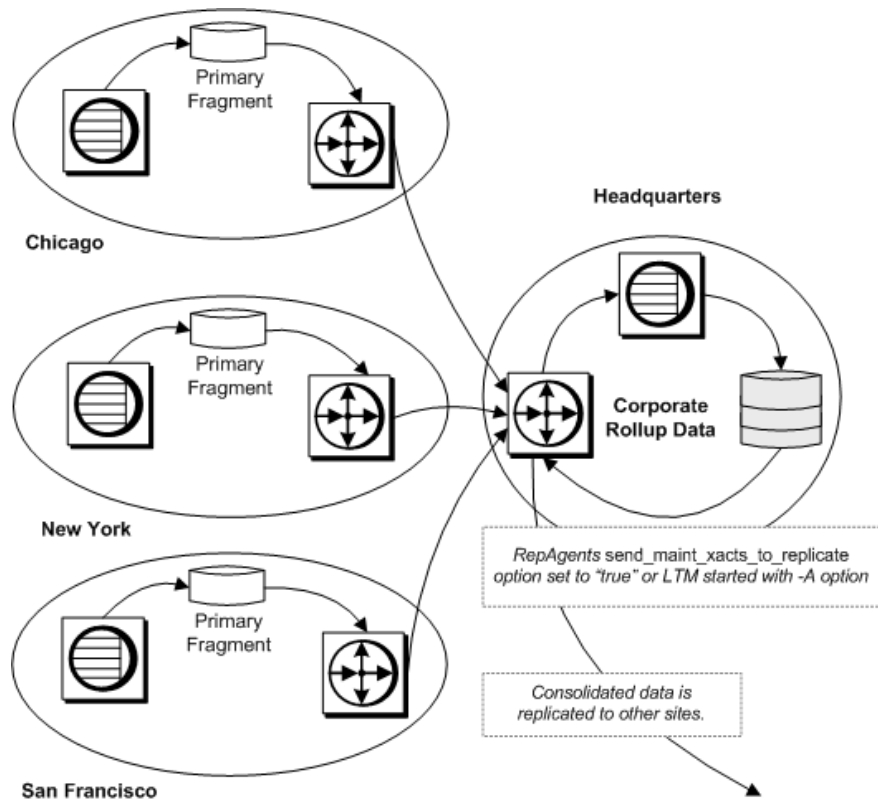
The RepAgent `send_maint_xacts_to_replicate` option is provided for the redistributed corporate rollup model. If you start RepAgent with `send_maint_xacts_to_replicate` set to “true,” RepAgent submits all updates to the Replication Server as if they were made by a client application.

If you use the redistributed corporate rollup model:

- Do not allow primary sites to resubscribe to their primary data. If they do, transactions could loop endlessly through the system.
- Do not allow applications to update the corporate rollup table. All updates should originate from the primary sites.

Figure 3-7 illustrates the flow of data in an application based on the Redistributed corporate rollup model.

Figure 3-7: Redistributed corporate rollup with distributed fragments



The design of the redistributed corporate rollup model is identical to the corporate rollup model, except that:

- RepAgent must be installed at the headquarters site for the DBHQ database. RepAgent must be started with the `send_maint_xacts_to_replicate` option set so that it will transfer log records from the maintenance user.
- RepAgent is required for the RSHQ RSSD, since data will be distributed from that site.
- A replication definition must be created for the `salesdetail` table at the headquarters site. Other sites can create subscriptions to this replication definition, but the primary sites must not subscribe to their own primary data.

- The RSHQ Replication Server must have routes to the other sites that create subscriptions for the consolidated replicate table. If the primary sites create subscriptions, routes must be created to them from RSHQ.

Warm standby applications

In a warm standby application, Replication Server maintains a pair of Adaptive Servers, one of which acts as the backup of the other.

Typically, client applications update the active database while Replication Server maintains the other database as a standby copy of the active database. If the active database fails, or if you need to perform maintenance on the active data server or database, you can switch to the standby database (and back) with little interruption of client applications.

In a warm standby application, you create three connections:

- A logical connection that Replication Server maps to the currently active database
- A physical connection for the active database
- A physical connection for the standby database

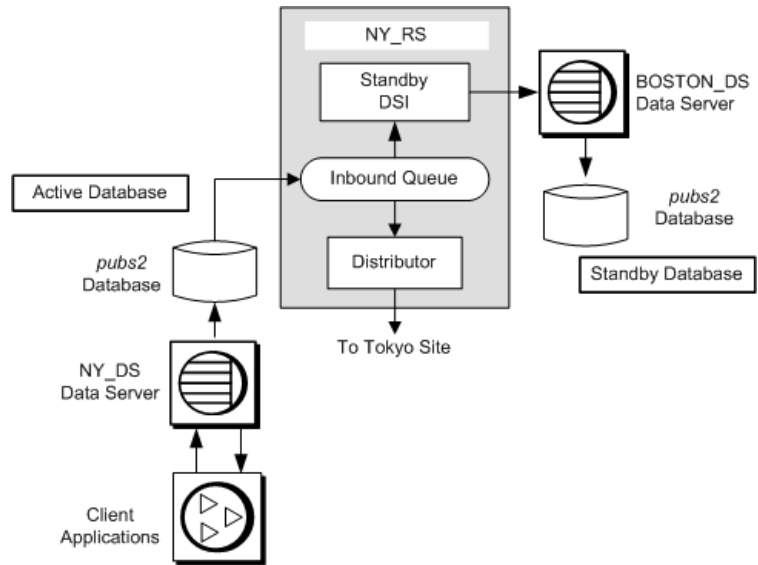
The logical database in a warm standby application may, with respect to other databases in the replication system, function as one of the following:

- A database that does not participate in replication
- A primary database
- A replicate database

The procedure in this section demonstrates how to set up a warm standby system for a database that acts as a primary database in a replication system.

Figure 3-8 illustrates a warm standby application operating on the BOSTON_DS data server for a pubs2 database on the NY_DS data server. The database is replicated to TOKYO_DS.

Figure 3-8: Warm standby system



In this scenario the pubs2 database acts as a primary database in a replication environment. The primary pubs2 database for which a standby is created is called the *active database*.

Setting up a warm standby application

You can use the following procedure to set up a warm standby application for an active database. In this procedure, an active database is already established. The procedure will be somewhat different if the active database has not yet been created. Make sure you review the warm standby information in the *Replication Server Administration Guide Volume 1* before proceeding.

Note You must use Adaptive Server databases for this procedure.

- 1 Mark the entire active database for replication to the standby database with the `sp_reptostandby` stored procedure.

`sp_reptostandby` enables replication of data manipulation language (DML) and supported data definition language (DDL) commands and stored procedures. Refer to Chapter 3, “Managing Warm Standby Applications,” in the *Replication Server Administration Guide Volume 2* for detailed information.

- 2 Reconfigure RepAgent using the `sp_config_rep_agent` stored procedure with the `send_warm_standby_xacts` option. Restart RepAgent.
- 3 Grant `replication_role` to the active database maintenance user.
- 4 On the active data server, add the maintenance user of the standby database to the active database, and grant `replication_role` to the new maintenance user. This step ensures that the maintenance user ID exists in the standby database after the database is loaded (step 8).
- 5 Log in to the Replication Server that is to manage the warm standby database, and create a logical connection for the active database, using the `create logical connection` command. The name of the logical connection must be the same as the name of the active database.

Note If you create the logical connection *before* you create the active database connection, use different names for the logical connection and the active database.

- 6 On the standby data server, create the standby database with the same size as the active database.
- 7 Use Sybase Central or `rs_init` to create the standby database connection. For more information, see the Replication Server online help and the Replication Server installation and configuration guide for your platform.

After the connection is created, log in to Replication Server and use the `admin logical_status` command to make sure that the new connection is “active.”

- 8 Initialize the standby database using `dump and load` without the `rs_init` “dump marker” option. (Or you can use `bcp`. Refer to the *Replication Server Administration Guide Volume 2* for more information.)
 - a On the Replication Server, suspend the active database connection.

Note If you cannot suspend the active database, use `dump and load` with the `rs_init` “dump marker” option.

- b On the active Adaptive Server, dump the active database.

- c Load the active database dump into the standby database.
 - d On the standby Adaptive Server, put the standby database online.
- 9 On the Replication Server, resume connections to the active and standby databases, using the resume connection command.
- Check the logical status, using the `admin logical_status` command. Do not continue unless both active and standby databases are marked “active.”
- 10 Verify that modifications occur from active to standby database.
- Using `isql`, update a record in the active database and then verify the update in the standby database.

Switching to the standby database

If it becomes necessary to switch from the active database to the standby database, you need to take steps to prevent client applications from executing transactions against or updating the active database. After the switch is complete, clients can connect to the new active database to continue their work. See “Switching clients to the new database” on page 62 for details.

Before switching to a standby database, you should determine whether a switch is necessary:

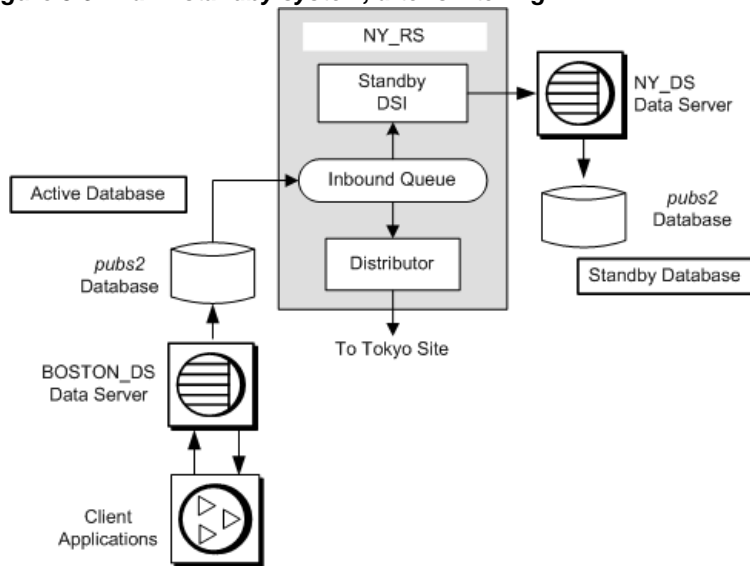
- Don’t switch if the active data server is experiencing a transient failure. A transient failure is a failure from which the Adaptive Server recovers when restarted, without additional recovery steps.
- Do switch if the active database will be unavailable for a long period of time.

You must use the `switch active` command to switch the active and standby databases. The following procedure illustrates how to switch the warm standby system illustrated in Figure 3-8 on page 59 from the active database to the standby database.

- 1 On the Replication Server, use `switch active` to switch processing to the standby database.
- 2 Monitor progress of the switch. The switch is complete when the standby connection is active and the previously active connection is suspended.
 - a On the Replication Server, check the logical status, using the `admin_logical_status` command.

- b To follow the progress of the switch, check the last several entries in the Replication Server error log.
- 3 Start the RepAgent for the new active database.
- 4 Decide what you want to do with the old active database. You can:
 - Bring the database online as the new standby database, and resume connections so that Replication Server can apply new transactions, or
 - Drop the database connection using the drop connection command. You can add it again later as the new standby database.
- 5 Using isql, update a record in the new active database, and then check the update the new standby database.

Figure 3-9: Warm standby system, after switching



Switching clients to the new database

Switching from the active to the standby database does not switch client applications to the new active data server and database. You must devise a method to handle client switching. For example, you could:

- Set up two interfaces files, one for client applications and one for Replication Server. At switch time, modify the client interfaces file to point to the new active server.

- Create an interfaces file entry with a symbolic data server name for use by client applications. At switch time, modify the address information associated with the symbolic name.
- Use a mechanism, such as an intermediate Open Server, to map the client application data server connections to the currently active data server automatically.

See the *Replication Server Administration Guide Volume 2* for more information.

Model variations and strategies

This section describes some model variations and other strategies you can use to implement your replication system design. They are:

- Multiple replication definitions – a strategy using multiple replication definitions for a primary table to specify different table names, column sets, and column names, thereby presenting differing views of the primary table to subscribing replicate tables
- Publications – a strategy that allows users to subscribe to a set of replication definitions with a single subscription
- Pending tables – a strategy used with request functions that allows users to see the results of updates to primary data before the update has been returned to the replicate site
- Implementing master/detail relationships – uses request functions and stored procedures to ensure proper subscription migration

Multiple replication definitions

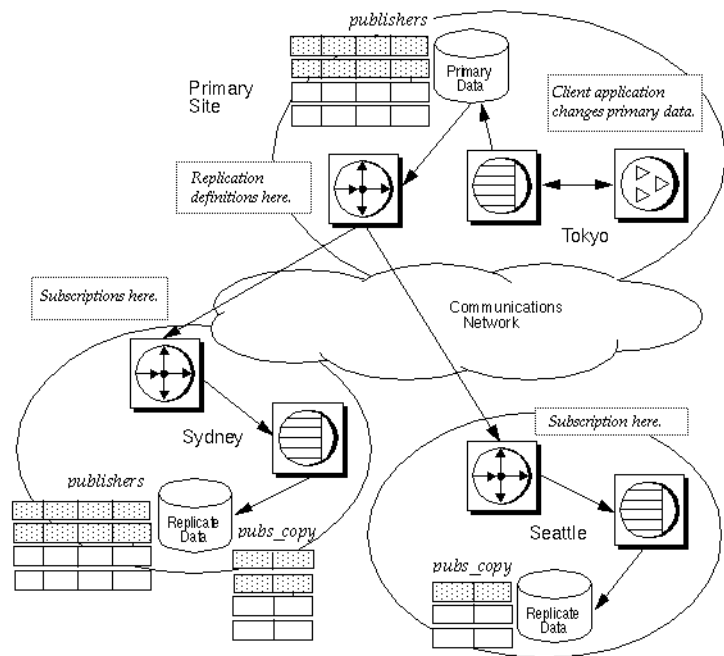
You can create multiple replication definitions for a single primary table. Each replication definition can specify different table names, column sets, and column names, thereby presenting different views of the primary table to each of the subscribing replicate tables.

Note You can create multiple replication definitions for a primary table, and a replicate table can subscribe to multiple table replication definitions. However, a replicate table can subscribe only to one replication definition per primary table.

To set up a system using multiple replication definitions, follow the directions in “Using table replication definitions” on page 41, creating multiple replication definitions as needed and a subscription for each one. When you use multiple replication definitions, you are using a variation of the primary copy model.

In Figure 3-10, a client application at the primary (Tokyo) site makes changes to the publishers table in the primary database. At one replicate (Sydney) site, the publishers table subscribes to the complete table, and the pubs_copy table subscribes only to the pub_id and pub_name columns. At another replicate (Seattle) site, the pubs_copy table subscribes to the pub_id and pub_name columns, where pub_id is equal to or greater than 1000.

Figure 3-10: Multiple replication definitions



Replication definitions

These scripts create table replication definitions for the publishers table at the primary Replication Server. Each replication definition describes a different view of the primary table.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definitions pubs_rep and
-- pubs_copy_rep
create replication definition pubs_rep
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
 city varchar(20),
 state varchar(2))
primary key (pub_id)
go
```

```
create replication definition pubs_copy_rep
with primary at TOKYO_DS.pubs2
with primary table named 'publishers'
with replicate table named 'pubs_copy'
(pub_id char(4),
 pub_name varchar(40))
primary key (pub_id)
go
/* end of script */
```

Subscriptions

These scripts create subscriptions for the replication definitions defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription pubs_sub and pubs_copy_rep
Create subscription pubs_sub
for pubs_rep
with replicate at SYDNEY_DS.pubs2
go

create subscription pubs_copy_sub
for pubs_copy_rep
with replicate at SYDNEY_DS.pubs2
go
/* end of script */
-- Execute this script at Seattle Replication Server
-- Creates subscription pubs_copy_sub
create subscription pubs_copy_sub
for publ_copy_rep
with replicate at SEATTLE_DS.pubs2
where pub_id >= 1000
go
/* end of script */
```

Publications

Use publications to collect replication definitions for tables and/or stored procedures and then subscribe to them as a group.

With publications, you can monitor the status of one publication subscription for a set of tables and procedures. Publication usage is not a separate model; it provides a grouping technique that can be used by any model.

When you use publications, you create and manage the following objects:

- Articles – replication definition extensions for tables or stored procedures that let you put table or function replication definitions in a publication.
- Publications – groups of articles from the same primary database.
- Publication subscriptions – subscriptions to a publication. When you create a publication subscription, Replication Server creates a subscription for each of the publication's articles.

The following steps summarize the procedure for replicating data using publications.

At the primary site:

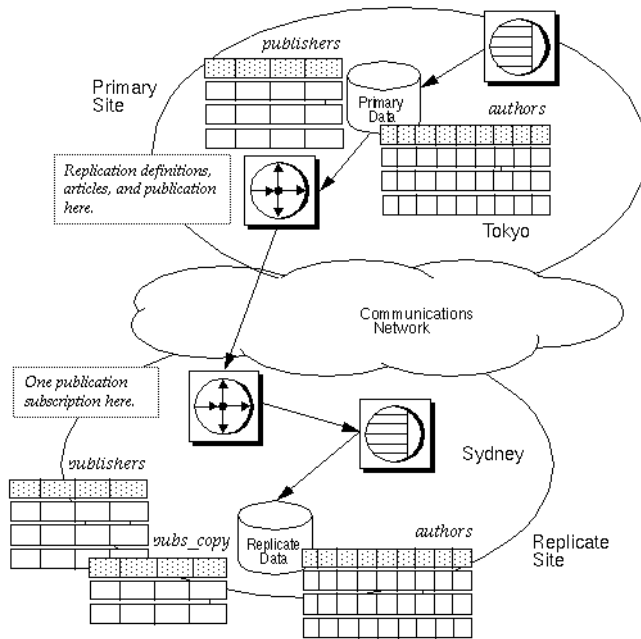
- 1 Create or select the replication definitions to include in the publication.
- 2 Create the publication, using the create publication command.
- 3 Create articles that reference the replication definitions you have chosen, using the create article command.
- 4 Validate the publication using the validate publication command.

At the replicate site:

Create a subscription for the publication using the create subscription command.

In Figure 3-11, a table replication definition `pubs_rep`, referenced by two articles, and a function replication definition, referenced by one article, are collected in the publication `pubs2_pub`.

Figure 3-11: Publications



Stored procedure

This script creates a stored procedure `update_authors_pubs2` that updates the `authors` table in the `pubs2` database. Create the same procedure at the primary and replicate sites.

```
-- Execute this script at the Tokyo and Sydney
-- data servers
-- Creates the stored procedure update_authors_pubs2
create procedure upd_authors_pubs2
(@au_id id,
 au_lname varchar(40),
 au_fname varchar(20),
 phone char(12),
 address varchar(12),
 city varchar(20),
 state char(2),
 country varchar(12),
 postalcode char(10))
```

```

as
update authors
set
  au_lname = @varchar(40),
  au_fname = @varchar(20),
  phone = @char(12),
  address = @varchar(12),
  city = @varchar(20),
  state = @char(2),
  country = @varchar(12),
  postalcode = @char(10)
where au_id = @au_id
go
/* end of script */

```

Function replication definition

This script creates an applied function replication definition at the primary site:

```

-- Execute this script at Tokyo Replication Server
-- Creates the applied function replication definition
-- upd_authors_rep_repdef
create applied function replication definition
upd_authors_rep
with primary at TOKYO_DS.pubs2
with all functions named upd_authors_pubs2
(@au_id id,
 au_lname varchar(40),
 au_fname varchar(20),
 phone char(12),
 address varchar(12),
 city varchar(20),
 state har(2),
 country varchar(12),
 postalcode char(10))
go
/* end of script */

```

Table replication definition

This script creates a table replication definition for the publishers table at the primary Replication Server.

```

-- Execute this script at Tokyo Replication Server
-- Creates replication definitions pubs_rep
create replication definition pubs_rep

```

```
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
 city varchar(20),
 state varchar(2))
primary key (pub_id)
go
/* end of script */
```

Publication

This script creates the publication `pubs2_pub` at the primary Replication Server.

```
-- Execute this script at Tokyo Replication Server
-- Creates publication pubs2_pub
create publication pubs2_pub
with primary at TOKYO_DS.pubs2
go
/* end of script */
```

Articles

This script creates articles for the publication `pubs2_pub` at the primary Replication Server. It creates two articles for the replication definition `pubs_rep`.

```
-- Execute this script at Tokyo Replication Server
-- Creates articles upd_authors_art, pubs_art, and
-- pubs_copy_art
create article upd_authors_art
  for pubs2_pub
with primary at TOKYO_DS.pubs2
with replication definition upd_authors_rep
go

create article pubs_art
  for pubs2_pub
with primary at TOKYO_DS.pubs2
with replication definition pubs_rep
go

create article pubs_copy_art
  for pubs2_pub
with primary at TOKYO_DS.pubs2
```



```
with replication definition pubs_rep
where pub_id >= 1000
go
/* end of script */
```

Validation

This script changes the status of the publication `pubs2_pub` to “valid.”

```
-- Execute this script at Tokyo Replication Server
-- Validates the publication pubs2_pub
validate publication pubs2_pub
with primary at TOKYO_DS.pubs2
go
/* end of script */
```

Subscription

This script creates the subscription `pubs2_pub_sub` for the publication `pubs2_pub`. When this script is run, Replication Server creates article subscriptions for `upd_authors_art`, `pubs_art`, and `pubs_copy_art`.

```
-- Execute this script at Sydney Replication Server
-- Creates publication subscription pubs2_pub_sub
create subscription pubs2_pub_sub
  for publication pubs2_pub
  with primary at TOKYO_DS.pubs2}
with replicate at SF.pubs2
without materialization
go
/* end of script */
```

Request functions

You can use request functions to allow the primary database user to invoke stored procedures on the replicate data. The following section illustrates a system that uses request functions, applied functions, and a local pending table.

An example using a local pending table

The pending table is a design enhancement of applied and request functions that allows clients at a remote site to update central data and see the updates at the remote site before they are returned from the central site. Use this model to implement local update applications.

In this strategy, a client application at a remote site executes a user stored procedure that updates data at the central site using a request function. Changes to the central data are replicated to the remote site via an applied function. A local pending table lets clients at the remote site see updates that are pending at the remote site before the replication system returns the updates.

When a client application executes the user stored procedure at the remote data server, it:

- Causes an associated stored procedure to execute and update data at the primary site
- Enters those updates in the local pending table

When the update succeeds at the central database, it is distributed to the remote sites, including the site where the transaction originated. At the remote site, a stored procedure updates the replicated table and deletes the corresponding updates from the pending table.

To use applied functions, request functions, and a local pending table, you must complete these tasks.

At the remote site:

- Create a pending table in the remote database. Grant appropriate permissions.
- Create a user stored procedure in the remote database that initiates the request function and inserts data updates into the pending table.
- Mark the user stored procedure for replicated function delivery using `sp_setreproc`.
- Grant procedure permissions to the appropriate user.
- Create a user stored procedure in the remote database that updates the remote table and deletes the corresponding update from the pending table. Grant appropriate permissions to the maintenance user.
- Create the request function replication definition for the request function.
- Create a subscription to the applied function replication definition created at the central site.

At the primary site:

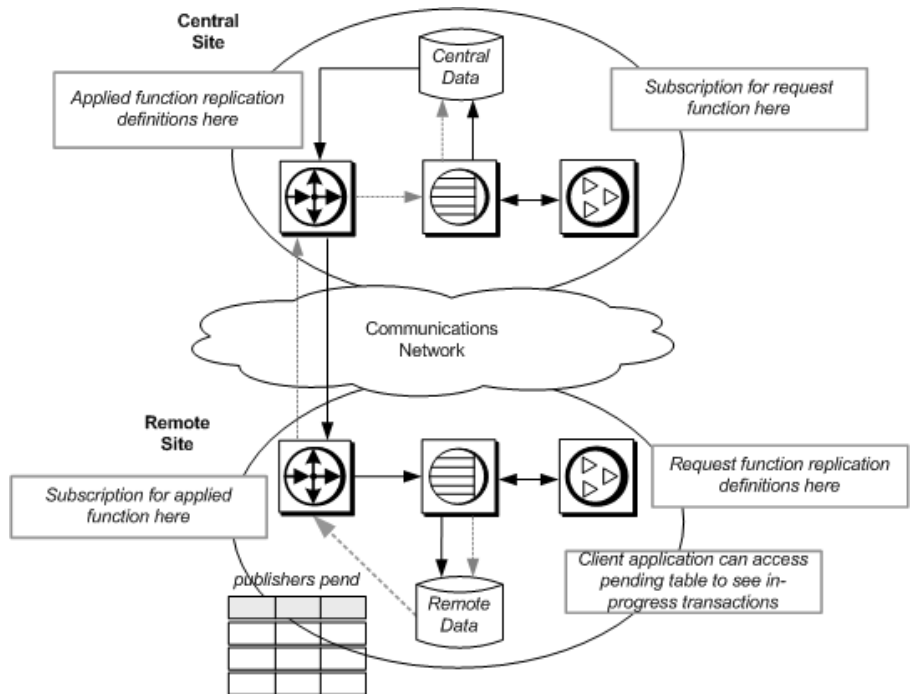
- Create the stored procedure that modifies the central data.

- Create the applied function replication definition for the applied function.
- Create a subscription to the request function replication definition.

In this example, a client application at the remote (Sydney) site executes a stored procedure `upd_publishers_pubs2_req`, which inserts values in the `publishers_pend` table and causes an associated stored procedure, `upd_publishers_pubs2`, to execute at the central (Tokyo) site. Execution of `upd_publishers_pubs2` at the central site causes the stored procedure `upd_publishers_pubs` to execute at the remote site, which updates the `publishers` table and deletes the corresponding information from the `publishers_pend` table.

Figure 3-12 illustrates the data flow when you use applied functions, request functions, and a local pending table. The gray arrows show the flow of the request function delivery. The black arrows show the flow of the applied function delivery.

Figure 3-12: Request functions and a local pending table



Pending table

This script creates a pending table in the remote database.

```
-- Execute this script at Sydney data server
-- Creates local pending table
create table publishers_pend
(pub_id char(4) not null,
 pub_name varchar(40) null,
 city varchar(20) null,
 statechar(2) null)
go
/* end of script */
```

Stored procedures

The script creates the stored procedure upd_publisher_pubs2 at the central (Tokyo) site:

```
-- Execute this script at Tokoyo data server
-- Creates stored procedure
create procedure upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
as
    insert into publishers
        values (@pub_id, @pub_name, @city, @state)
go
/* end of script */
```

The following script creates the upd_publishers_pub2_req stored procedure at the remote (Sydney) site. The insert into clause inserts values into the publishers_pend table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure
create procedure upd_publishers_pubs2_req
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
as
    insert into publishers_pend
        values (@pub_id, @pub_name, @city, @state)
go
/* end of script */
```

This script creates the `upd_publishers_pubs2` procedure for the remote (Sydney) site. It updates the `publishers` table and deletes the corresponding information from the `publishers_pend` table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure upd_publishers_pubs2
create procedure upd_publishers_pubs2
  (@pub_id char(4),
  @pub_name varchar(40),
  @city varchar(20),
  @state char(2))
as
update publishers
set
  pub_name = @pub_name,
  city = @city,
  state = @state
where
  pub_id = @pub_id
delete from publishers_pend
where
  pub_id = @pub_id
go
/* end of script */
```

Function replication definitions

This script creates the applied function replication definition at the central (Tokyo) Replication Server:

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
create applied function replication definition
  upd_publishers_pubs2
with primary at TOKYO_DS.pubs2
  (@pub_id char(4),
  @pub_name varchar(40),
  @city varchar(20),
  @state char(2))
go
/* end of script */
```

This script creates the request function replication definition at the remote (Sydney) Replication Server:

```
-- Execute this script at Sydney Replication Server
-- Creates replication definition
create request function replication definition
```

```
    upd_publishers_pubs2_req
with primary at SYDNEY_DS.pubs2
with primary function named upd_publishers_pubs2_req
with replicate function named upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
go
/* end of script */
```

Subscription

This script creates a subscription at the remote Replication Server using the no-materialization method for the applied function replication definition defined at the central Replication Server:

```
-- Execute this script at Sydney Replication Server
-- Creates subscription using no-materialization
for upd_publishers_pubs2
create subscription upd_publishers_pubs2_sub
for upd_publishers_pubs2
with replicate at SYDNEY_DS.pubs2
without materialization
go
/* end of script */
```

This script creates a subscription at the central Replication Server using the no-materialization method for the request function replication definition defined at the remote Replication Server.

```
-- Execute this script at Tokoyo Replication Server
-- Creates subscription using no-materialization
for upd_publishers_pubs2_req
create subscription upd_publishers_pubs2_req_sub
for upd_publishers_pubs2_req
with replicate at TOKOYO_DS.pubs2
without materialization
go
/* end of script */
```

Implementing master/detail relationships

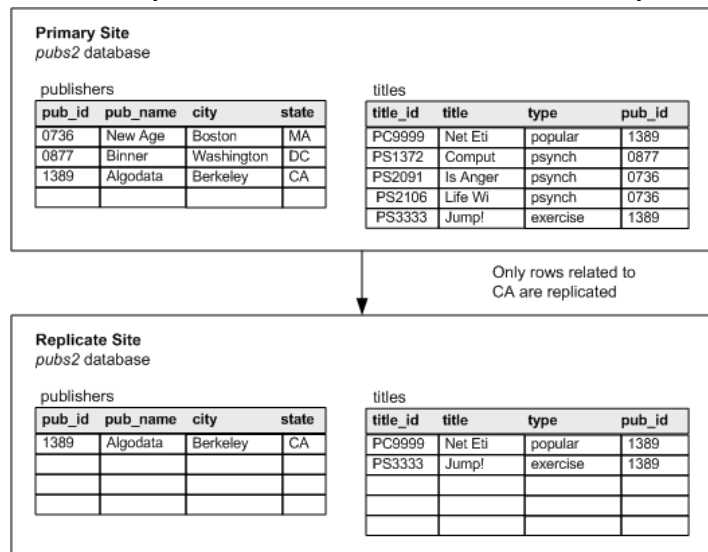
You can use applied functions to replicate only selected data to remote sites. Using applied functions in this way reduces network traffic.

To implement master/detail relationships, use applied functions to support selective subscription to the master/detail tables. In this example,

- The publishers and titles tables exist in the pubs2 database at the primary and replicate sites.
- NY_DS is the primary site data server and SF_DS is the replicate site data server.

Figure 3-13 describes the publishers (master) and titles (detail) tables at the primary and replicate sites:

Figure 3-13: Sample tables used in master/detail relationship



The primary site contains all records, but the replicate site is interested only in records related to the state of California (CA). Only a selection of publishers and titles records need to be replicated, based on the state column. However, only the publishers table contains a state column.

Adding a state column to the titles table adds redundancy to the system. A second, more efficient solution ties updates to master and detail tables through stored procedures and then replicates the stored procedures using applied functions. The logic to maintain selective subscription is contained in the stored procedures.

For example, if, at the primary site, a publisher's state is changed from NY to CA, a record for that publisher must be inserted at the replicate site. Having replicate rows inserted or deleted as a result of updates that cause rows in a subscription to change is called *subscription migration*.

To ensure proper subscription migration, subscriptions are needed for a set of “upper-level” stored procedures that control the stored procedures that actually perform the updates. It is the invocation for the upper-level stored procedure that is replicated from the primary site to the replicate site.

To handle changes in the state column, the replicate site must subscribe to updates when either the new state or the old state is CA.

The sections below list the activities you must perform to enable selective substitution at the replicate Replication Server.

At the primary and replicate sites:

- Create stored procedures that insert records into the publishers and titles tables and an upper-level stored procedure that controls the execution of both insert procedures.
- Create stored procedures that delete records from the publishers and titles tables and an upper-level stored procedure that controls the execution of both delete procedures.
- Create stored procedures that update records in the publishers and titles tables and an upper-level stored procedure that controls the execution of both update procedures.
- Grant appropriate permissions on all upper-level stored procedures.

At the primary site:

- Mark each upper-level stored procedure for replication, using `sp_setrepproc`.
- Create a function replication definition for each upper-level stored procedure.

At the replicate site:

- Create subscriptions to the function replication definitions.

Stored procedures with insert clauses

The insert procedures are identical at the primary and replicate sites. The upper-level stored procedure that controls the insert procedures and the insert procedures observe the following logic:

- A publisher record is inserted only when there is no title ID.
- A title record is inserted only when the publisher exists.

These scripts create the `ins_publishers` and `ins_titles` insert stored procedures and the upper-level stored procedure `ins_pub_title`.

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure ins_publishers
```



```
(@pub_id char(4), @pub_name varchar(40)=null,
city varchar(20)=null, @state char(2)=null)
as
    insert publishers values (@pub_id,
        @pub_name, @city, @state)
/* end of script */

-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure ins_titles
(@title_id tid, @title varchar(80), @type char(12),
@pub_id char(4)=null, @price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime, @contract bit)
as
if not exists (select pub_id from publishers
    where pub_id=@pub_id)
    raiserror 20001 "*** FATAL ERROR: Invalid publishers id ***"
else
    insert titles values (@title_id, @title, @type, @pub_id,
        @price,@advance, @total_sales, @notes, @pubdate, @contract)
/* end of script */

-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure ins_pub_title
(@pub_id char(4), @pub_name varchar(40)=null,
@city varchar(20)=null, @state char(2),
@title_id tid=null, @title varchar(80)=null, @type char(12)=null,
@price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime=null, @contract bit)
as
begin
    if @pub_name != null
        exec ins_publishers @pub_id, @pub_name, @city, @state
    if @title_id != null
        exec ins_titles @title_id, @title, @type, @pub_id, @price,
            @advance, @total_sales, @notes, @pubdate, @contract
end/*
end of script */
```

Stored procedures with delete clauses

The delete procedures are identical at the primary and replicate sites. The upper-level stored procedure that controls the delete procedures and the delete procedures observe the following logic:

- When a record is deleted, all dependent child records are also deleted.
- A publisher record is not deleted when a title record exists.

These scripts create the `del_publishers` and `del_titles` stored procedures and the upper-level stored procedure `del_pub_title`.

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure del_publishers
(@pub_id char(4))
as
begin
if exists (select * from titles where pub_id=@pub_id)
    raiserror 20005 "***FATAL ERROR: Existing titles**"
else
    delete from publishers where pub_id=@pub_id
end
/* end of script */
```

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure /
create procedure del_titles
(@title_id tid, @pub_id char(4)=null)
as
if @pub_id=null
    delete from titles where title_id=@title_id
else
    delete from titles where pub_id=@pub_id
end
/* end of script */
```

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure del_pub_title
(@pub_id char(4), @state char(2), @title_id tid=null)
as
begin
    if @title_id != null
        begin
```

```

        exec del_titles @title_id
        return
    end
if @pub_id != null
    begin
        exec del_titles @title_id, @pub_id
        exec del_publishers @pub_id
    end
end
end
/* end of script */

```

Stored procedures with update clauses

The update procedures differ at the primary and replicate sites.

At the primary site: Update procedures observe the following logic:

- Raise an error on an unknown *pub_id*.
- If a title does not exist, insert one.

These scripts create the `upd_publishers` and `upd_titles` stored procedures and the upper-level stored procedure `upd_pub_title` that controls the execution of `upd_publishers` and `upd_titles`.

Note that the `upd_pub_title` stored procedure has an additional column, `old_state`, that enables replicates to subscribe to rows that migrate.

```

-- Execute this script at NY data servers
-- Creates stored procedure
create procedure upd_publishers
    (@pub_id char(4), @pub_name varchar(40),
    @city varchar(20), @state char(2))
as
if not exists
    (select * from publishers where pub_id=@pub_id)
    raiserror 20005 "***FATAL ERROR: Unknown publishers id**"
else
    update publishers set
        pub_name=@pub_name,
        city=@city,
        state=@state
    where pub_id = @pub_id
end
/* end of script */

```

```
-- Execute this script at NY data servers
-- Creates stored procedure
create procedure upd_titles
(@title_id tid, @title varchar(80), @type char(12),
@pub_id char(4)=null, @price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime, @contract bit)
as
if not exists
(select * from titles where title_id=@title_id)
raiserror 20005 "***FATAL ERROR: Unknown title id**"
else
update titles set
title=@title,
type=@type,
pub_id=@pub_id,
price=@price,
advance=@advance,
total_sales=@total_sales,
notes=@notes,
pubdate=@pubdate,
contract=@contract
where title_id = @title_id
end
/* end of script */

-- Execute this script at NY data server
-- Creates stored procedure
create procedure upd_pub_title
(@pub_id char(4), @pub_name varchar(40)=null,
@city varchar(20)=null, @state char(2)=null,
@title_id tid=null, @title varchar(80)=null, @type char(12)=null,
@price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime=null, @contract bit, @old_state char(2))
as
begin
if not exists (select * from publishers where pub_id=@pub_id)
raiserror 20005 "***FATAL ERROR: Unknown publishers id**"
else
exec upd_publishers @pub_id, @pub_name, @city, @state
if @title_id != null
begin
if not exists
(select * from titles where title_id=@title_id)
exec ins_titles @title_id, @title, @type, @pub_id,
```

```

        @price, @advance, @total_sales, @notes, @pubdate,
        @contract
    else
    exec upd_titles @title_id, @title, @type, @pub_id, @price,
        @advance, @total_sales, @notes, @pubdate, @contract
    end
end
/* end of script */

```

At the replicate site: Update procedures observe the following logic:

- Raise an error on an unknown *pub_id*.
- If title does not exist, insert one.
- Implement correct update migration as shown in Table 3-1.

Table 3-1: Migration strategy for replicate site (CA)

Old state	New state	Update procedure needs to
CA	CA	Update publishers and titles tables normally.
CA	NY	Delete publisher and cascade delete of all titles associated with publisher.
NY	CA	Insert new publisher and title (if any).

These scripts create the `upd_publishers` and `upd_titles` stored procedures and the managing stored procedure `upd_pub_title` that controls the execution of `upd_publishers` and `upd_titles`.

```

-- Execute this script at SF data servers
-- Creates stored procedure
create procedure upd_publishers
(@pub_id char(4), @pub_name varchar(40),
@city varchar(20), @state char(2))
as
if not exists
(select * from publishers where pub_id=@pub_id)
raiserror 20005 "***FATAL ERROR: Unknown publishers id**"
else
update publishers set
pub_name=@pub_name,
city=@city,
state=@state
where pub_id = @pub_id
end
/* end of script */

```

```
-- Execute this script at SF data servers
-- Creates stored procedure
create procedure upd_titles
(@title_id tid, @title varchar(80), @type char(12),
@pub_id char(4)=null, @price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime, @contract bit)
as
if not exists
(select * from titles where title_id=@title_id)
exec ins_titles @title_id, @title, @type, @pub_id,
    @price, @advance, @total_sales, @notes, @pubdate,
    @contract
else
update titles set
title=@title,
type=@type,
pub_id=@pub_id,
price=@price,
advance=@advance,
total_sales=@total_sales,
notes=@notes,
pubdate=@pubdate,
contract=@contract
where title_id = @title_id
end
/* end of script */

-- Execute this script at SF data servers
-- Creates stored procedure
create procedure upd_pub_title
(@pub_id char(4), @pub_name varchar(40)=null,
@city varchar(20)=null, @state char(2),
@title_id tid=null, @title varchar(80)=null, @type char(12)=null,
@price money=null, @advance money=null,
@total_sales int=null, @notes varchar(200)=null,
@pubdate datetime=null,@contract bit, @old_state char(2))
as
declare @rep_state char (2)
begin
select @rep_state=state from publishers
where pub_id=@pub_id

if @old_state = @state
```

```

begin
  exec upd_publishers @pub_id, @pub_name,@city, @state
  if @title_id != null
    exec upd_titles @title_id, @title, @type,
      @pub_id, @price,@advance, @total_sales,
      @notes, @pubdate, @contract
  end
else if @rep_state = @old_state
  begin
    exec del_titles @title_id, @pub_id
    exec del_publishers @pub_id
  end
else if @rep_state = null
  begin
    exec ins_publishers @pub_id, @pub_name, @city,
      @state
    if @title_id != null
      exec ins_titles @title_id, @title, @type,
        @pub_id,@price, @advance, @total_sales,
        @notes, @pubdate,@contract
    end
  end
end
/* end of script */

```

Function replication definitions

Create applied function replication definitions on the primary Replication Server for `ins_pub_title`, `del_pub_title`, and `upd_pub_title`. Note that for inserts and deletes, only state is a searchable column; for updates, `old_state` is also searchable.

```

-- Execute this script at NY data servers
-- Creates replication definition ins_pub_title
create applied function replication definition ins_pub_title
with primary at MIAMI_DS.pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2),
 @title_id varchar(6),
 @title varchar(80),
 @type char(12),
 @price money,
 @advance money,
 @total_sales int,
 @notes varchar(200),

```

```
@pubdate datetime,@contract bit)
searchable parameters (@state)
go
/* end of script */

-- Execute this script at NY data servers
-- Creates replication definition upd_pub_title
create applied function replication definition upd_pub_title
with primary at MIAMI_DS.pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2),
 @title_id varchar(6),
 @title varchar(80),
 @type char(12),
 @price money,
 @advance money,
 @total_sales int,
 @notes varchar(200),
 @pubdate datetime,
 @contract bit,
 @old_state char(2))
searchable parameters (@state, @old_state)
go
/* end of script */
```

Subscriptions

This script creates the subscriptions at the replicate Replication Server using the no-materialization method. Use this method when you don't need to load data at the replicate site.

To ensure proper subscription migration, you must create two subscriptions for upd_pub_title.

```
-- Execute this script at SF data servers
-- Creates subscription for del_pub_title,
ins_pub_title,
    and upd_pub_title
create subscription del_pub_title_sub
for del_pub_title
with replicate at SF_DS.pubs2
where @state='CA'
without materialization
go
```



```
create subscription ins_pub_title_sub
for ins_pub_title
with replicate at SF_DS.pubs2
where @state='CA'
without materialization
go

create subscription upd_pub_title_sub1
for upd_pub_title
with replicate at SF_DS.pubs2
where @state='CA'
without materialization
go

create subscription upd_pub_title_sub2
for upd_pub_title
with replicate at SF_DS.pubs2
where @old_state = 'CA'
without materialization
go
/* end of script */
```


Planning for Backup and Recovery

This chapter describes the tools and methods you can use to return primary and replicate sites to a consistent state after a system component failure.

Topic	Page
Protecting against data loss	89
Preventive measures	90
Recovery measures	94

Protecting against data loss

Replication Server runs in distributed database systems with many other hardware and software components, including Adaptive Servers and other data servers, Replication Agents, LANS, WANs, and client application programs.

Any of these components, including Replication Server, may occasionally fail. Replication Server is a fault-tolerant system, designed with this possibility in mind. During most failures, it waits for the failure to be corrected and then continues its work. When failures require restarting Replication Server or a Replication Agent, the start-up process guarantees that replication is resumed without loss or duplication of data.

Protecting against data loss is the same with a replication system as with a centralized database system. In both cases, there is one definitive version of the data, and you should invest considerable planning and resources to protect it.

In a replication system, if the primary data is protected, all replicate data can ultimately be recovered. A site can replace lost replicate data simply by re-creating its subscriptions.

If replicated transactions are lost at the primary site (as happens, for example, when the primary database is rolled back to a previous dump), consistency may be lost between primary and replicated data.

The backup and recovery methods available in a replication system include preventive and recovery measures. This chapter describes both of them.

Preventive measures include:

- Warm standby
- Hardware data mirroring (hot standby)
- Longer save intervals
- Coordinated dumps

Recovery measures include:

- Subscription initialization
- Subscription reconciliation utility (rs_subcmp)
- Database recovery
- Restoring coordinated dumps

Preventive measures

The following sections describe the preventive measures you can take to protect data in your replication system.

Standby applications

You can protect the data in your replication system by maintaining separate (standby) copies of primary data. Two possible standby methods are:

- *Warm standby application* – a pair of Adaptive Server databases, one of which is a backup copy of the other. Client applications update the active database; Replication Server maintains the standby database by copying supported operations to the active database.
- *Hardware data mirroring* – a form of hot standby application. With the use of additional hardware, data mirroring maintains an exact copy of the primary data by reproducing *all* operations on the primary data.

Comparing methods

In a hot standby application, a standby database can be placed into service without interrupting client applications and without losing any transactions. A hot standby database guarantees that transactions committed on the active database are also committed on the standby. When both databases are up, the active database and the standby database are in sync, and the hot standby database is ready for immediate use.

Alternately, a warm standby application maintained by Replication Server:

- Can be used in environments where data mirroring applications cannot, especially when necessary hardware is not available.
- Tolerates temporary network failures better than some hot standby applications because committed transactions can be stored on the active database, even when the standby database is down.
- Minimizes overhead on the active database because the active database does not need to verify that the databases are in sync.

However, a warm standby application maintained by Replication Server also:

- Requires some interruption of client applications when switching to the standby database.
- May not have executed in the standby database the most recent transactions committed in the active database.

Warm standby

Replication Server's warm standby application is described in detail in Chapter 3, "Managing Warm Standby Applications," in the *Replication Server Administration Guide Volume 2*.

Note Replication Server version 12.0 and later supports Sybase Failover available in Adaptive Server Enterprise version 12.0. Failover support is not a substitute for warm standby. While warm standby applications keep a copy of a database, Failover support accesses the same database from a different machine. Connections from Replication Server to warm standby databases work the same way.

For detailed information about how Failover support works in Replication Server, see "Configuring the Replication System to Support Sybase Failover" in Chapter 7, "Replication System Recovery," and Appendix B, "High Availability on Sun Cluster 2.2," in the *Replication Server Administration Guide Volume 2*.

Hardware data mirroring

To ensure the highest data availability, you can mirror the most critical data in the replication system. Mirroring duplicates I/O operations to maintain two identical copies of the data.

If the active media fails, the standby is brought online instantly. Mirroring all but eliminates the possibility of transaction loss.

The most beneficial places to use mirroring in a replication system are listed here in priority order:

- 1 Primary database transaction logs

Transaction logs store transactions that have not been dumped to tape. If the primary transaction log is lost, transactions must be resubmitted.

- 2 Primary database

A database can be recovered by reloading a previous database dump and subsequent transaction dumps. However, recovering a database that stores primary data also requires recovering or reinitializing the data that has been replicated throughout the enterprise. Extended downtime is often catastrophic for OLTP systems. Mirroring the primary data can prevent this type of catastrophe.

- 3 Replication Server stable queues

Replication Server stores transactions in store-and-forward disk queues called stable queues. It allocates the queues from disk partitions assigned to the Replication Server using the create partition command.

Note create partition makes a partition available to Replication Server. This command replaces the existing add partition command. add partition continues to be supported for backward compatibility. The syntax and usage of the two commands are identical. See the “create partition,” in Chapter 3, “Replication Server Commands” in the *Replication Server Reference Manual*.

The data stored in stable queues is redundant; it originates in the primary database transaction log. However, if a stable queue is lost, Replication Server cannot deliver transactions to replicate sites. As a result, subscriptions at replicate sites must be reinitialized. Mirroring disk partitions protects stable queues and minimizes potential downtime for replicate databases.

4 Replication Server System Database (RSSD)

Recovering from a failure of the RSSD can be a complex process if data such as replication definitions, subscriptions, routes, or function or error classes have been modified since the last backup. Refer to Chapter 7, “Replication System Recovery,” in the *Replication Server Administration Guide Volume 2* for detailed recovery information.

Mirroring the RSSD can prevent system data loss and the necessity of a complex recovery process. If you don’t mirror the RSSD, be sure to back up the RSSD after any RCL operation that changes system data.

Save interval

You can configure a route from one Replication Server to another, or a connection from a Replication Server to a database, so that the Replication Server stores stable queue messages for a period of time after delivery to the destination. This period of time is called the save interval.

The save interval creates a backlog of messages at the source, which allows the replication system to tolerate a partition failure if it is corrected within the save interval. If the stable queues at the destination fail, you can rebuild them and have the source Replication Server resend the backlogged messages.

Refer to the *Replication Server Administration Guide Volume 2* for details.

Coordinated dumps

When a database must be recovered by restoring a backup, replicated data in the affected databases at other sites must somehow be made consistent with the primary data. Replication Server provides a method for coordinating database dumps and transaction dumps at all sites in a distributed system. A database dump or transaction dump is initiated from the primary database. The Replication Agent retrieves the dump record from the log and submits it to the Replication Server so that the dump request can be distributed to the replicate sites. This method guarantees that the data can be restored to a known point of consistency.

A coordinated dump can be used only with databases that store primary data or replicated data, but not both. It is initiated from within a primary database.

Refer to the *Replication Server Administration Guide Volume 2* for instructions on creating coordinated dumps.

Recovery measures

The following sections describe methods for recovering data that is lost after a component failure in a replication system.

Re-creating subscriptions

The primary version of the data is definitive, so all inconsistencies should be resolved in its favor. One way to recover from a failure at a remote site is to re-create the subscriptions. This recovery method is most expedient for small replicated tables. Large subscriptions and primary data failures require other recovery methods.

Subscription reconciliation utility (*rs_subcmp*)

rs_subcmp tests for rows that are missing, orphaned, or inconsistent in a replicate table and corrects the discrepancies. Using *rs_subcmp* may be more appropriate for recovering from minor inconsistencies than a more disruptive recovery procedure such as a coordinated load. See the *Replication Server Reference Manual* for instructions on executing *rs_subcmp*.

Database recovery

When a primary database fails, all committed transactions can be recovered if the database and the transaction log are undamaged. If the database or the transaction log is damaged, you must load a database dump and transaction dumps to bring the database to a known state, and then resubmit the transactions that were executed after the last transaction dump.

When you run Replication Server and Replication Agents in recovery mode, you can replay transactions from reloaded dumps and make sure that all transactions in the primary database are replicated and that no transactions are duplicated. For more information about recovering primary databases from dumps, see Chapter 7, “Replication System Recovery,” in the *Replication Server Administration Guide Volume 2*.

Restoring coordinated dumps

Restoring database or transaction dumps created by the coordinated dump process returns the primary and replicated data to a previous, consistent state.

For more information about the coordinated load procedure, see Chapter 7, “Replication System Recovery,” in the *Replication Server Administration Guide Volume 2*.

Introduction to Replication Agents

This chapter provides an overview of the Replication Agent component of a Sybase replication system. It also provides details about how the RepAgent for Adaptive Server and other Replication Agent products work.

Topic	Page
Replication Agent overview	97
Replication Agent transaction logs	98
Replication Agent products	99

Replication Agent overview

A Replication Agent is a Replication Server client that retrieves information from the transaction log for a primary database and formats it for the primary Replication Server. RepAgent is the Replication Agent component for Adaptive Server.

Replication Agent detects changes to primary data and ignores changes to nonprimary data. Using Log Transfer Language (LTL), a subset of Replication Control Language (RCL), Replication Agent sends changes in primary data to the primary Replication Server, which distributes the information to replicate databases.

The Replication Agent connections status is derived from the status of the Replication Agent thread in the Replication Server, and the status of the Replication Agent process that is extracting data from the primary database. If either the Replication Agent thread or the Replication Agent process is down, the RMS returns a state of “Suspended”. The RMS also returns a description if either of the components is down.

Sybase provides Replication Agent components for other non-ASEe data servers, including DB2 Universal Database, Microsoft SQL Server, and Oracle. See the *Replication Server Heterogeneous Replication Guide* and the Replication Server Options documentation for the databases actively supported by Replication Server.

Replication Agent for DB2 is the Replication Agent component for the OS/390-based DB2 Universal Database. Sybase Replication Agent is the Replication Agent component for DB2 Universal Database (on UNIX and Windows platforms), Microsoft SQL Server, and Oracle.

RepAgent is an Adaptive Server thread. Replication Agent for DB2 is a separate process that resides on the OS/390 host. Sybase Replication Agent is a separate application that resides on a UNIX or Windows host.

A Replication Agent performs these tasks:

- 1 Logs in to the Replication Server.
- 2 Sends a connect source command to identify the session as a log transfer source and to specify the database for which transaction information will be transferred.
- 3 Gets the name of the maintenance user for the database from the Replication Server. RepAgent filters out operations executed by the maintenance user, unless the `send_maint_xacts_to_replicate` or `send_warm_standby_xacts` configuration parameter is set to true.
- 4 Requests the secondary truncation point for the database from the Replication Server. This returns a value, called the *origin queue ID*, that RepAgent uses to find the location in the transaction log where it is to begin transferring transaction operations. The Replication Server has already received operations up to this location.
- 5 Retrieves records from the transaction log, beginning at the record following the secondary truncation point, and formats the information into LTL commands.

Replication Agent transaction logs

Some Replication Agents cannot access the native transaction log of a primary database to acquire the information necessary to replicate transactions.

When a native transaction log cannot be used, the Replication Agent uses its own proprietary transaction log to capture and record transactions in the primary database for replication. Such a Replication Agent generates SQL scripts that run in the primary database to create the Replication Agent transaction log. See the *Replication Agent Administration Guide Volume 1* for more information about the Replication Agent transaction log.

Replication Agent products

Replication Agent products extend the capabilities of Replication Server by allowing non-Sybase database servers to serve as primary database servers in a Sybase replication system.

Sybase offers the following Replication Agent products for non-Sybase databases:

- Replication Agent for DB2
- Sybase Replication Agent

The following sections describe these Replication Agent products.

Replication Agent for DB2

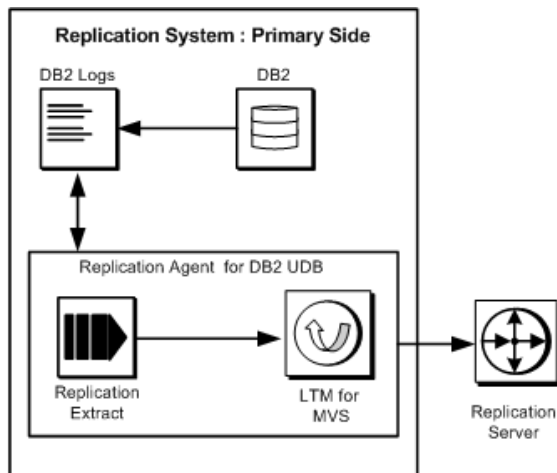
Replication Agent for DB2 is a replication system component that captures database transactions in a DB2 primary database on an OS/390 mainframe platform and sends them to Replication Server.

Figure 5-1 illustrates how Replication Agent for DB2 sends data to Replication Server.

Figure 5-1: Replication Agent data flow for DB2

Replication Agent for DB2 fits into a replication system as follows:

- With Replication Agent for DB2, the primary database is DB2, which runs as a subsystem in OS/390. The transaction logs are DB2 logs.
- Replication Agent for DB2 provides a log extract, called Replication Extract, that reads the DB2 logs and retrieves the relevant DB2 active and archive log entries for tables marked for replication.



- LTM for MVS receives the data marked for replication from Replication Extract and transfers this data to Replication Server using the TCP/IP communications protocol.
- Replication Server then applies the changes to the replicate databases.

DB2 transaction log

The DB2 database server logs changes to rows in DB2 tables as they occur. The information written to the transaction log includes copies of the data before and after the changes. In DB2, these records are known as undo and redo records. Control records are written for commits and aborts. These records are translated to commits and rollbacks.

The DB2 log consists of a series of data sets. Replication Extract uses these log data sets to identify DB2 data changes. Since DB2 writes change records to the active log as they occur, Replication Extract can process the log records immediately after they are entered.

Sybase Replication Agent

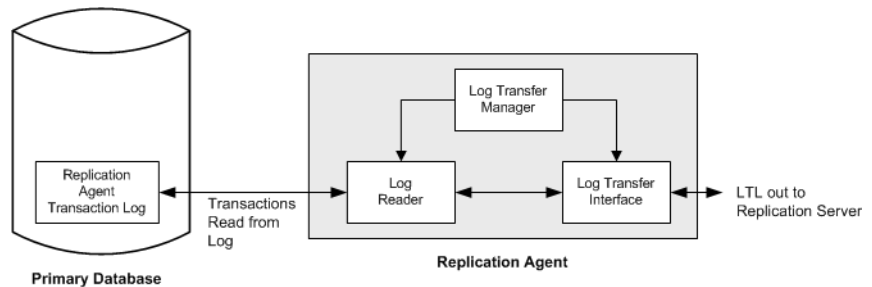
Sybase Replication Agent is a replication system component that captures transactions in a DB2 Universal Database (on UNIX and Windows platforms), Microsoft SQL Server, or Oracle primary database, and then transfers those transactions to Replication Server.

The Sybase Replication Agent for DB2 Universal Database uses the native DB2 transaction log to acquire transaction data to be replicated.

The Sybase Replication Agent for Microsoft SQL Server and Oracle creates its own transaction log to record the transactions (or procedure invocations) to be replicated. The Log Reader component of Replication Agent reads the transaction log to retrieve the transaction from the primary database.

After transaction data is retrieved from the primary database, the Log Transfer Interface (LTI) component of Replication Agent processes the transaction and the resulting “change set” data and generates LTL output, which Replication Server uses to distribute the transaction to the subscribing replicate database(s).

Figure 5-2: Sybase Replication Agent data flow



Sybase Replication Agent uses information stored in the Replication Server System Database (RSSD) of the primary Replication Server to determine how to process the replicated transactions to generate the most efficient LTL.

After it receives LTL from Replication Agent, the primary Replication Server sends the replicated transaction to a replicate database, either directly or by way of a replicate Replication Server. The replicate Replication Server converts the replicated data to the native language of the replicate database, and then sends it to the replicate database server for processing. When the replicated transaction is processed successfully by the replicate database, the replicate database is synchronized with the primary database.

Sybase Replication Agent runs as a stand-alone application, independent of the primary database server, the primary Replication Server, and any other components of a replication system.

Sybase Replication Agent can reside on the same host machine as the primary database or any other component of the replication system, or it can reside on a separate machine from any other replication system components.

Replication Agent communications

Sybase Replication Agent uses the Java Database Connectivity (JDBC) protocol for all its communications.

Replication Agent uses a single instance of the Sybase JDBC driver (jConnect for JDBC) to manage all of its connections to Open Client/Server applications, including the primary Replication Server and its RSSD.

In the case of the primary database server, Sybase Replication Agent connects to the JDBC driver for the primary database.

While replicating transactions, Replication Agent maintains connections with both the primary database and the primary Replication Server. In addition, Replication Agent occasionally connects to the RSSD of the primary Replication Server to retrieve replication definition data.

Java implementation

Sybase Replication Agent components are implemented in the Java programming language. Therefore, to run Sybase Replication Agent, you must have a Java Runtime Environment (JRE) installed on the computer that will act as the Replication Agent host machine.

Replicating Data into Non-Adaptive Server Data Servers

This chapter describes the replication system components required to replicate data into non-ASE data servers.

Topic	Page
Interfacing with non-ASE data servers	103
Sybase database gateway products	104
Maintenance user	105
Function-string class	105
Error class	107
rs_lastcommit table	108
rs_get_lastcommit function	110

Interfacing with non-ASE data servers

Replication Server updates the replicate data stored in databases by submitting requests to data servers. Support for Adaptive Server is provided with Replication Server. If your database is managed by a data server other than Adaptive Server, you must provide an interface for Replication Server to use.

This interface includes:

- Sybase Enterprise Connect™ Data Access (ECDA) to receive instructions from Replication Server and apply them to the data server.
- A maintenance account that Replication Server can use to log in to the gateway.
- A function-string class to use with the database. The function strings in the class tell Replication Server how to format requests for the data server.
- An error class and error action assignments to handle errors the data server returns to Replication Server via the gateway.

- An `rs_lastcommit` table in each database that has replicated data. Replication Server uses this table to keep track of the transactions that have been successfully committed in the database.
- An `rs_get_lastcommit` function call to retrieve the last transaction from each source primary database.

Sybase offers several Open Server gateway application products that you can use to access non-Sybase database servers for a replicate database.

The non-ASE data server support design of Replication Server provides several components of this interface for actively supported database servers. The design provides function string classes, error classes and error actions, user defined datatypes, and connection profiles to create the necessary tables and procedures in the replicate database.

See the *Replication Server Administration Guide Volume 1* and the *Replication Server Configuration Guide* for your platform for more information about the non-ASE support feature. See the *Replication Server Heterogeneous Replication Guide* and the Replication Server Options documentation for the databases actively supported by Replication Server.

Sybase database gateway products

Sybase Enterprise Connect Data AccessConnect provides the Sybase middleware building blocks for connectivity between clients and enterprise data sources. Using ECDA simplifies the integration of non-Sybase replicate databases into a Sybase replication system.

You can use ECDA to connect directly to a database server. The DirectConnect component in ECDA acts as an Open Server gateway by interpreting the Open Client/Server protocol used by Replication Server to the native communication protocol used by the non-Sybase replicate database.

You can use ECDA to connect to:

- Microsoft SQL Server
- OS/390 (UDB and DB2)
- Oracle
- ODBC accessible data sources

For more information, see the Replication Server Options documentation.

Maintenance user

Replication Server logs in as the maintenance user specified in create connection for the database. The gateway can log in to the data server with the same login name, or it can use another login name. The only requirements are that the login name must have the permissions needed to modify the replicate data.

Function-string class

The Replication Server managing a database requires a function-string class. Replication Server provides function-string classes for Adaptive Server, and with its non-ASE data server support features, Replication Server provides function-string classes for all actively supported data servers. See the Replication Server Options documentation for the actively supported data servers.

If you are replicating to a non-ASE data server that is not actively supported by Replication Server, you must create a function-string class for that data server. You can either:

- Create a function-string class that inherits function strings from a system-provided class, or
- Create all the function strings yourself.

Replication Server sends the gateway a command that it constructs by mapping runtime values into the function string supplied for the function. Depending on how the function string is written and the requirements of the data server, your gateway can pass the command directly to the data server or process it in some way before it sends the request to the data server.

Note Replication Server 15.2 and later includes for actively supported databases, connection profiles pre-loaded with function-string classes. See “Connection profiles,” in Chapter 7, “Managing Database Connections” in the *Replication Server Administration Guide Volume 1*.

Refer to Chapter 2, “Customizing Database Operations,” in the *Replication Server Administration Guide Volume 2* for a list of Replication Server system functions that your database gateway may need to process.

Creating function-string classes using inheritance

Replication Server lets you share function-string definitions between function-string classes by creating relationships between classes using a mechanism called *function-string inheritance*.

The system-provided classes `rs_default_function_class` and `rs_db2_function_class` can serve as parent classes for derived classes that inherit function strings from the parent class. You can create a derived class in order to customize certain function strings for your data server while retaining all other function strings from the parent class.

Use the `create function string class` command to create a derived class from the parent class `rs_default_class` or `rs_db2_function_class` that inherits from the parent class. Create customized function strings only as needed.

Note `rs_db2_function_class` does not support replication of text or image data. To enable replication of text or image data for DB2 databases, you must customize the `rs_writetext` function string using the RPC method through a gateway. Refer to “Creating distinct function-string classes” on page 106 for information about `rs_writetext`.

Refer to Chapter 2, “Customizing Database Operations,” in the *Replication Server Administration Guide Volume 2* for a detailed discussion of function-string inheritance.

Creating distinct function-string classes

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

Use the `create function string class` command to create a new function-string class, and then create function strings for all of the functions with function-string-class scope.

You must create `rs_insert`, `rs_update`, and `rs_delete` function strings for each table you replicate in the database.

If you are replicating columns with text or image datatypes, you must create `rs_datarow_for_writetext`, `rs_get_textptr`, `rs_textptr_init`, and `rs_writetext` function strings for each text or image column. The function-string name must be the text or image column name for the replication definition.

The `rs_select` and `rs_select_with_lock` function strings are needed only if the database has the primary data for a replication definition.

Error class

An error class determines how Replication Server handles the errors that are returned by your gateway. You must use the `create error class` command to create an error class for your gateway.

You can define error processing for data server errors with the Replication Server API. You can create an error class for a database and specify responses for each error that the data server returns.

Note Replication Server 15.2 and later includes for actively supported databases, connection profiles pre-loaded with error classes. See “Connection profiles,” in Chapter 7, “Managing Database Connections” in the *Replication Server Administration Guide Volume 1* and “Default non-ASE error classes,” in Chapter 6, “Handling Errors and Exceptions” in the *Replication Server Administration Guide Volume 2*.

Use the `assign action` command to tell Replication Server how to respond to the errors returned by your gateway. Table 6-1 lists the possible actions.

Table 6-1: 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 configure connection. If the error recurs 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 configure connection. 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 suspend connection, and this is the default action. 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 another action.

The default error action is stop_replication. If you do not assign another action to an error, Replication Server shuts down the connection to your gateway.

See the *Replication Server Reference Manual* for more information about create error class and assign action.

rs_lastcommit table

Each row in the rs_lastcommit table identifies the most recent committed transaction that was distributed to the database from a primary database. Replication Server uses this information to ensure that all transactions are distributed.

The `rs_lastcommit` table should be updated by the `rs_commit` function string before the transaction is committed. This guarantees that the table is updated with every transaction Replication Server commits in the database.

Replication Server maintains the `rs_lastcommit` table as the maintenance user for the database. You must make sure that the maintenance user has all of the permissions needed for the table.

Table 6-2 lists the columns in the `rs_lastcommit` table.

Table 6-2: `rs_lastcommit` table structure

Column name	Datatype	Description
<code>origin</code>	<code>int</code>	An integer assigned by Replication Server that uniquely identifies the database where the transaction originated
<code>origin_qid</code>	<code>binary(36)</code>	The origin queue ID for the commit record in the transaction
<code>secondary_qid</code>	<code>binary(36)</code>	A queue ID for a stable queue used during subscription materialization
<code>origin_time</code>	<code>datetime</code>	Time at origin for the transaction
<code>dest_commit_time</code>	<code>datetime</code>	Time the transaction was committed at the destination

The `origin_time` and `dest_commit_time` columns are not required.

The `origin` column is a unique key for the table. There is one row for each primary database whose data is replicated in this database.

If you use a coordinated dump with the database, you should update `rs_lastcommit` with the `rs_dumpdb` and `rs_dumptran` function strings.

For Adaptive Server databases, the `rs_commit`, `rs_dumpdb`, and `rs_dumptran` function strings execute a stored procedure named `rs_update_lastcommit` to update the `rs_lastcommit` table. This is the text of that stored procedure:

```

/* Create a procedure to update the
** rs_lastcommit table. */
create procedure rs_update_lastcommit
    @origin int,
    @origin_qid binary(36),
    @secondary_qid binary(36),
    @origin_time datetime
as
update rs_lastcommit
    set origin_qid = @origin_qid,
        secondary_qid = @secondary_qid,

```

```
        origin_time = @origin_time,
        commit_time = getdate()
where origin = @origin
if (@@rowcount = 0)
begin
    insert rs_lastcommit (origin,
        origin_qid, secondary_qid,
        origin_time, commit_time,
        pad1, pad2, pad3, pad4,
        pad5, pad6, pad7, pad8)
    values (@origin, @origin_qid,
        @secondary_qid,@origin_time,
        getdate(), 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00)
end
go
```

Note Replication Server 15.2 and later creates the `rs_lastcommit` table for actively supported non-ASE data servers when you initially create a connection using a connection profile.

See the reference pages for the `rs_commit`, `rs_dumpdb`, and `rs_dumptran` functions in the *Replication Server Reference Manual* for more information.

***rs_get_lastcommit* function**

Replication Server sends an `rs_get_lastcommit` function call to the gateway to retrieve the last transaction committed in the database from each source primary database. The gateway is expected to return the first three columns described in Table 6-2.

The function string for `rs_get_lastcommit` can execute a simple select:

```
select origin, origin_qid, secondary_qid
    from rs_lastcommit
```

Note The `rs_get_lastcommit` function is pre-defined in connection profiles for actively supported non-ASE data servers.

International Replication Design Considerations

This chapter discusses issues that pertain to setting up a replication system in an international environment.

Topic	Page
Designing an international replication system	111
Message language	112
Character sets	113
Sort order	116
Changing the character set and sort order	121
Summary	125

Designing an international replication system

Replication Server and Replication Manager (RM), the Sybase Central graphical plug-in that you can use to manage your replication system, support international environments. They provide these features:

- Localization of messages into several languages.
- Support for all Sybase-supported character sets, with character-set conversion between Replication Server sites.
- Support for nonbinary sort orders.

When you design a replication system for an international environment, it is important to understand the impact that language, character set, and sort order settings have on your system. Replication Server and the RM provide great flexibility in the configuration of these settings. This flexibility may lead to unexpected or unwanted results unless you follow the configuration guidelines presented in this chapter.

Message language

You can configure Replication Server to print messages to the error log and to clients in several languages. The language you choose must be compatible with the chosen character set. English is the default language; it is compatible with all Sybase character sets. See the *Replication Server Configuration Guide* for your platform for a list of supported languages.

Each server program in your replication system, including Replication Server, Adaptive Server, other data servers, writes messages to its error log in the configured language. However, whether messages are sent to a client in the client's language depends on the server.

For example, Adaptive Server checks for the language setting of its client (Replication Server) and returns messages in that language. RepAgent, an Adaptive Server thread, also returns messages in the client language.

However, Replication Server do not check for a client's language; instead, they return messages to a client in their own language. Thus, error logs can contain messages in different languages if the servers are configured in different languages.

Note To avoid the confusion that can result from a mixed-language error log, configure the same language setting for all servers and clients at a given site.

❖ Changing the Replication Server message language

You can change the Replication Server message language using this procedure.

Note Because RepAgent automatically returns messages in the Replication Server language, you do not have to set a language parameter for RepAgent.

- 1 Shutdown the Replication Server.
- 2 Using a text editor, change the value of RS_language in the Replication Server configuration file.
- 3 Restart Replication Server.

Character sets

Replication Server supports all Sybase-supported character sets and perform character-set conversion of data and identifiers between primary and replicate sites. Character sets must be compatible for character-set conversion to be successful. For example, single-byte character set data cannot be converted to a multibyte character set. For details about character-set compatibility, see the *Adaptive Server Enterprise System Administration Guide Volume 1*.

Your choice of a character set for a given server is influenced by the languages the server supports, the hardware and operating system it runs on, and the systems with which it interacts.

These things are true of Sybase-supported character sets:

- They are all supersets of 7-bit ASCII.
- Some are completely incompatible with each other, meaning that they have no characters in common beyond 7-bit ASCII.
- Among compatible character sets, some characters are not common to both sets—that is, no two character sets have all the same characters.

To change the default character set, follow the procedure provided in “Changing the character set and sort order” on page 121. Although changing the default character set requires only changing the value of the `RS_charset` parameter in the Replication Server configuration file, follow the steps provided in the procedure to ensure that no replicate data is corrupted by the change.

Character-set conversion

Character-set conversion takes place at the destination Replication Server. Every message packed for the Replication Server Interface (RSI) includes the name of the source Replication Server’s character set. The destination Replication Server uses this information to convert the character data and identifiers to its own character set.

When it attempts character-set conversion, Replication Server checks to see whether the character sets are compatible. If they are incompatible, no conversion occurs. If they are compatible and one or more characters that are not common to both sets are encountered, a ? (question mark) is substituted for the unrecognized characters.

In the Replication Server, context determines whether a ? is substituted or a character-set conversion exception is raised. For example, if Replication Server detects an incompatibility in a character being replicated, it substitutes a ? for the character; if it detects an incompatibility when converting the configuration file parameters, it prints an error message and shuts down.

You can use the `dsi_charset_convert` configuration parameter to specify whether or not Replication Server attempts character-set conversion. For a description of this parameter, see the `configure` connection command in the *Replication Server Reference Manual*.

For details about how Replication Server handles character-set conversion during subscription materialization, resolution, and reconciliation, see “Subscriptions” on page 116.

Unicode UTF-8 and UTF-16 support

Replication Server supports the default character set Unicode UTF-8 and three Unicode datatypes, `unichar`, `univarchar`, and `unitext` which are UTF-16 encoded. Unicode allows you to mix different languages from different language groups in the same data server. For more information about using the Unicode character set, see the *Adaptive Server Enterprise System Administration Guide Volume 1*.

UTF-8

UTF-8 (UCS Transformation Format, 8-bit form) is an international character set that supports more than 650 of the world’s languages. UTF-8 is a variable-length encoding of the Unicode standard using 8-bit sequences. UTF-8 supports all ASCII code values, from 0 to 127, as well as values from many other languages. Each nonsurrogate code value is represented in 1, 2, or 3 bytes. Code values beyond the basic multilingual plane (BMP) require surrogate pairs and 4 bytes.

Adaptive Server, Oracle, IBM UDB, and Microsoft SQL Server data servers all support UTF-8.

UTF-16

UTF-16 (UCS Transformation Format, 16-bit form) is a fixed-length encoding of the Unicode standard using 16-bit sequences, where all characters are 2 bytes long. As with UTF-8, code values beyond the BMP are represented using surrogate pairs that require 4 bytes.

Both Replication Server and Adaptive Server encode three character datatype values in UTF-16:

- `unichar` – fixed-width Unicode character datatype.
- `univarchar` – variable-width Unicode character datatype.
- `unitext` – variable-width Unicode large object datatype introduced with ASE 15.0 and Replication Server 15.0. `unitext` can hold up to 1,073,741,823 Unicode characters or the equivalent of 2,147,483,647 bytes.

Requirements

To use the Unicode UTF-8 default character set or the `unichar` and `univarchar` datatypes, you must be running Replication Server version 12.5 or later and have set the site version to 12.5 or later.

The `unitext` datatype will be fully supported if you have a site version and route version of 15.0 or later for the primary and replicate Replication Servers, and the LTL version must be 700. If the LTL version is less than 700 at connect-source time, RepAgent and other Sybase Replication Agents will convert `unitext` columns to image.

Guidelines for using character sets

In setting up a replication system, it is strongly recommended that all servers at a given Replication Server site use the same character set. It is also recommended that all of the Replication Servers in your replication system use compatible character sets.

Follow these guidelines to minimize character-set conversion problems:

- Use 7-bit ASCII (if possible) for all data servers, data, and object names.

If your data and object names are all 7-bit ASCII or if all of your data servers and Replication Servers use the same character set, character set conversion will not present problems.

- If you need to replicate data between a single-byte and a multibyte server, restrict character data and object names to 7-bit ASCII to avoid corruption. Otherwise, you may experience problems. For example, Replication Server does not restrict server names to 7-bit ASCII, but Adaptive Server or the Connectivity Libraries may do so.
- When replicating between servers with different but compatible character sets (for example, ISO_1 and CP850), make sure that object names and character data do not include any 8-bit characters that are not common to both character sets.

Sort order

Replication Server uses sort orders, or collating sequences, to determine how character data and identifiers are compared and ordered. Replication Server supports all Sybase-supported sort orders, including non-binary sort orders. Non-binary sort orders are necessary for the correct ordering of character data and identifiers in European languages.

To change the default or Unicode sort order, follow the procedure provided in “Changing the character set and sort order” on page 121. Although changing the default sort order requires only changing the value of the RS_sortorder parameter in the Replication Server configuration file, follow the steps provided in the procedure to ensure that no replicate data is corrupted by the change.

Note To make sure that data and identifiers are ordered consistently across your replication system, *configure all Replication Server components with the same sort order.*

Subscriptions

Subscriptions involve comparisons of data at the:

- Primary data server during materialization
- Primary Replication Server during resolution
- Replicate Replication Server during initialization and dropping

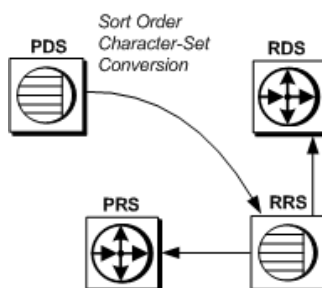
- Replicate data server during dropping

Sort order and character-set conversion play important roles in processing subscriptions and must be consistent everywhere for subscriptions to be valid.

Subscription materialization

Figure 7-1 illustrates the typical message flow during subscription materialization.

Figure 7-1: Subscription materialization



During subscription materialization:

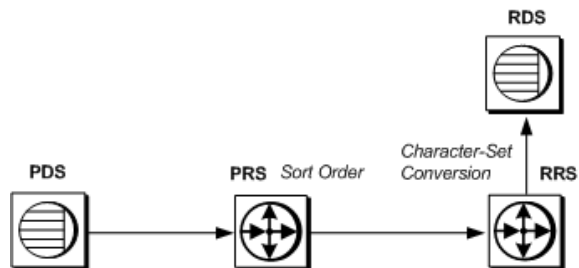
- The replicate Replication Server logs in to the primary data server and issues a `select` statement to retrieve the primary data.
- The primary data server converts all character data to the replicate Replication Server's character set. The replicate Replication Server's character set, if it is different, must be installed at the primary data server.
- The replicate Replication Server inserts the data at the replicate data server.

Note In bulk materialization, a subscription is initialized by a user-chosen mechanism outside the replication system. Therefore, you must make sure that the initial data selection at the primary data server uses the correct sort order and that the character data is converted to the replicate data server character set, if need be.

Subscription resolution

Figure 7-2 illustrates the typical message flow during the normal lifetime of a subscription.

Figure 7-2: Subscription resolution

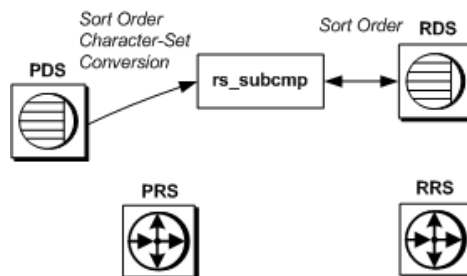


During subscription resolution:

- Adaptive Server RepAgent thread scans the log for updates.
- The primary Replication Server uses its sort order to determine what rows qualify for the subscription. The primary Replication Server also adds the name of the primary Replication Server's character set to the RSI message.
- The replicate Replication Server converts the data to its character set, if necessary, and applies updates to the replicate data server.

Subscription reconciliation

Figure 7-3 illustrates the rs_subcmp process during subscription reconciliation.

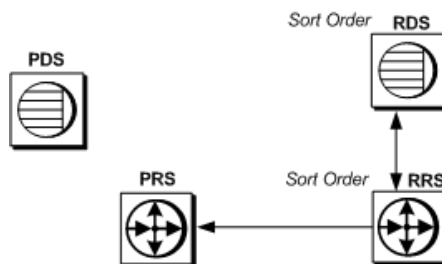
Figure 7-3: Subscription reconciliation

During subscription reconciliation:

- rs_subcmp connects to the primary data server and to the replicate data server using the replicate data server's character set.
- The primary data server converts all character data to the replicate data server's character set (all rs_subcmp operations are performed in the replicate data server's character set). The replicate data server's character set, if it is different from the primary data server's character set, must be installed at the primary data server.
- rs_subcmp sends a select statement to both data servers. The sort order of each data server must be the same for this process to produce sensible results.

Dematerialization

Figure 7-4 illustrates the typical message flow during subscription dematerialization.

Figure 7-4: Subscription dematerialization

During subscription dematerialization:

- The replicate Replication Server selects data from the replicate data server to construct the dematerialization queue. The replicate data server uses its sort order to select the rows.
- The replicate Replication Server uses its sort order to throw out rows belonging to other subscriptions.
- The replicate Replication Server deletes the remaining rows from the replicate database.

Unicode sort order

The Unicode sort order is different from the Replication Server sort order, and must be set independently. To set the Unicode sort order, use a text editor to add the following line to the Replication Server configuration file:

```
RS_unicode_sort_order=unicode_sort_order
```

unicode_sort_order can be any of the Sybase-supported Unicode sort orders listed in Table 7-1. The default value is binary.

Make sure that you suspend the connection to the data server and shut down Replication Server before changing the Unicode sort order.

To change the current sort order, use the procedure described in “Changing the character set and sort order” on page 121.

Table 7-1: Supported Unicode sort orders

Name	Description
defaultml	UTF-16 default ML
altnoacc	CP850 alt: no accent
altdict	cp850 alt: lowercase first
altnocsp	CP850 alt: no case preference
scandict	CP850 Scandinavian dictionary
scannocp	CP850 Scandinavian, no case preference
binary	UTF-16 binary
dict	Latin-1 English dictionary
nocase	Latin-1 English, no case
nocasep	Latin-1 English, no case preference
noaccent	Latin-1 English, no accent
espdict	Latin-1 Spanish dictionary
espnoes	Latin-1 Spanish no case
espnoac	Latin-1 Spanish, no accent
rusnoes	8859-5 Russian, no case
cyrnoes	8859-5 Cyrillic, no case
elldict	8859-7 Greek dictionary
hundict	8859-2 Hungarian dictionary
hunnoac	8859-2 Hungarian, no accents
hunnoes	8859-2 Hungarian, no case
turknoac	8859-9 Turkish, no accent
turknoc	8859-9 Turkish, no case
thaidict	CP874 Thai dictionary
utf8bin	UTF-16 ordering matching UTF-8

You can also specify a Unicode sort order for `rs_subcmp`. See `rs_subcmp` in the *Replication Server Reference Manual*.

Changing the character set and sort order

If you change the character set or sort order of Adaptive Server, you must also change the character set or sort order of:

- Each Replication Server that manages replication for the server

- Each associated RSSD that resides on a separate Adaptive Server

If you change the sort order of Adaptive Server, you must also change the sort order of the replicate Replication Server and replicate data server to ensure that subscriptions are processed consistently.

After changing the character set or sort order, subscription semantics may change. Sort order changes can have obvious consequences. Suppose a subscription contains the clause “where last_name = MacGregor.” If the sort order is changed from dict to binary, for example, “MacGregor” no longer qualifies for sorting.

Synchronize the primary and replication databases

You must make sure that the primary and replicate databases are resynchronized after you change the character set or sort order. Sybase recommends that you use one of these methods:

- Use `rs_subcmp` *after* changing the character set or sort order, or
- Purge all subscriptions *before* changing the character set or sort order, and then rematerialize all subscriptions *after* changing the character set or sort order.

Note Use the purge and rematerialize method if any subscriptions contain character clauses. Only this method ensures that subscriptions with character clauses are resynchronized.

❖ Changing the character set or sort order

All replicated transactions originating from Adaptive Server must arrive at the replicate data server before the character set or sort order is modified. In addition to changing the sort order or character set, this procedure ensures that no data corruption occurs resulting from changing the character set or sort order.

- 1 Identify all Replication Servers and Adaptive Servers that are associated with the primary Adaptive Server—including all RSSDs for associated Replication Servers.

Note *If you are changing the character set:* look up the character set of all Adaptive Servers in the Replication Server domain to ascertain if their character sets must also be changed. Sybase supports alternate character sets for servers in the same domain, but the implication for users is significant.

If you are changing the sort order: Sybase recommends that all data servers in the Replication Server domain share the same sort order. This ensures that data and identifiers are ordered consistently throughout the replication system.

- 2 Quiesce all primary updates and make sure that they have been processed by Replication Server.

Note If you are changing the character set, make sure that there are sufficient empty transactions to span a page in Adaptive Server. This ensures that after the Adaptive Server transaction log is emptied (see step 9), there will be no data still in the old character set.

- 3 Quiesce all associated Replication Servers.
- 4 Shut down all associated Replication Servers, RepAgents, and Replication Agents.
- 5 Change the character set and/or sort order in configuration files for the Replication Servers and, if applicable, for the Replication Agents.
- 6 Follow Adaptive Server procedures for changing the default character set and/or sort order of each associated Adaptive Server. See “Configuring Character Sets, Sort Orders, and Languages” in the *Adaptive Server Enterprise System Administration Guide Volume 1*.
- 7 Shut down all associated Adaptive Servers.
- 8 Start up the associated Adaptive Servers in single-user mode—unless you can guarantee that there will be no activity at the primary and replicate databases.

- 9 Remove the secondary truncation point from each associated Adaptive Server. This step allows Adaptive Server to truncate log records that the RepAgent has not yet transferred to the Replication Server. From the Adaptive Server, enter:

```
dbcc settrunc('ltm', 'ignore')
```

- 10 Truncate the transaction log. From the Adaptive Server, enter:

```
dump transaction db_name with truncate_only
```

- 11 Reset the secondary truncation point. From the Adaptive Server, enter:

```
dbcc settrunc('ltm', 'valid')
```

- 12 Reset the locator value for the primary database to zero. This step instructs Replication Server to get the new secondary truncation point from Adaptive Server and set the locator to that value. From the Adaptive Server, enter:

```
rs_zeroltm data_server, db_name
```

- 13 Shut down and restart Adaptive Server in normal mode.

- 14 Restart the associated Replication Servers.

- 15 Allow RepAgents (or Replication Agents) to reconnect to Replication Servers by resuming log transfer. From the Replication Server, enter:

```
resume log transfer from data_server.db_name
```

- 16 Start up RepAgents.

- 17 Restart replication.

When changing the character set changes the character width

If the character set change involves a change of character width, the stored procedure messages of all databases controlled by the Replication Server must be reloaded. The stored procedure messages are in `$$SYBASE/$SYBASE_REP/scripts/rspmsg1.sql` and `rspmsg2.sql` for UNIX, and `%SYBASE%\%SYBASE_REP%\scripts\rspmsg1.sql` and `rspmsg2.sql` for Windows. The commands to change the character width are described for UNIX and Windows in the following sections.

UNIX

When the change is from a single-byte to a multibyte character set:

```
isql -User_name -Ppassword -Srssd_name -Jeucjis  
< $$SYBASE/$SYBASE_REP/scripts/rsspmsg2.sql
```

When the change is from a multibyte to a single-byte character set:

```
isql -User_name -Ppassword -Srssd_name -Jiso_1  
< $$SYBASE/$SYBASE_REP/scripts/rsspmsg1.sql
```

Windows

When the change is from a single-byte to a multibyte character set:

```
isql -User_name -Ppassword -Srssd_name -Jeucjis  
< %SYBASE%\%SYBASE_REP%\scripts\rsspmsg2.sql
```

When the change is from a multibyte to a single-byte character set:

```
isql -User_name -Ppassword -Srssd_name -Jiso_1  
< %SYBASE%\%SYBASE_REP%\scripts\rsspmsg1.sql
```

When the change is to the UTF-8 character set, install both *rsspmsg1.sql* and *rsspmsg2.sql*

Summary

- Replication Server can be configured to print messages to error logs and to clients in English, French, German, and Japanese. English is the default language.
- Sybase recommends that all servers at a replication site be configured with the same language.
- Replication Server supports all Sybase-supported character sets and sort orders, including non-binary sort orders and the Unicode UTF-8 character set.
- Replication Server performs character-set conversion of data and identifiers between primary and replicate Replication Servers and databases.

- Sybase recommends that all servers at a replication site use the same character set and that all Replication Servers in your system use compatible character sets.
- Sort order plays an important role in processing subscriptions, and it must be consistent everywhere for subscriptions to be valid.

Capacity Planning

This appendix contains information to help you plan the CPU, memory, disk, and network resources you need for your replication system.

Topic	Page
Overview of requirements	127
Data volume (queue disk space requirements)	129
Other disk space requirements	147
Memory usage	149
CPU usage	151
Network requirements	152

Overview of requirements

A replication system consists of Replication Servers, Replication Agents (RepAgent or other Replication Agent), Replication Manager (RM), Replication Monitoring Services (RMS), and data servers.

Warning! As versions of Adaptive Server change, Replication Server capacity planning may change as a consequence of new Adaptive Server transaction log space requirements.

All capacity planning in this chapter assumes that Adaptive Server is using a 2KB page size. If you are using larger page sizes, you must recalculate your space utilization needs to accommodate Adaptive Server's larger page size.

Replication Server requirements

The minimum requirements for each Replication Server are:

- One Replication Agent for the Replication Server System Database (RSSD) if there will be a route from this Replication Server.

- At least one 20MB raw partition or operating system file for the stable queues.
- An Adaptive Server for the RSSD or SQL Anywhere (SA) SA for the ERSSD.

An Adaptive Server for the RSSD must have:

- At least 10MB of free device space for the RSSD database directory
- Another 10MB of free device space for the RSSD transaction log directory

An SA for an ERSSD must have:

- At least 5MB of free device space for the ERSSD database directory
- At least 3MB of free device space for the ERSSD transaction log directory
- Another 12MB of free device space for the ERSSD backup directory
- 20 user connections for the RSSD, in addition to the number of user connections needed by Adaptive Server users. When Replication Server starts up, several threads attempt to read the RSSD at the same time. To accommodate this demand, increase the number of user connections by 20.
- One RSSD user connection for each RM and for each RMS in the replication system and one user connection per data server for each RM process and for each RMS process.
- Two user connections for each database containing primary data.
- One user connection for each replicate-only database.
- At least 512MB of RAM for the Replication Server executable program and all the Replication Agents, plus data and stack memory. (RepAgent is an Adaptive Server thread; it does not require any Replication Server memory.)

Replication Server requirements for primary databases

For each primary database it manages, a Replication Server needs:

- RepAgent thread for Adaptive Server databases or other Replication Agent for non-Sybase databases
- One inbound stable queue

- One outbound stable queue
- One connection to the data server for the Data Server Interface (DSI)

See the release bulletin for more information about Adaptive Server compatibility requirements.

Note If you are using a Replication Agent with a non-Sybase data source, see the appropriate Replication Agent documentation for information about compatibility requirements.

Replication Server requirements for replicate databases

For each replicate database it manages, a Replication Server needs:

- One outbound stable queue
- One connection to the data server for the DSI.

Replication Server requirement for routes

For each direct route to another Replication Server, a Replication Server needs:

- One outbound stable queue

Data volume (queue disk space requirements)

The most significant components in estimating the amount of resources required by the replication system are the volume of the data being replicated and the rate at which it is being updated.

To calculate data volume, you need to know the following things about your replication system:

- The number of sites
- The widths of the rows in replicated tables
- The widths of parameters in replicated functions
- The number of modifications per second

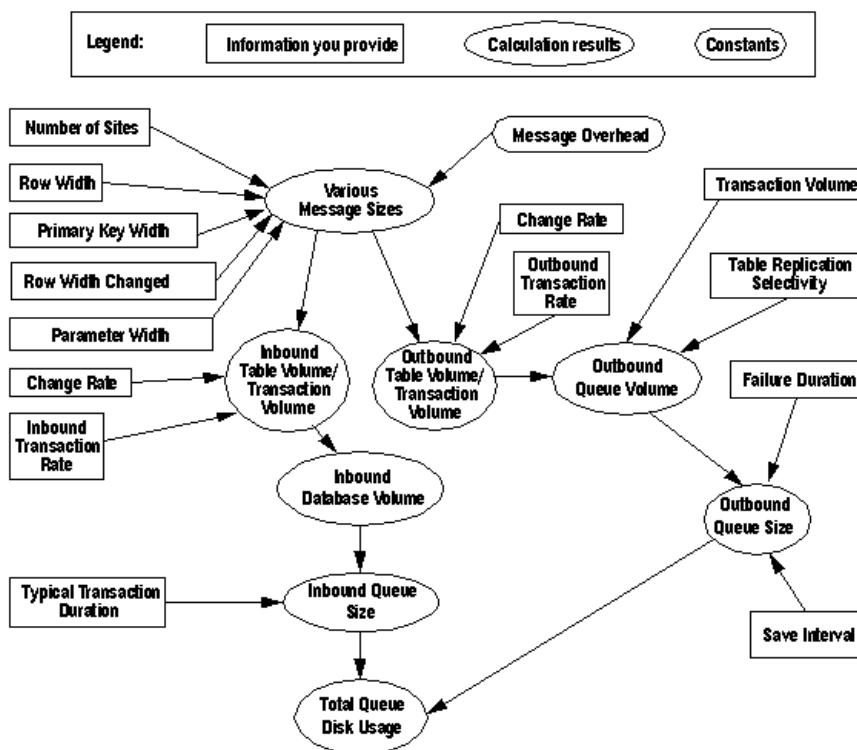
- The duration of a typical transaction
- The selectivity of replication for replicated tables (the fraction of that table replicated through the queue)
- The length of time you want queues to hold transactions when a destination is unavailable

The following sections provide formulas for calculating queue size requirements. You can also use the `rs_fillcaptable` and `rs_capacity` procedures in an RSSD to get a queue size estimate. See the *Replication Server Reference Manual* for information about these stored procedures.

Overview of disk queue size calculation

Figure A-1 illustrates the sequence and flow of calculating data volume and queue disk usage

Figure A-1: Calculating queue disk usage



Message sizes

A Replication Server distributes database modifications using messages in ASCII format. Most replication system messages correspond to the delete, insert, update, and function execution operations.

Each table has one message size for inserts or deletes and another for updates. Each function has its own message size. Message sizes are expressed in bytes.

Message sizes for the same type of modification (insert, delete, or update) may vary depending on whether the message is in the inbound or outbound queue.

You can calculate message sizes in bytes for table updates, inserts, and deletes, for functions, and for begin/commit pairs using the formulas presented below. See “Formula components” on page 133 for a description of the components.

Table updates

This formula will give you a rough estimate of the upper limit of message size:

$$\begin{aligned} \text{InboundMsgSizeupdate} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + (\text{RowWidth} * 2) \\ \text{OutboundMsgSizeupdate} &= \text{OutboundMsgOverhead} + \\ &\quad (\text{RowWidth} * 2) + (\text{NumSites} * 8) \end{aligned}$$

To get a more precise estimate, use the *RowWidthChanged* figure in the calculation, as follows:

$$\begin{aligned} \text{InboundMsgSizeupdate} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + \text{RowWidth} + \text{RowWidthChanged} \\ \text{OutboundMsgSizeupdate} &= \text{OutboundMsgOverhead} + \\ &\quad \text{RowWidth} + \text{RowWidthChanged} + (\text{NumSites} * 8) \end{aligned}$$

If you use the minimal columns feature, use this formula to calculate message size:

$$\begin{aligned} \text{InboundMsgSizeupdate} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + (\text{RowWidthChanged} * 2) + \\ &\quad \text{PrimaryKeyWidth} \\ \text{OutboundMsgSizeupdate} &= \text{OutboundMsgOverhead} + \\ &\quad (\text{RowWidthChanged} * 2) + \text{PrimaryKeyWidth} + \\ &\quad (\text{NumSites} * 8) \end{aligned}$$

See “Table update calculations” on page 143 for examples of these calculations.

Table inserts

Use the following formula to calculate message size for table inserts. (This formula also applies if you use the minimal columns feature.)

$$\begin{aligned} \text{InboundMsgSizeinsert} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + \text{RowWidth} \\ \text{OutboundMsgSizeinsert} &= \text{OutboundMsgOverhead} + \\ &\quad \text{RowWidth} + (\text{NumSites} * 8) \end{aligned}$$

Table deletes

If you do *not* use minimal columns, use this formula to calculate message size in table deletes:

$$\begin{aligned} \text{InboundMsgSizeinsert} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + \text{RowWidth} \\ \text{OutboundMsgSizeinsert} &= \text{OutboundMsgOverhead} + \\ &\quad \text{RowWidth} + (\text{NumSites} * 8) \end{aligned}$$

If you use minimal columns, use this formula to calculate table deletes:

$$\text{InboundMsgSizedelete} = \text{InboundMsgOverhead} +$$

$$\begin{aligned} & \text{ColOverhead} + \text{PrimaryKeyWidth} \\ \text{OutboundMsgSize}_{\text{delete}} &= \text{OutboundMsgOverhead} + \\ & \text{PrimaryKeyWidth} + (\text{NumSites} * 8) \end{aligned}$$

Functions

Use these formulas to calculate message size for functions:

$$\begin{aligned} \text{InboundMsgSize}_{\text{function}} &= \text{InboundMsgOverhead} + \\ & \text{ParameterWidth} + (\text{RowWidth} * 2) \\ \text{OutboundMsgSize}_{\text{function}} &= \text{OutboundMsgOverhead} + \\ & \text{ParameterWidth} + (\text{NumSites} * 8) \end{aligned}$$

In the formula for inbound message size, *RowWidth* does not apply to the replicated functions feature because before and after images of replicated functions are not sent to the inbound queue.

Begin and commit pairs

Use these formulas to calculate message sizes for begins and commits:

$$\begin{aligned} \text{InboundMsgSize}_{\text{begin}} &= \text{OutboundMsgSize}_{\text{begin}} = 250 \\ \text{InboundMsgSize}_{\text{commit}} &= \text{OutboundMsgSize}_{\text{commit}} = 200 \end{aligned}$$

The total size of a begin/commit pair is 450 bytes. If typical transactions have many modifications, omit the begin or commit message sizes from your calculations. Their contribution to overall message size is negligible.

Formula components

In the preceding formulas:

- *InboundMsgOverhead* equals 380 bytes. Each message includes a transaction ID, duplicate detection sequence numbers, and so on.
- *OutboundMsgOverhead* equals 200 bytes. Each message includes a transaction ID, duplicate detection sequence numbers, and so on.
- *ColOverhead*, which applies to the inbound queue only, equals 30 bytes of overhead per column, as follows:

For an update operation without minimal columns:

$$(\text{NumColumns} + \text{NumColumnsChanged}) * 30$$

For an update operation with minimal columns:

$$((\text{NumColumnsChanged} * 2) + \text{NumPrimaryKeyColumns}) * 30$$

For an insert operation (with or without minimal columns):

$$\text{NumColumns} * 30$$

For a delete operation without minimal columns:

$$\text{NumColumns} * 30$$

For a delete operation with minimal columns:

$$\text{NumPrimaryKeyColumns} * 30$$

- *RowWidth* is the size of the ASCII representation of the table columns. For example: a char(10) column uses 10 bytes, a binary(10) column uses 22 bytes.

For table updates, the *RowWidth* is multiplied by 2 because both the before and after images of the row are distributed to the replicate sites.

- *RowWidthChanged* is the width of the changed columns in the row. For example, you have 10 columns with a total *RowWidth* of 200 bytes. Half of the columns in the row change, giving you a *RowWidthChanged* measurement of approximately 100 bytes.
- *NumSites* is the number of sites that a message will go to. This matters only when small messages are distributed to many sites and may be insignificant if you do not have many sites. If small, you might want to omit the number of sites factor from the formulas that use it.
- *ParameterWidth* is the size of the ASCII representation of the function parameters.
- *Begin/Commit Pair* is the combined size of the begin message header and the commit trailer, which equals 450 bytes.

Change rate (number of messages)

The change rate is the maximum number of modifications (inserts, deletes, and updates) made to the table per time period. Change rate varies from table to table.

When planning capacity, always use a change rate greater than the maximum change rate recorded for that table.

Note Figures for change rates are expressed as *operations per unit of time* (for example, 5 updates per second). Always use the same time period when calculating rates. If you measure updates per second, then you must use seconds as the basis for all other time period calculations.

Change volume (number of bytes)

The change volume is the amount of data that is changed in the table per time period. This amount varies from table to table. It is a function of the change rate and data size for the table.

Calculating table volume

The volume of data being replicated is the size of each message times the number of messages replicated per second. If you know the message size of each modification to a table and of each replicated function, you can calculate the total volume of messages generated in a database by summing the volumes of the individual tables and replicated functions.

Table volume upper bound method

For purposes of later calculations, figure table volumes and database volumes for inbound and outbound queues separately. Or, calculate the volume for outbound queues and use that figure as an approximation for both inbound and outbound volumes.

The upper bound for each table can be generated by multiplying the update rate on the table by the size of the longest message. This gives you the upper bound of the table volume.

$$\begin{aligned} \text{InboundTableVolumeupper} &= (\text{Max}[\text{InboundMsgSize}] * \\ &\quad \text{ChangeRate}) \\ \text{OutboundTableVolumeupper} &= (\text{Max}[\text{OutboundMsgSize}] * \\ &\quad \text{ChangeRate}) \end{aligned}$$

where:

- *Max[InboundMsgSize]* and *Max[OutboundMsgSize]* are the sizes of the largest inbound and outbound message in bytes (typically, the message size with the largest parameters).
- *ChangeRate* is the maximum number of modifications to the table per time period.

Table volume sum of values method

A more precise calculation of data volume can be achieved by summing all of the volumes for the different types of messages.

To calculate *InboundTableVolume*:

$$\begin{aligned} \text{InboundTableVolume} &= \\ &(\text{InboundMsgSizeupdate} * \text{ChangeRateupdate}) + \\ &(\text{InboundMsgSizeinsert} * \text{ChangeRateinsert}) + \\ &(\text{InboundMsgSizedelete} * \text{ChangeRatedelete}) + \\ &(\text{InboundMsgSizefunction1} * \text{ChangeRatefunction1}) + \\ &(\text{InboundMsgSizefunction2} * \text{ChangeRatefunction2}) + \\ &\dots \end{aligned}$$

To calculate *OutboundTableVolume*:

$$\begin{aligned} \text{OutboundTableVolume} = & \\ & (\text{OutboundMsgSize}_{\text{update}} * \text{ChangeRate}_{\text{update}}) + \\ & (\text{OutboundMsgSize}_{\text{insert}} * \text{ChangeRate}_{\text{insert}}) + \\ & (\text{OutboundMsgSize}_{\text{delete}} * \text{ChangeRate}_{\text{delete}}) + \\ & (\text{OutboundMsgSize}_{\text{function1}} * \text{ChangeRate}_{\text{function1}}) + \\ & (\text{OutboundMsgSize}_{\text{function2}} * \text{ChangeRate}_{\text{function2}}) + \\ & \dots \end{aligned}$$

where:

- *ChangeRate* is the maximum number of modifications to the table per time period. *ChangeRate_{update}* refers to data updates, *ChangeRate_{insert}* refers to data insertions, and so forth.
- *InboundMsgSize* and *OutboundMsgSize* are the different types of maximum inbound and outbound message sizes. (See “Message sizes” on page 131.)

Transaction volume

TransactionVolume estimates data volume generated by the begin and commit records.

The formula for *InboundTransactionVolume* is:

$$\text{InboundTransactionVolume} = (\text{MsgSize}_{\text{commit}} + \text{MsgSize}_{\text{begin}}) * \text{InboundTransactionRate}$$

where:

- *MsgSize_{begin}* + *MsgSize_{commit}* equals 450 bytes per transaction.
- *InboundTransactionRate* is the total number of transactions per time period at the primary database. This includes all transactions—those that update replicated tables and those that do not. One transaction may include one or more modifications to one or more tables.

The formula for *OutboundTransactionVolume* is:

$$\text{OutboundTransactionVolume} = (\text{MsgSize}_{\text{commit}} + \text{MsgSize}_{\text{begin}}) * \text{OutboundTransactionRate}$$

where:

- *MsgSize_{begin}* + *MsgSize_{commit}* equals 450 bytes per transaction.

- *OutboundTransactionRate* is the total number of replicated transactions per time period replicated through a particular outbound queue. Each transaction may include one or more modifications to one or more tables.

OutboundTransactionRate depends on:

- How many transactions actually update the replicated data
- How many of those transactions are replicated through a particular outbound queue

For example, if the *InboundTransactionRate* is 50 transactions per second, and half of those transactions update replicated tables, and 20 percent of the updates to the replicated tables are replicated through queue 1, the *OutboundTransactionRate* for queue 1 is 5 transactions per second ($50 * 0.5 * 0.2$).

Calculating the *OutboundTransactionRate* can be complicated if transactions contain updates to many different replicated tables with different replication selectivities. In such cases, the *InboundTransactionRate* provides a convenient upper bound for the *OutboundTransactionRate*, and you may use it in your formulas instead of the *OutboundTransactionRate*.

If all transactions in your database are replicated through an outbound queue, then the *OutboundTransactionRate* is the same as the *InboundTransactionRate*.

Database volume

A rough estimate of the upper limit of database volume can be calculated by summing the upper bound message rates for each table.

$$\text{InboundDatabaseVolumeupper} = \text{sum}(\text{InboundTableVolumeupper}) + \text{InboundTransVolume}$$

See “Inbound database volume” on page 144 for an example of these calculations.

A more accurate method of determining the *InboundDatabaseVolume* is to sum the Table Volumes that were calculated using the method described in “Table volume upper bound method” on page 135.

$$\text{InboundDatabaseVolume} = \text{sum}(\text{InboundTableVolume}) + \text{InboundTransactionVolume}$$

Inbound queue size

An inbound queue contains the updates to all of the replicated tables in a primary database. The inbound queue keeps these updates for the duration of the longest open transaction. Queue sizes are expressed in bytes.

To calculate the average size of an inbound queue:

$$\text{InboundQueueSize}_{\text{typical}} = \text{InboundDatabaseVolume} * \text{TransactionDuration}_{\text{typical}}$$

To calculate the maximum size of an inbound queue:

$$\text{InboundQueueSize}_{\text{longest}} = \text{InboundDatabaseVolume} * \text{TransactionDuration}_{\text{longest}}$$

where:

- *InboundDatabaseVolume* is the volume of messages calculated by the formulas described in “Database volume” on page 137.
You can use the *OutboundDatabaseVolume* to calculate the *InboundQueueSize* if you do not need to be precise.
- *TransactionDuration*_{typical} represents the number of seconds for a typical transaction.
- *TransactionDuration*_{longest} represents the number of seconds for the absolute transaction.

See “Inbound queue size example calculation” on page 145 for an example of calculating the maximum size of an inbound queue.

In practice, the maximum size of the inbound queue is slightly longer than the calculation, because while a long transaction is being read, new transactions are being added to the log. Since the duration of the longest transaction is an approximation, when determining its value add an estimate for the short period required to read and process it from the stable queue.

Also note that the size of the queue may be larger if messages trickle slowly into the Replication Server. Replication Server writes messages to stable queues in fixed blocks of 16K. To reduce latency, it writes even partially full blocks every second. (You can use the `init_sqm_write_delay` configuration parameter to change the time delay to something other than 1 second.) If several messages arrive almost simultaneously, all of them will be written into one block. But if the same messages arrive at staggered intervals, each may occupy a block of its own.

Note Since Adaptive Server transaction log requirements may increase with new versions of Adaptive Server, Replication Server stable queue space requirements may also increase.

Outbound queue volume

An outbound queue sends data to another Replication Server or to a data server. The destination is called a Direct Destination Site in these calculations. For each database replicated to (or through) the Direct Destination Site, some part of the Table Volumes passes through the queue. The fraction of a table replicated through a queue is called replication selectivity.

An upper bound of the *OutboundQueueVolume* can be calculated by assuming the highest possible replication selectivity of 1 (or 100 percent) for all tables. Then, the *OutboundQueueVolume* is the sum of all the *OutboundTableVolumes* for all tables replicated through the queue.

$$\text{OutboundQueueVolume}_{\text{upper}} = \text{sum}(\text{OutboundTableVolumes}) + \text{OutboundTransactionVolume}$$

For example, if two tables with *OutboundTableVolumes* of 20K per second and 10K per second and an assumed *OutboundTransactionVolume* of 1K per second are replicated through an outbound queue, the *OutboundQueueVolume* would be:

$$20\text{K}/\text{Sec} + 10\text{K}/\text{Sec} + 1\text{K}/\text{Sec} = 31\text{K}/\text{Sec}$$

A more accurate measure of queue volume can be obtained by factoring in a value for replication table selectivity. The formula for *OutboundQueueVolume* using *ReplicationSelectivity* is:

$$\text{OutboundQueueVolume} = \text{sum}(\text{OutboundTableVolume} * \text{ReplicationSelectivity}) + \text{OutboundTransactionVolume}$$

For example, in the previous example, if only 50 percent of the first table was replicated and 80 percent of the second table was replicated, the

OutboundQueueVolume would be:

$$(20\text{K}/\text{Sec} * 0.5) + (10\text{K}/\text{Sec} * 0.8) + 1\text{K}/\text{Sec} = 19\text{K}/\text{Sec}$$

See “Outbound queue volume example calculation” on page 145 for an example of this calculation.

Failure duration

If the direct destination site is unavailable, the queue buffers its messages. The failure duration figure you use to establish queue size determines how long a failure the queue can withstand.

Save interval

The save interval specifies how long data is kept in outbound queues before it is deleted. The save interval helps in recovering from a disk or node loss.

Use the `configure connection` command to specify the save interval for a database connection. Use the `configure route` command to specify the save interval for a route to another Replication Server. See the *Replication Server Reference Manual* for additional information.

Outbound queue size

The size of an outbound queue is calculated by the formula:

$$\text{OutboundQueueSize} = \text{OutboundQueueVolume} * (\text{FailureDuration} + \text{SaveInterval})$$

where:

- *OutboundQueueVolume* is the amount of data in bytes entering the queue per time period.
- *FailureDuration* is the maximum time the queue is expected to buffer data for an unavailable site.
- *SaveInterval* is the time the messages are kept in the queue before being deleted.

If an outbound queue with an *OutboundQueueVolume* of 18K per second has a *SaveInterval* of 1 hour, and is intended to withstand 1 hour of unavailability at the destination site, the size of the queue should be:

```

OutboundQueueSize =
  18K/sec * (60min + 60min) =
  0.018MB/sec * 120min * 60sec/min = 130MB

```

See “Outbound queue size” on page 146 for a more detailed example of these calculations.

In practice, if the actual change rates and table selectivities result in less than a full 16K block being filled each second, then the figures calculated above are low. Replication Server writes a block every second, whether or not the block is full. The worst usage of queue space occurs when small transactions are executed more than a second apart. Statistics indicate that most blocks are from 50 percent to 95 percent full.

Overall queue disk usage

To calculate the overall total disk space required for worst-case queue usage, use the formula:

$$\text{Sum}[\text{InboundQueueSize}] + \text{Sum}[\text{OutboundQueueSize}]$$

where:

- *Sum[InboundQueueSize]* is the sum of the inbound queue sizes for each of the databases.
- *Sum[OutboundQueueSize]* is the sum of the outbound queue sizes for each direct destination.

Additional considerations

When planning your replication system, take these considerations into account:

- When one of the remote sites is unavailable because of a network failure, it is likely that the other sites on the same network will be unavailable as well, so many queues will grow simultaneously. Thus, you need to allocate enough disk space for all queues to withstand a simultaneous failure of the planned-for duration.
- When the queues start filling up, you can always add more partitions if you have spare disk space.

- If all the queues fill up and the inbound queue cannot accept more messages, the primary database will not be able to truncate its log. If you manually override this restriction, replicate databases will not receive the truncated updates.

Example queue usage calculations

The following sections present a series of example calculations using the preceding formulas.

Examples, calculation parameters

These calculations use the following assumptions about the example replication system:

- Tables T1 and T2 are in database DB1.
- Table T3 is in database DB2.
- All three tables are replicated to sites S1, S2, and S3.
- The parameters are shown in the following tables (note that not all messages are replicated to all sites).

Table A-1: Table parameters

Database	Table	# Cols	Columns changed	Change rate (num/sec)	Row width (bytes)	Row width changed	# Sites
DB1	T1	10	5	20	200	100	3
DB1	T2	10	5	10	400	200	2
DB2	T3	120	100	2	1500	1000	2

Table A-2: Site parameters—table replication selectivity

Table	Percent of updates to S1	Percent of updates to S2	Percent of updates to S3
T1	10%	40%	40%
T2	40%	80%	0%
T3	20%	0%	20%

Table A-3: Transaction rates

Transaction rate	DB1	DB2	S1	S2	S3
Inbound	20/sec	20/sec	-	-	-
Outbound	-	-	5/sec	10/sec	8/sec

Message size example calculations

These examples use the *RowWidthChanged* formula (without minimal columns) for the calculation. See “Message sizes” on page 131 for other formulas that can be used to calculate message size.

Table update calculations

Using the following formulas:

$$\begin{aligned} \text{InboundMsgSize}_{update} &= \text{InboundMsgOverhead} + \\ &\quad \text{ColOverhead} + \text{RowWidth} + \text{RowWidthChanged} \\ \text{OutboundMsgSize}_{update} &= \text{OutboundMsgOverhead} + \\ &\quad \text{RowWidth} + \text{RowWidthChanged} + (\text{NumSites} * 8) \end{aligned}$$

The calculations for updates for each site are:

$$\begin{aligned} \text{InboundMsgSize}_{T1} &= 380 + 450 + 200 + 100 = 1130 \\ &\quad \text{bytes} \\ \text{InboundMsgSize}_{T2} &= 380 + 450 + 400 + 200 = 1430 \\ &\quad \text{bytes} \\ \text{InboundMsgSize}_{T3} &= 380 + 6600 + 1500 + 1000 = 9480 \\ &\quad \text{bytes} \\ \text{OutboundMsgSize}_{T1} &= 200 + 200 + 100 + 24 = 500 \text{ bytes} \\ \text{OutboundMsgSize}_{T2} &= 200 + 400 + 200 + 16 = 800 \text{ bytes} \\ \text{OutboundMsgSize}_{T3} &= 200 + 1500 + 1000 + 16 = 2700 \\ &\quad \text{bytes} \end{aligned}$$

Begin/commit pairs

$$\begin{aligned} \text{MsgSize}_{begin} &= 250 \\ \text{MsgSize}_{commit} &= 200 \end{aligned}$$

Change rate

The change rate is the maximum number of data modifications performed on a table per time period. Naturally, this number is different for every table in your replication system.

These example calculations use the change rates shown in Table A-1 on page 142.

Table volume example calculations

Now calculate the table volumes for tables T1, T2, and T3. For simplicity, what is presented here is a worst-case analysis that assumes all changes are due to SQL update statements plus their associated begin/commit pairs.

Using the upper bound formula for *InboundTableVolume*:

$$\text{InboundTableVolume} = (\text{Max}[\text{InboundMsgSize}] * \text{ChangeRate})$$

The inbound table volumes for tables T1, T2, and T3 are:

$$\text{InboundTableVolumeT1} = 1130 * 20 = 23\text{K/sec}$$

$$\text{InboundTableVolumeT2} = 1430 * 10 = 14\text{K/sec}$$

$$\text{InboundTableVolumeT3} = 9480 * 2 = 19\text{K/sec}$$

Using the upper bound formula for *OutboundTableVolume*:

$$\text{OutboundTableVolume} = (\text{Max}[\text{OutboundMsgSize}] * \text{ChangeRate})$$

The outbound table volumes for tables T1, T2, and T3 are:

$$\text{OutboundTableVolumeT1} = 524 * 20 = 10\text{K/sec}$$

$$\text{OutboundTableVolumeT2} = 816 * 10 = 8\text{K/sec}$$

$$\text{OutboundTableVolumeT3} = 2716 * 2 = 5\text{K/sec}$$

where:

- *InboundMsgSize* and *OutboundMsgSize* are taken from the calculations performed in “Table update calculations” on page 143.
- *ChangeRate* is taken from Table A-1 on page 142.

RSSD and log size example calculations

Each Replication Server has an RSSD with its own inbound queue. The RSSD also contributes to the queue volumes of the outbound queues. However, since very little activity is expected in the RSSD, and all transactions are very small, you can assume that the disk space requirements for an RSSD are minimal:

- Inbound Queue = 2MB
- Outbound Queues = nothing

Inbound database volume

Based on the *InboundTableVolumes* calculated above, you can calculate the *InboundDatabaseVolume* using the upper bound formula:

$$\text{InboundDataBaseVolume} = \text{sum}(\text{InboundTableVolume}) + \text{InboundTransactionVolume}$$

where:

- *InboundTableVolumes* are taken from the calculations on the previous page.
- *TransactionVolume* is the sum of the *Message_{begin}* and *Message_{commit}* pairs (250 bytes + 200 bytes) times the *TransactionRate*.
- In these examples, the transaction rates for each database are taken from Table A-3 on page 142.

Thus, the *InboundDatabaseVolumes* are:

$$\begin{aligned} \text{InboundDatabaseVolumeDB1} &= 23\text{K} + 14\text{K} + \\ &\quad (450 \text{ bytes/tran} * 20 \text{ tran/sec}) = 46\text{K/sec} \\ \text{InboundDatabaseVolumeDB2} &= 19\text{K} + \\ &\quad (450 \text{ bytes/tran} * 20 \text{ tran/sec}) = 28\text{K/sec} \end{aligned}$$

Inbound queue size example calculation

Based on the *InboundDatabaseVolumes*, you can calculate the maximum sizes of inbound queues for DB1 and DB2 using the formula:

$$\text{InboundQueueSize} = \text{InboundDatabaseVolume} * \text{TransactionDuration}$$

where:

- *Inbound DatabaseVolume* is the volume of messages calculated above.
- *TransactionDuration* is the number of seconds for the absolute longest transaction. For the purpose of this example, assume that the *TransactionDuration* for DB1 is 10 minutes, and for DB2 it is 5 minutes.

The *InboundQueue* sizes are:

$$\begin{aligned} \text{InboundQueueDB1} &= 46\text{K/sec} * 600\text{sec} = 28\text{MB} \\ \text{InboundQueueDB2} &= 28\text{K/sec} * 300\text{sec} = 8\text{MB} \end{aligned}$$

Outbound queue volume example calculation

Now calculate the *OutboundQueueVolume* for tables T1, T2, and T3. Because selectivity is measured on a table basis (rather than a database basis), you must calculate outbound queue size for each table replicated through it. The formula for calculating the maximum *OutboundQueueVolume* is:

$$\text{OutboundQueueVolume} = \text{sum}(\text{OutboundTableVolume} * \text{ReplicationSelectivity}) + \text{OutboundTransactionVolume}$$

where:

- *OutboundTableVolume* is taken from the calculations performed in “Calculating table volume” on page 135.
- *ReplicationSelectivity* values are from *Table A-2: Site parameters—table replication selectivity*.
- *TransactionVolume* is the size of the *Message_{begin}* and *Message_{commit}* pairs (250 bytes + 200 bytes) times the *TransactionRate*.

The formula for calculating the *OutboundQueueVolume* for Site 1 is:

$$\begin{aligned} \text{OutboundQueueVolume}_{S1} = & \text{OutboundTransactionVolume} + \\ & (\text{TableVolume}_{T1} * \text{ReplicationSelectivity}_{T1, S1}) + \\ & (\text{TableVolume}_{T2} * \text{ReplicationSelectivity}_{T2, S1}) + \\ & (\text{TableVolume}_{T3} * \text{ReplicationSelectivity}_{T3, S1}) \end{aligned}$$

The *OutboundQueueVolumes* for the three sites are:

$$\begin{aligned} \text{Site1} &= (450 * 5) + (10\text{K}/\text{sec} * 0.1) + (8\text{K}/\text{sec} * 0.4) + \\ & (5\text{K}/\text{sec} * 0.2) = 7\text{K}/\text{sec} \\ \text{Site2} &= (450 * 10) + (10\text{K}/\text{sec} * 0.4) + (8\text{K}/\text{sec} * 0.8) + \\ & (5\text{K}/\text{sec} * 0) = 15\text{K}/\text{sec} \\ \text{Site3} &= (450 * 8) + (10\text{K}/\text{sec} * 0.4) + (8\text{K}/\text{sec} * 0) + \\ & (5\text{K}/\text{sec} * 0.2) = 9\text{K}/\text{sec} \end{aligned}$$

Outbound queue size

The size of an outbound queue is calculated by the formula:

$$\text{OutboundQueueSize} = \text{OutboundQueueVolume} * (\text{FailureDuration} + \text{SaveInterval})$$

where:

- *OutboundQueueVolume* is the amount of data in bytes entering the queue per time period.
- *FailureDuration* is the maximum time the queue is expected to buffer data for an unavailable site. For the purpose of these example calculations, assume that failure duration is set at 4 hours (14,400 seconds).
- *SaveInterval* is the time configured for that particular queue. For the purpose of these calculations, assume that the save interval is 2 hours (7200 seconds).

- *FailureDuration + SaveInterval* is 21,600 seconds.

The *OutboundQueueSizes* for the three sites are:

```
Site1 = 7K/sec * 21,600sec = 151MB
Site2 = 15K/sec * 21,600sec = 324MB
Site3 = 9K/sec * 21,600sec = 194MB
```

Total disk queue usage example calculation

To calculate the total disk space necessary to handle all queues during a worst-case failure (4 hours), use the formula:

$$\text{Sum}(\text{InboundQueueSizes}) + \text{Sum}(\text{OutboundQueueSizes})$$

where:

- *InboundQueueSizes* are the two queues calculated in “Inbound queue size example calculation” on page 145.
- *OutboundQueueSizes* are the three queues calculated above.

The total disk space needed in the worst case, including 2MB for the RSSD inbound queue, is:

$$2\text{MB} + 28\text{MB} + 8\text{MB} + 151\text{MB} + 324\text{MB} + 194\text{MB} = 707\text{MB}$$

Sybase recommends that you allocate enough space for your worst case scenario. If you use the save interval feature (outbound queues are not truncated even after messages are delivered to the next site), you must be sure to allocate enough queue space to sustain your peak transaction activity. If you do not use save interval, then under normal circumstances, your queue utilization will be very small, perhaps 1MB or 2MB per queue.

The example calculations assumed that all outbound queues would have to tolerate the same duration of failure. This assumption may not be true in your environment. Typically, connections across a WAN must tolerate longer duration failures than local connections.

Other disk space requirements

This section covers space requirements for Replication Servers, Replication Agents, and data servers in a replication system.

Stable queues

The calculations for stable queues are explained in the previous sections of this chapter.

RSSD

For the RSSD, allocate at least 10MB for the data and 10MB for the log. These replication system defaults are designed for a relatively small system. If you want sufficient space for hundreds of replication definitions and thousands of subscriptions, you should increase your data and log space to 12MB.

Errors and rejected transactions are also placed in the RSSD. The administrator should periodically truncate the tables that hold these errors and rejected transactions. If you dump stable queues to the RSSD to help diagnose problems, truncate those tables when they are no longer needed.

ERSSD

For the Embedded RSSD (ERSSD), allocate at least 5MB for the data, 3MB for the log, and 12MB for the backup. These replication system defaults are designed for a relatively small system. If you want sufficient space for hundreds of replication definitions and thousands of subscriptions, increase your data and log space to approximately 25MB each.

Errors and rejected transactions are also placed in the ERSSD. The administrator should periodically truncate the tables that hold these errors and rejected transactions. If you dump stable queues to the ERSSD to help diagnose problems, truncate those tables when they are no longer needed.

Logs

Replication Servers write information, error messages, and trace output into their error log files. You should allocate 1MB to 2MB of disk storage for these log files. If you are asked by Sybase Technical Support to turn on trace flags, you may need to make substantially more log space available.

RepAgent writes messages to the Adaptive Server error log file.

Memory usage

Replication Servers are separate processes in the replication system. RepAgent is an Adaptive Server thread.

The following sections estimate the approximate memory requirements of each one.

Replication Server memory requirements

As a general guideline, on a Sun SPARC station, estimate that:

- A newly installed Replication Server uses about 7MB of memory for data and stacks.
- Each DSI connection adds about 500K. This value increases if you configure larger values for MD memory (`md_sqm_write_request_limit` configuration parameter).
- Every RepAgent connection adds 500K. This value increases if you configure larger values for SQT memory (`sqm_max_cache_size` configuration parameter).
- If you have thousands of subscriptions, or if you increase the function string cache size, you must account for that by adding more memory.
- Memory for subscription rules is a function of the columns referenced in the subscription and the number of rules. A typical subscription adds less than 80 bytes plus the combined size of all subscription predicate values.
- Function-string cache size can grow to 200K.
- Remaining system table cache size is based on the number of objects defined. Each replication definition consumes memory equal to approximately 250 times the number of its columns. This might be a factor if you have numerous replication definitions with many columns.

RepAgent memory requirements

Most RepAgent memory comes from allocated Adaptive Server procedure cache (shared memory). It is used for:

- Overhead
- Schema cache

- Transaction cache
- text and image cache

Refer to the *Adaptive Server Enterprise System Administration Guide Volume 2* for information about increasing server memory.

Overhead

Adaptive Server allocates 5612 bytes for each database for log transfer—whether or not RepAgent is enabled.

When RepAgent is enabled, Adaptive Server allocates an additional 2332 bytes of memory at startup for each RepAgent.

Schema cache

The amount of memory used for the schema cache depends of number of objects (tables and stored procedures) and descriptors (columns and parameters) that are replicated. Each object and descriptor requires 128 bytes.

At startup, Adaptive Server allocates 8K of memory—which is sufficient for 64 object/descriptors. Thereafter, memory is allocated in 2K chunks. Adaptive Server also allocates 2048 bytes for the hash table at startup.

A “least recently used” (LRU) mechanism keeps the schema cache size manageable by removing from memory those objects/descriptors not recently referenced. Thus, RepAgent does not need enough memory to describe all replicated objects. At minimum, RepAgent needs enough memory to describe one replicated object.

Transaction cache

RepAgent requires 256 bytes for each open transaction. Transaction cache memory is allocated in 2K chunks. As transactions are committed or aborted, free memory is placed in a pool which must be used before new memory can be allocated.

RepAgent requires memory for the maximum number of open transactions. If sufficient memory is unavailable, RepAgent shuts down.

At startup, Adaptive Server allocates 2048 bytes for a hash table.

text and image cache

The text and image cache is not affected by the size of the text and image data. Rather, the amount of memory used is dependent on the number of replicated tables containing text and image data and the number of columns in the tables that contain text and image data. Each replicated table containing text and image data requires 170 bytes; each replicated column requires 52 bytes.

text and image cache memory is allocated in 2K chunks. Memory is allocated only when replicated tables exist that contain text and image data.

The text and image cache uses a free memory pool and requires sufficient memory for all text and image data.

Other memory

- If RepAgent is enabled, Adaptive Server allocates one extra process descriptor for RepAgent.
- RepAgent uses ct-lib to connect to Replication Server. ct-lib allocates memory directly (dynamic memory) as needed.

CPU usage

Replication Server runs on multiprocessor or single-processor platforms. Replication Server's multi-threaded architecture supports both hardware configurations.

When Replication Server is configured with the symmetric multiprocessor (SMP) feature turned off, Replication Server threads run serially. A server-wide mutually exclusive lock (mutex) enforces serial thread execution ensuring that threads do not run concurrently on different processors.

When Replication Server is configured with the SMP feature turned on, Replication Server threads can run in parallel, thereby improving performance and efficiency. 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.

Replication Server support for multiple processors is based on Open Server support for multiple processors, that is, a single process running multiple threads. Replication Server uses the POSIX thread library on UNIX platforms and the WIN32 thread library on Windows platforms. For detailed information about Open Server support for multiple processor machines, see the *Open Server Server-Library/C Reference Manual*.

Enabling SMP

To enable SMP on a multiprocessor machine, use configure replication server with the `smp_enable` option. For example:

```
configure replication server set smp_enable to 'on'
```

Network requirements

When you have calculated the insert rates into the outbound queues, you can get an idea of the network throughput required. The throughput should allow a Replication Server to send messages faster than messages are being added to the queues. Network messages are usually fractionally larger than what is written to the outbound queues.

Sometimes the queues are filled in bursts and are slowly drained through a potentially low bandwidth network. You should take this behavior into account when you calculate queue sizes and make sure that your queues can handle bursts of incoming data in addition to connection and destination site failures.

Log Transfer Language

This appendix describes the Log Transfer Language (LTL) that a Replication Agent component sends to the primary Replication Server.

Topic	Page
Log Transfer Language overview	153
connect source	154
get maintenance user	157
get truncation	158
distribute	159
Sample RepAgent session	175

Log Transfer Language overview

Table B-1 lists the LTL commands used by replication agents.

Table B-1: Log Transfer Language commands

Command	Description
connect source	Identifies the Replication Server connection as a log transfer session. See “connect source” on page 154
get maintenance user	Retrieves the login name for the maintenance user. See “get maintenance user” on page 157
get truncation	Retrieves the truncation point for the database from Replication Server. See “get truncation” on page 158
distribute	Submits a log record to Replication Server. This command has several subcommands. See “distribute” on page 159.

The first three commands in Table B-1 coordinate the Replication Agent session with the Replication Server. The last command, distribute, submits log records to the Replication Server.

The following sections describe the syntax and use of the LTL commands.

connect source

After logging in to Replication Server, RepAgent sends a connect source command to identify itself. The command specifies the data server and database log RepAgent is forwarding, and the version of the LTL it proposes to use.

Here is the syntax for the connect source command:

```
connect source lti data_server.database version_no
[sendallxacts] [warmstdb] [in recovery]
```

where *data_server* and *database* identify the data server and the database whose log is being forwarded by the RepAgent. The *version_no* parameter identifies the LTL version to be used with the connect source command.

Replication Agents and Replication Server negotiate the version of LTL to use for the session. This ensures that Replication Agents written for earlier versions of Replication Server remain compatible with more recent and current versions. Replication Server and LTL compatibility are listed in Table B-2.

Table B-2: Replication Server and LTL compatibility

Replication Server version	LTL version
15.2	720
15.1	710
15.0	700
12.6	500
12.5	400
12.1	300
12.0	300
11.5	200
11.0	103
10.1.1	102
10.1	101
10.0	100

- If the Replication Agent uses Replication Server version 15.2 features, it must use LTL version 720 and connect to Replication Server version 15.2 or later.
- If the Replication Agent uses Replication Server version 15.1 features, it must use LTL version 710 and connect to Replication Server version 15.1 or later.

- If the Replication Agent uses Replication Server version 15.0 features, it must use LTL version 700 and connect to Replication Server version 15.0 or later.
- If the Replication Agent uses Replication Server version 12.6 features, it must use LTL version 500 and connect to Replication Server version 12.6 or later.
- If the Replication Agent uses Replication Server version 12.5 features, it must use LTL version 400 and connect to Replication Server version 12.5 or later.
- If the Replication Agent uses Replication Server version 12.x features, it must use LTL version 300 and connect to Replication Server version 12.x or later.
- If the Replication Agent uses Replication Server version 11.5 features, it must use LTL version 200 and connect to Replication Server version 11.5 or later.
- If the Replication Agent uses Replication Server version 11.0 features, it must use LTL version 103 and connect to Replication Server version 11.0 or later.
- If the Replication Agent uses Replication Server 10.1.1 features, it must use LTL version 102 and connect to Replication Server version 10.1.1 or later.
- If the Replication Agent uses Replication Server 10.1 features, it must use LTL version 101 and connect to Replication Server version 10.1 or later.
- If the Replication Agent uses only Replication Server 10.0 features, it can use LTL 100 and can connect to any Replication Server.

Keywords

RepAgent uses different keywords with the connect source command, based on the parameters set for RepAgent. These keywords are:

- `sendallxacts` – when RepAgent is started with the `send_maint_xacts_to_replicate` flag, it submits all updates on replicated tables, including updates made by the maintenance user, to the Replication Server for distribution to subscribing replicate sites.

- warmstdb – when RepAgent is started with the send_warm_standby_xacts flag, it submits all updates on replicated tables, including updates made by the maintenance user, to the Replication Server for application to the standby database.
- in recovery – when RepAgent is started with the for_recovery flag it is in recovery mode. When Replication Server is in recovery mode, it permits connections only from RepAgents that are also in recovery mode. Recovery mode is used to replay restored transaction logs so that lost messages can be recovered. For more information about the role of the RepAgent in recovery mode, see Chapter 7, “Replication System Recovery,” in the *Replication Server Administration Guide Volume 2*.

Upgrade locator

In Replication Server version 11.0 and later, connect source returns an additional row that provides the Replication Server system version number and an upgrade locator.

The upgrade locator gives the origin queue ID of the last message written into the inbound queue before the system was upgraded. For details about the origin queue ID, see “Format of the origin queue ID” on page 158.

The upgrade locator is useful when you are creating a Replication Agent, upgrading from a pre-11.0 version of Replication Server to version 11.0 or later, and mixed-mode transactions (both applied and request functions) are in progress.

Example of connect source

The following connect source example identifies the session as a RepAgent session for the Stocks_db database in the NY_DS data server. RepAgent and Replication Server agree to use LTL version 700, for version 15.0 of the Replication Server:

```
connect source lti NY_DS.Stocks_db 700
VERSION
-----
700
Sysversion      UpgradeLocator
-----
1500           0x000000000...
```

get maintenance user

After connecting to Replication Server, RepAgent sends a `get maintenance user` command to find the login name of the maintenance user for the database. Replication Server updates replicated copies of data as the maintenance user. RepAgent uses the login name to distinguish primary data updates from updates distributed through the replication system.

- Changes made by the maintenance user are the result of a distribution, and Replication Server does not redistribute them (except to the warm standby database, if RepAgent is configured to do so using `send_warm_standby_xacts`).
- Changes made to primary data by users other than the maintenance user are primary updates that Replication Server distributes to other databases.

The RepAgent that processes the log of a database containing replicated data must filter out all changes made by the maintenance user.

Note Most replication agents can be run in a mode that does not filter out transactions executed by the maintenance user. This feature is used when a consolidated replicated table is replicated to other sites.

Use this syntax for the `get maintenance user` command:

```
get maintenance user for data_server.database
```

Replication Server returns one row with a `char(30)` column that contains the maintenance user login name for the database.

The following example finds the maintenance user for the `pubs2` database in the `NY_DS` data server:

```
get maintenance user for NY_DS.pubs2_db
```

```
Maintenance_user
-----
pubs2_db_maint
```

get truncation

The `get truncation` command returns a 36-byte binary value called the *origin queue ID*. The format of the origin queue ID is described in the next section.

RepAgent uses the origin queue ID to:

- Locate the last log record saved in the Replication Server inbound queue
- Update the truncation point in the database log

The data server log must be truncated periodically to make room for more log records. RepAgent uses the origin queue ID to update the truncation point and to allow the data server to truncate the log records already received by Replication Server.

Use this syntax for the `get truncation` command:

```
get truncation data_server.database
```

where *data_server* and *database* identify the database whose log is being forwarded by the Replication Agent.

Replication Server returns a single row with one 36-byte column that contains the origin queue ID of the last command saved in its inbound queue. The first 32 bytes of the ID are generated by the Replication Agent. The last 4 bytes are appended by Replication Server for its own use and are ignored by the Replication Agent.

Format of the origin queue ID

The origin queue ID is a unique 32-byte binary string that increases in value as each new log record is transferred. The value must increase, because Replication Server ignores records with an ID lower than the highest ID stored in the inbound queue.

When the Replication Server is restarted, it must be able to map the origin queue ID to the original log record so that it can send the next log record with an increased ID value.

Table B-3 shows the format of the origin queue ID generated by the Adaptive Server RepAgent. A Replication Agent for a different data server can generate origin queue IDs in any format as long as the value increases and can be used to find the original log record. See the *Replication Server Heterogeneous Replication Guide* and Replication Server Options documentation for replication agent information for non-ASE actively supported data servers.

Table B-3: Format of the origin queue ID for Adaptive Server RepAgent

Bytes	Contents
1–2	Database generation number used for recovering after reloading coordinated dumps.
3–8	Log page timestamp for the current record.
9–14	Row ID of the current row. Row ID = page number (4 bytes) + row number (2 bytes).
15–20	Row ID of the begin record for the oldest open transaction.
21–28	Date and time of the begin record for the oldest open transaction.
29–30	An extension used by the RepAgent to roll back orphan transactions.
31–32	Unused.

Bytes 21–28 contain an 8-byte datetime datatype value that is the time of the oldest partially transferred transaction in the database log. Replication Server prints the date and time in this field in a message that helps the Database Administrator locate the offline dumps needed for recovery. If you do not store a valid date in this field, the date and time printed in the message are meaningless. However, the message also contains the entire origin queue ID printed in hexadecimal, so if you put the date and time in a location other than bytes 21–28, the replication system administrator can extract it from the origin queue ID.

distribute

The `distribute` command describes transaction control and data manipulation operations. It also conveys information that Replication Server uses to ensure that transaction log information is transferred without loss or duplication, in the event of failure.

In general, after establishing a session, RepAgent generates a `distribute` command for each operation it retrieves from the database log.

The format of the `distribute` command is:

```
distribute command_tags subcommand [values]
```

The command has three parts:

- The `command_tags` field is used by Replication Server to associate the command with its transaction and to ensure reliable transfer of the database log.

- A *subcommand* name specifies the operation.
- The data *values* associated with the operation are required with all subcommands except commit transaction and rollback transaction.

Command tags

Replication Server normally uses command tags for two purposes:

- To reassemble the commands in a transaction so that data servers at remote sites execute the transaction as a unit
- To ensure that each command is processed only once

The syntax for *command_tags* is:

```
[@origin_time=datetime_value]
@origin_qid=binary_value
@tran_id=binary_value
[@mode=0x08]
[@standby_only={1 | 0}]
```

origin_time

The *origin_time* parameter is a *datetime* value that specifies the time when the transaction or data manipulation operation occurred. It is used to report errors. *origin_time* is used only with the transaction control subcommands: begin transaction, commit transaction, and rollback transaction.

origin_qid

The *origin_qid* parameter is a 32-byte binary value that uniquely identifies the command in the log. It is a sequence number used by Replication Server to reject duplicate commands after a RepAgent connection has been reestablished. The value is generated as shown in Table B-3 on page 159.

tran_id

The *tran_id* parameter is a 120-byte binary value that identifies the transaction the command belongs to. The transaction ID must be globally unique. One way to guarantee this is to first construct a unique transaction ID for the database log, and then attach the data server name and database name to it.

For example, RepAgent constructs a *tran_id* in the following format:

tran_id.data_server.database

where *tran_id* is a value generated by the RepAgent from information in the log. It contains the generation number, log page timestamp, and the row number of the log record. It is guaranteed to be unique within the database. *data_server* and *database* identify the database whose log is being transferred by the RepAgent.

mode

The mode parameter is set if the owner name is to be used when Replication Server looks up replication definitions. This parameter is optional for applied commands. It should not be set if the owner name is unavailable.

mode is an LTL version 200 parameter; it is available with Replication Server version 11.5 or later.

standby_only

The standby_only parameter determines whether the command is sent to the standby and/or replicate databases. If standby_only is set to 1, the command is sent to the standby database and not to the replicate database. If standby_only is set to 0, the command is sent to the standby and replicate databases.

standby_only is an LTL version 200 parameter and is available with Replication Server version 11.5 or later. It is optional for applied commands.

Transaction-control subcommands

The transaction-control subcommands are begin transaction, commit transaction, and rollback transaction.

begin transaction

The begin transaction subcommand has the following syntax:

```
distribute command_tags begin [system] transaction  
[tran_name] [for 'user'/'password' | no_password]]
```

- For the syntax and description of *command_tags*, see “Command tags” on page 160.

- The system keyword tells Replication Server not to apply this transaction inside begin transaction/commit transaction pairs. In Adaptive Server, it is used for transactions started internally or started in system stored procedures. For LTL version 200 or later, system is available with Replication Server version 11.5 or later.
- *tran_name* is an optional varchar(30) value that identifies the transaction. The transaction name does not have to be unique. Replication Server makes the transaction name available to function strings in a system-defined parameter.
- *user* and *password* are varchar(30) values that identify the login name and password of the user executing the transaction. *user* and *password* should be enclosed in quotes. They are both required for asynchronous procedure calls submitted from a non-primary site.

For LTL version 101 or later, *password* is optional and can be omitted when *user* is supplied. For LTL version 100, *password* must be supplied if *user* is supplied.

Use the `no_password` option when the primary database employs a “unified login” or when the user on the primary database has set a proxy. In both cases RepAgent does not recognize a user password. For LTL version 200 or later, `no_password` is available with Replication Server version 11.5 or later.

commit transaction, rollback transaction, and rollback

After RepAgent has submitted all of the commands in a transaction to the Replication Server, it sends either a commit transaction, rollback transaction, or rollback command.

Use this syntax for commit transaction, rollback transaction, and rollback:

```
distribute command_tags
{commit transaction |
rollback transaction |
rollback [from oqid] to] oqid}
```

Values are not used by commit transaction and rollback transaction.

The rollback subcommand, without the transaction keyword, requires specification of origin queue ID (*oqid*) values. The three possible forms of this subcommand are:

- `rollback oqid` – rolls back a single log record corresponding to the specified origin queue ID. This option supports the mini-rollback capability in DB2.

- rollback to *oqid* – rolls back all log records between the specified origin queue ID and the current log record.
- rollback from *oqid1* to *oqid2* – rolls back a sequence of log records whose origin queue IDs fall in the specified range.

Note Replication Server ignores rolled back transactions received from a RepAgent.

applied subcommand

The applied subcommand describes operations recorded in the database, including:

- Row updates
- Row inserts
- Row deletes
- Executions of applied stored procedures (request stored procedures use the execute subcommand)
- Modifications to text or image data
- Truncate table or partition

The syntax for the applied subcommand is:

```
distribute command_tags applied [owner=owner_name]
{table'.rs_update
  yielding before param_list after param_list |
'table'.rs_insert yielding after param_list |
'table'.rs_delete yielding before param_list |
'table'.function_name [param_list]
  yielding after param_list before param_list |
'table'.rs_datarow_for_writetext
  yielding datarow column_list |
'table'.rs_writetext
  append [first] [last] [changed] [with log]
  [textlen=100] column_list}
'table'.rs_updatetext
  {partialupd | _pu} [{first | _fi}] [last] [{changed | _ch}] [with log]
  [{withouttp | _wo}]
  [{offset | _os}=offset {deletelen | _dln}=deletelength]
  [{textlen | _tl}=length] text_image_column |
'table'.rs_truncate [partition_name[, partition_name]...] yielding}
```

- For syntax and description of *command_tags*, see “Command tags” on page 160.
- *table* is the name of the database table to which the operation was applied. It must be enclosed in quotation marks.
- Replication Server uses *table* to associate the command with a replication definition. Beginning with Replication Server version 11.5 and version 200 LTL, if the tag `@mode=0x08` is set, Replication Server also associates the owner name with the replication definition. The create replication definition command’s `with all tables named table_identifier` clause determines how *table* is mapped to a replication definition:
 - If the replication definition has a `with all tables named table_identifier` or `with primary table named table_identifier` clause, *table* above is matched to the *table_identifier* or with the primary table named.
 - If the `with all tables named table_identifier` clause and the `with primary table named table_identifier` clauses were omitted, then *table* above is the name of the replication definition.

RepAgent does not need to be aware of replication definitions. It can use the table name on the data source.

yielding clause

For `rs_update`, `rs_insert`, and `rs_delete`, the yielding clause introduces before and after images of the row affected by the operation. Depending on the operation, the before image, the after image, or both, must be provided. For `rs_truncate`, the yielding clause is empty. Table B-4 shows which operations require before and after images:

Table B-4: Applied subcommand before and after images

Operation	Before image	After image
rs_update	Yes	Yes
rs_insert	—	Yes
rs_delete	Yes	—

The *table.function_name* form of the applied subcommand is used to distribute replicated stored procedures when you use the method associated with table replication definitions. This method is described in Appendix A, “Asynchronous Procedures,” in the *Replication Server Administration Guide Volume 2*.

Note The preferred method for replicating stored procedures, which uses applied and request functions, is described in Chapter 10, “Managing Replicated Functions,” in the *Replication Server Administration Guide Volume 1*. This method uses the `execute` subcommand to distribute replicated stored procedures (known as replicated functions).

If the stored procedure execution results in an insert or delete operation, RepAgent converts it to an `rs_insert` or `rs_delete` LTL command. If the execution results in an update operation, RepAgent uses the *function_name* form and supplies the before and after images of the updated row to the Replication Server.

A Replication Server function with the same name and parameters as the stored procedure is defined with the `create function` command, and the *function_name* in the applied command references this function. The *param_list* following the function name is the list of parameters of the stored procedure.

The yielding clause contains before and after images of the table row modified by the function. Subscriptions on that table determine where the function is distributed.

Before and after images are specified by a *param_list*, which is a list of column or parameter values. The syntax for *param_list* is:

```
[@param_name=]literal[, [:@param_name=]literal]...
```

- *param_name* is a column name or, for replicated stored procedures, a parameter name.
- *literal* is the value of the column or parameter.

All column names in the replication definition must appear in the list. Replication Server ignores any additional columns. Column or parameter names can be omitted if the values are supplied in the same sequence as they are defined in the replication definition. If the column names are included, you can list them in any order, although there is a performance advantage if the columns are supplied in replication definition order.

Replication Server version 10.1 and later supports an optimized yielding clause. An after image value can be omitted if it is the same as the before image value. For example, if a table has three columns a, b, and c, for an update where only column b changes, the yielding clause could be:

```
yielding before @a=5, @b=10, @c=15 after @b=12
```

Note If the minimal columns feature is used, a RepAgent using LTL version 101 or later *must* omit identical after images. See the create replication definition command in the *Replication Server Reference Manual* for more information about replicating minimal columns.

Modifications to text or image data

The `rs_datarow_for_writetext` and `rs_writetext` forms of the applied subcommand are used to distribute modifications to text or image data. These subcommands are built on the Replication Server version 10.1 performance feature of packing data as structured tokens. Each text or image column has a special character and length field, followed by the actual data value. Packing the data in this way eliminates the need for Replication Server to interpret every byte of data, which provides performance benefits.

`rs_datarow_for_writetext` carries an image of the data row associated with a text or non-image column that has been modified by the Transact-SQL `writetext` command, by the Client-Library™ function, `ct_send_data`, or by the DB-Library™ functions `dbwritetext` and `dbmoretext`. The image is used by Replication Server to construct the primary key for subsequent modification at the replicate database. The syntax for `rs_datarow_for_writetext` is:

```
distribute command_tags applied  
  'table'.rs_datarow_for_writetext  
  yielding datarow column_list
```


yielding datarow *column_list* carries the column names, the values of non-text or image columns, and the replication status of text or image columns. The replication status can be *always_rep*, *rep_if_changed*, or *never_rep*. The *column_list* field also carries additional information, called *text_status*, about the text or image columns. The *text_status* can be one of the following keywords:

Table B-5: Text_status values for text and image data

Keyword	Description
tpnull	The column has a null text pointer. There are no modifications to text or image columns.
tpinit	Modifications were made at the primary database, which caused a text pointer allocation.
hastext	The current text or image data value follows.
notrep	The text or image column is not replicated. No commands are required in the replicate database because the data did not change value and the text or image column has a <i>replicate_if_changed</i> status.
zerolen	The text or image column contains a null value after an operation at the primary database. For example, after a text pointer has been allocated, there may be data values in a text or image column and an application at the primary database sets them to null.

rs_insert and *rs_update* also carry the replication status and additional *text_status* information for text and image columns. *rs_delete* carries the replication status only.

The *rs_writetext* form of the applied subcommand carries the text or image data. *rs_writetext* can carry up to 4K of the text or image data, so the data can be segmented and carried in multiple *rs_writetext* iterations. The syntax for *rs_writetext* is:

```
distribute command_tags applied
'table'.rs_writetext
append [first] [last] [changed] [with log]
[textlen=100] column_list
```

- *append* indicates that there are more segments of text or image data to follow.
- *first* marks the first segment of data for the text or image column.
- *last* marks the last segment of data for the text or image column.

- changed indicates that the text or image column changed value. If the changed keyword is omitted, the text or image value did not change. This flag is used by the minimal columns feature to discard the data after Replication Server determines it is not needed.
- with log indicates that the modification is logged at the primary database transaction log. It is required only in the first segment of text or image data.
- textlen indicates the total size of the text or image column. It is required only in the first segment of text or image data.
- *column_list* contains the column name, followed by the text or image data. The data begins with a token header, which is constructed in this order:
 - a The tilde (~) character, which denotes a structured token.
 - b The period (.) character, if it is text data, or the slash (/) character, if it is image data.
 - c The 3-byte character representation of the length of the text or image data segment being carried in an rs_writetext command. The 3-byte length is calculated by converting the length into base-64 representation, then adding the base character ! to each digit to ensure it is a printable character. The formula for calculating the base-64 number is the resultant 3 digits in d3, d2, and d1:

$$d3 = \text{len} / (64 * 64)$$

$$\text{len} = \text{len} - (d3 * 64 * 64)$$

$$d2 = \text{len} / 64$$

$$d1 = \text{len} \% 64$$

For example, the length of the text segment is 126, and d3= 0, d2=1, and d1=62. The base character ! is added to the digits (the integer value for the base character is 33), and they become !, ", and _ (0+33, 1+33, and 62+33). Thus, the structure token header is represented as:

~.!"_
- Following is an example of the LTL commands generated from the writetext command to update a text column called blurb:

```

distribute
  @origin_qid=0x00010000010000,
  @tran_id=0x00018238,
  applied 'textttest'.rs_datarow_for_writetext
  yielding datarow @title_id='BU1032', @price=$90.00,
    @blurb=hastext always_rep,
    @picture = hastext always_rep
distribute
  @origin_qid=0x00010000010001,
  @tran_id=0x00018238,
  applied 'textttest'.rs_writetext
  append first last changed with log textlen = 126
  @blurb = ~.!"_ Straight Talk About Computers is an
  annotated analysis of what computers can do for
  you: a no-hype guide for the critical user
distribute
  @origin_qid=0x00010000010002,
  @tran_id=0x00018238,
  applied 'textttest'.rs_writetext
  append first with log textlen = ...
  @picture = ~/&*^ '0X010203...'
distribute
  @origin_qid=0x00010000010003,
  @tran_id=0x00018238,
  applied 'textttest'.rs_writetext
  append last @picture = ~/!( * 0x4990...

```

Partial update of LOB datatypes

The `rs_updatetext` form of the `applied` subcommand supports the partial update of large object datatypes. Partial update lets you directly insert a character string or overwrite an existing character string of a table column without issuing `delete` and `replace` commands, as would happen in a full update.

```

rs_updatetext syntax      {distribute | _ds} command_tags {applied | _ap} 'table'.rs_updatetext
                          {partialupd | _pu} [{first | _fi}] [last] [{changed | _ch}] [with log]
                          [{withouttp | _wo}]
                          [{offset | _os}=offsetvalue {deletelen | _dln}=deletelength]
                          [{textlen | _tl}=length] text_image_column

```

- Parameters
- `partialupd` or `_pu` indicates that there is one segment of LOB data for partial update.
 - `first` or `_fi` marks the first segment of data for the LOB column.
 - `last` marks the last segment of data for the LOB column.

- changed or `_ch` indicates that the LOB column changed value. Omit the changed keyword if the LOB value did not change. The minimal columns feature uses this flag to discard data after Replication Server determines it is not needed.
- with log indicates that the modification is logged at the primary database transaction log. with log is required only in the first segment of the LOB data.
- withouttp or wo indicates that the datatype is an LOB datatype without a text pointer such as `varchar(max)`, `nvarchar(max)`, and `varbinary(max)`.
- offset or `_os` indicates the starting point in the value of LOB column at which the partial update is performed. *offsetvalue* is a zero-based integer and cannot be a negative number. offset is required only in the first partial update command.
- deletelen or `_dln` indicates the length of the section in the LOB column, starting from offset, that is to be replaced. *deletelength* is a zero-based integer and cannot be a negative number. deletelen is required only in the first partial update command.
- textlen or `_tl` indicates the length of the LOB data that is to be inserted into the LOB column. The value of textlen can be smaller than or equal to the the new length of the LOB column.

Examples

A partial update transaction that contains the commands begin transaction, `rs_update`, `rs_updatetext`, and commit transaction:

```
distribute @origin_qid=~ ,A{0x}0000000036e
800000011000700000007000036e8000000110001000000100002,
@origin_time=~*620080317 18:01:40:653,@tran_id=~ , ;
{0x}72616d6c696e647361792e6d7332303035766d31000486020
000 begin transaction for ~"(qafuser osid 52
distribute @origin_qid=~ ,A{0x}0000000036e
800000011000700000008000036e8000000110001000000100002,
@origin_time=NULL,@tran_id=~ , ;{0x}72616d6c696e6473617
92e6d7332303035766d31000486020000 applied
owner=~"$dbo ~"+qaf_oldlob.rs_update yielding
before ~$%pkey=1,~$)text_col=hastext always_rep
,~$*ntext_col=hastext always_rep ,~$*image_col=hastext
always_rep_isbinary after
distribute @origin_qid=~ ,A{0x}0000000036e
800000011000700000009000036e8000000110001000000100002,
@origin_time=~*620080317 18:01:40:653,@tran_id=~ , ;
{0x}72616d6c696e647361792e6d7332303035766d31000486020
000 applied owner=~"$dbo ~"+qaf_oldlob.rs_updatetext
partialupd first last changed with log offset=10
```

```

deletelen=10 textlen=20~$*ntext_col=~8!!5
{0x}00620062006200620062006200620062006200620062
distribute @origin_qid=~ ,A{0x}0000000036e
80000001100080000000a000036e8000000110001000000100002,
@origin_time=NULL,@tran_id=~ , ; {0x}72616d6c696e6473617
92e6d7332303035766d31000486020000 commit transaction

```

Limitation

- rs_updatetext does not support multiple character set conversion.
- Partial update support is restricted to Microsoft SQL Server 2005.

Truncate table or partition

The rs_truncate form of the applied subcommand is used to support the replication of the truncate table and truncate table partition commands. The syntax for rs_truncate is:

```

distribute command_tags applied [owner=owner_name]
'table'.rs_truncate [partition_name [, partition_name]...] yielding

```

You must assign a *partition_name* for each partition specified in the truncate table partition command.

Note RepAgent sends an rs_truncate applied subcommand with parameters only if the LTL version is 700. If the LTL version is below 700, RepAgent skips the distribute command.

execute subcommand

The execute subcommand is used to send a replicated function or stored procedure call to another Replication Server. This subcommand is used with the preferred method for distributing stored procedures—applied and request functions—and with the older method—request stored procedures.

This is the syntax for the execute subcommand:

```

distribute command_tags execute
{[refunc] function | [replication_definition.]function |
sys_sp stored_procedure} [param_list]

```

- The `refunc` keyword (available only with LTL version 103 or later) indicates that the *function* name that follows is a user-defined function associated with a function replication definition. When you create a function replication definition for a replicated stored procedure, a user-defined function with the same name is created for you. In this case, the `execute` subcommand does not include the function replication definition name.

Replication Server distributes the `execute refunc` subcommand from a primary Replication Server to any replicate Replication Servers with subscriptions for the associated function replication definition. However, for request functions in Replication Server 15.0.1 and earlier, Replication Server distributes the `execute refunc` subcommand from a replicate Replication Server to the primary Replication Server.

- When the `refunc` keyword is omitted, the *function* name that follows is a user-defined function associated with a table replication definition, and *replication_definition* is the name of the replication definition. Refer to the *Replication Server Administration Guide Volume 2* for a detailed description of user-defined functions.

Without the `refunc` keyword, the `execute` subcommand is used only for request stored procedures associated with table replication definitions. (Applied stored procedures associated with table replication definitions use the `applied` subcommand.) Replication Server distributes the `execute` subcommand from a replicate Replication Server to the primary Replication Server for the table replication definition.

If the `execute` subcommand does not specify a replication definition, Replication Server searches its system tables for the function name and then finds the associated table replication definition. If the function name is not unique, and the replication definition is not specified, an error message reports that the function name is valid for more than one replication definition.

- *function* is the name of both the user-defined function and the replicated stored procedure. When Replication Server receives the `execute` command, it maps the *function* name to a user-defined function previously created by `create applied function replication definition`, `create request function replication definition`, `create function replication definition`, or `create function`.
- With LTL version 200 or later, RepAgent uses `sys_sp` to send system stored procedures to the standby database.

- *param_list* is a list of the data values supplied when the procedure was executed. You must enclose parameter values in parentheses.

See the *Replication Server Reference Manual* for more information about the create applied function replication definition, create request function replication definition, and create function commands. Also see the *Replication Server Administration Guide Volume 2* for more information about replicated functions and stored procedures.

Processing the rs_marker function

When `rs_marker` is executed, RepAgent processes it so that Replication Server synchronizes subscription materialization cycles and warm standby applications.

`rs_marker` is executed with one `varchar(255)` parameter named `@rs_api`. RepAgent passes the parameter to the Replication Server in a distribute command with the following syntax:

```
distribute command_tags param_string
```

For example, if a client executes `rs_marker` with the following command:

```
rs_marker @rs_api='0x1234567'
```

The Adaptive Server RepAgent submits the following command to the Replication Server:

```
distribute command_tags 0x1234567
```

Notice that the `@rs_api` parameter is not in quotes.

sqlddl append subcommand

The `sqlddl append` subcommand (LTL version 200 or later) is used to apply DDL commands such as `create table` to the warm standby application as original text strings. Because long DDL may span several commands, `sqlddl append` lets you indicate the first and last text strings for the DDL command you want to apply.

`sqlddl append` has the following syntax:

```
sqlddl append [ first | last ] ddl_string
```

- *first* indicates the first part of a DDL sequence.
- *last* indicates the last part of a DDL sequence.

purge subcommand

The purge subcommand instructs the Replication Server to purge all open transactions in the inbound queue for which the origin queue ID (*oqid*) of the begin record is less than that specified in the command.

distribute *command_tags* purge open_xact to *oqid*

- *oqid* is the origin queue ID number below which you want to purge all open transactions.

The purge subcommand requires LTL version 102 or higher.

Sample RepAgent session

This section contains a sample dialog between a RepAgent and a Replication Server. The example transfers two transactions to the Replication Server.

The transaction log contains two concurrent transactions, “T1” and “T2”, with log records:

```
T1: begin transaction
T2: begin transaction
T1: insert into authors ('karsen', '510 534-9219')
T2: update authors set phone = '510 986-7020'
     where name = 'green' and phone = '415 986-7020'
T2: commit transaction
T1: commit transaction
```

Note LTL commands are usually submitted to Replication Server by a RepAgent using Open Client Client-Library routines. However, you can submit LTL commands interactively using isql.

```
distribute
  @origin_time='Dec 10 1992  8:48:12:750AM',
  @origin_qid=0x00000000000000000000000000000001,
  @tran_id=0x000000111111
  begin transaction 'T1' for 'user'/'password'
distribute
  @origin_time='Dec 10 1992  8:48:12:750AM',
  @origin_qid=0x00000000000000000000000000000002,
  @tran_id=0x000000222222
  begin transaction 'T2' for 'user'/'password'
```

```
distribute
  @origin_time='Dec 10 1992  8:48:13:750AM',
  @origin_qid=0x00000000000000000000000000000003,
  @tran_id=0x0000001111
  applied 'authors'.rs_insert yielding
    after @name='karsen', @phone='510 534-9219'
distribute
  @origin_time='Dec 10 1992  8:48:13:750AM',
  @origin_qid=0x00000000000000000000000000000004,
  @tran_id=0x000000222222
  applied 'authors'.rs_update yielding
    before @name='green', @phone='415 986-7020'
    after @name='green', @phone='510 986-7020'
distribute
  @origin_time='Dec 10 1992  8:48:14:750AM',
  @origin_qid=0x00000000000000000000000000000005,
  @tran_id=0x000000222222
  commit transaction
distribute
  @origin_time='Dec 10 1992  8:48:14:750AM',
  @origin_qid=0x00000000000000000000000000000006,
  @tran_id=0x000000111111
  commit transaction
```

You can use the get truncation command to verify that the truncation point is set to the origin_qid from the last distribute command.

Index

Symbols

@ xv
– xv

A

activate subscription command 46, 47
active database 31
add partition command 93
admin_logical_status command 60, 61
advantages of replicating data 2–3
 greater data availability 3
 improved performance 2
after image 164
application models
 basic primary copy 40
 corporate rollup 51
 distributed primary fragments 47
 overview 39–40
 redistributed corporate rollup 55, 56
 variations and strategies 63–87
 warm standby 58
applied functions
 definition 5
 using 43
applied, distribute subcommand 163
articles, definition of 67
assign action command 108
 actions for data server errors 107
assign action command actions
 ignore 108
 log 108
 retry_log 108
 retry_stop 108
 stop_replication 108
 warn 108
asynchronous procedure call and local update
 applications 30

asynchronous procedure execution, concurrency 35

B

backup and recovery methods 89–95
 preventive measures 90–94
 protecting against data loss 89–90
 recovery measures 94–95
basic multilingual plane. *See* BMP
basic primary copy model 40
 applied functions 43
 example using table replication definitions 42
 table replication definitions 41
before image 164
begin transaction, distribute subcommand 161
begin/commit pairs, calculating message size for 133
binary(10) datatype 134
binary(36) datatype 109
BMP 114, 115
bulk materialization
 character sets and 117
 sort orders and 117
 subscription method 47

C

C/SI. *See* Client/Server Interfaces
case in RCL commands xv
centralized and distributed database system 1
certifications
 component xii
 product xii
change rate, calculating 134
change volume, calculating 134
char(10) datatype 134
char(30) datatype 157
character sets
 changing 121

- changing character width 124
- configuring 113, 116
- conversion 113
- guidelines for using 115
- requirements for Unicode 115
- supported 113
- Unicode 114
- UTF-16 115
- UTF-8 114
- client applications 12
- Client/Server Interfaces 12, 18, 24
- column overhead 133
- command tags
 - in distribute command 160
 - mode** 161
 - origin_qid** 160
 - origin_time** 160
 - standby_only** 161
 - tran_id** 160
- commands
 - activate subscription** 46, 47
 - add partition** 93
 - admin_logical_status** 60, 61
 - assign action** 107, 108
 - configure connection** 108, 114, 140
 - configure route** 140
 - connect source** 23, 154–156
 - create article** 67, 70
 - create connection** 105
 - create error class** 107, 108
 - create function** 165, 172, 173
 - create function string class** 106
 - create logical connection** 60
 - create partition** 92, 93
 - create publication** 67, 70
 - create replication definition** 42, 50, 54, 65, 69, 164, 166
 - create subscription** 43, 46, 51, 55, 66, 67, 71, 86
 - define subscription** 46, 47
 - distribute** 153, 159–175, 176
 - drop connection** 62
 - get maintenance user** 157
 - get truncation** 158, 176
 - grant** 21, 22
 - resume connection** 61
 - revoke** 21, 22
 - rs_subcmp** 90, 94, 118, 119, 121, 122
 - suspend connection** 108
 - switch active** 61
 - validate publication** 67, 71
 - validate subscription** 46, 47
- commit transaction**, distribute subcommand 162
- communication
 - JDBC protocol 102
 - Replication Agent protocols 102
- components of replication system
 - client applications 12
 - data servers 11
 - ID Server 10
 - overview 7
 - Replication Agent 11
 - replication environment 10
 - Replication Manager (RM) 11
 - Replication Monitoring Services (RMS) 11
 - Replication Server 8
 - replication system domain 8
- concurrency control 6, 7
 - optimistic 6
 - pessimistic distributed 6
- configure connection** command 108, 114, 140
- configure route** command 140
- conflicting updates
 - preventing 36
 - version control 36
- connect source** keywords
 - in recovery** 156
 - sendallxacts** 155
 - warmstdb** 156
- connect source** LTL command 23, 154–156
 - example 156
 - in RepAgent process 98
 - upgrade locator 156
- connect source** permission 23
- connecting replication system components 13
- connection profiles 18
- connection profiles, for non-ASE data servers 104
- connections
 - definition 14
- conventions
 - examples xiii
 - syntax statements xiv
- conversion of character sets 113

coordinated dump, restoring 95
 corporate rollup model 52, 55
 CPU requirements, planning 127
 CPU usage 151
create article command 67, 70
create connection command 105
create error class command 107, 108
create function command 165, 172, 173
create function string class command 106
create logical connection command 60
create object permission 23
create partition command 92, 93
create publication command 67, 70
create replication definition command 42, 50, 54,
 65, 69, 164, 166
create subscription command 43, 46, 51, 55, 66, 67,
 71, 86

D

data recovery
 automatic 94
 by re-creating subscriptions 94
 data servers
 described 11
 login names 22
 non-ASE 18
 processing errors 19
 data, primary. See primary data
 database volume, calculating 137
 datatype translations 18
 datatypes
 binary(10) 134
 binary(36) 109
 char(10) 134
 char(30) 157
 datetime 109, 159, 160
 image 106, 115, 150, 151, 163, 166
 in **rs_lastcommit** table 109
 int 109
 text 106, 150, 151, 163, 166
 unichar 114, 115
 unitext 114, 115
 univarchar 114, 115
 varbinary(36) 174

varchar(255) 173
 varchar(30) 162, 174
datetime datatype 109, 159, 160
 decision-support applications 26–28, 34, 40
define subscription command 46, 47
 definition
 identifiers xv
 deletes
 calculating message size for 132
 direct routes 15
 disk partitions 9
 disk space requirements 129, 148
 planning 127
distribute LTL command 153, 159–175, 176
distribute subcommands
 applied 163
 begin transaction 161
 commit transaction 162
 dump 174
 execute 171
 purge 175
 rollback 162
 rollback transaction 162
 sqlddl append 173
 transaction-control subcommands 161
 distributed OLTP applications 28–30
 distributed primary fragments model 47, 49, 51
drop connection command 62
dump, distribute subcommand 174
 example 174
 parameters 174
 syntax 174

E

ECDA 18
 embedded Replication Server System Database. *See*
 ERSSD
 error class 18, 19, 107
 ERSSD
 described 9
 examples
 style conventions xiii
execute, distribute subcommand 171

F

Failover 92
 failure duration 140
 fault tolerance 16
 for non-Sybase databases 99–??
for_recovery RepAgent option 156
 function replication definition
 creating subscriptions for 46
 described 5
 sample script 46, 69, 75, 85–86
 function strings 20
 function variable 20
 functions 18, 20
 calculating message size for 133
 function-string classes 20
 creating 106
 described 20
 for DB2 106
 for foreign data server 105
 inheriting 105

G

get maintenance user LTL command 157
get truncation LTL command 158, 176
grant command 21, 22

H

hastext
 value for text_status 167
 hierarchical configuration 16

I

icons
 Adaptive Server xvi
 client application xvi
 Replication Agent xvi
 Replication Manager xvi
 Replication Server xvi
 ID Server
 described 10

 in Replication system domain 8
 login name 10
 requirements 10
 identifiers
 definition of xv
 format xv
 function parameters xvi
 length xv
 types of xv
ignore, error action 108
image datatype 106, 115, 150, 151, 163, 166
in recovery, connect source keyword 156
 inbound database volume 138
 example calculations 144
 inbound message overhead 133
 inbound queue size
 calculating 138
 example calculations 145
 indirect routes 15
 fault tolerance 16
 reducing load with additional Replication Servers
 16
 reducing volume on WAN 15
 inserts
 calculating message size for 132
int datatype 109
 interfaces file 13
 and warm standby applications 62
 international environments
 support for 111, 126
 internationalization
 Replication Server 111
 isql 9

J

Java (programming language) 102
 Java Runtime Environment (JRE) 102
 JDBC driver 102

L

lag time. *See* latency
 languages

- configuring 112
 - large object datatypes
 - partial update 169
 - latency
 - described 32
 - limiting transaction risk 33
 - measuring 32
 - measuring replication performance 33
 - LDAP 13
 - local pending table 30, 72
 - local-area network 1
 - localization of messages 111
 - locator
 - upgrade 156
 - Log Reader. *See* Replication Agent components
 - Log Transfer Interface (LTI). *See* Replication Agent components
 - Log Transfer Language. *See* LTL
 - Log Transfer Manager. *See* Replication Agent components
 - log**, error action 108
 - log, transaction. *See* transaction log
 - logical connections
 - definition 14
 - login names 21
 - data server 22
 - ID Server 10
 - maintenance user 22
 - Replication Server 21
 - loose consistency 31, 32
 - LTI. *See* Replication Agent components
 - LTL
 - overview of commands 153
 - versions of 154
 - LTL commands
 - connect source** 23, 154–156
 - distribute** 153, 159–175, 176
 - get maintenance user** 157
 - get truncation** 158, 176
 - LTL compatibility table 154
- M**
- maintenance user 157
 - permissions for 22
 - master database
 - replication. *See* master database replication
 - supported DDL and system procedures 18
 - master database replication 17
 - MSA, with 18
 - warm standby, with 18
 - master/detail implementation
 - strategy for 76
 - memory requirements 148
 - planning 127
 - RepAgent 149
 - Replication Server 149
 - message languages
 - configuring 112
 - message overhead
 - inbound 133
 - outbound 133
 - message sizes
 - calculating 131, 134
 - example calculations 143
 - minimal columns
 - calculating message size for 132
 - mode command tag 161
 - mode**, distribute command tag 161
 - multiple primaries
 - designing around update conflicts 35
 - managing update conflicts 35
 - multiple replication definitions 63, 66
 - MySybase xii
- N**
- network resources, planning 127
 - network-based security
 - credential 23
 - no-materialization subscription method 46
 - non-ASE data servers
 - connection profiles 104
 - support for 18, 103
 - non-binary
 - sort orders 116
 - notrep
 - value for text_status 167
 - number of sites 134

O

OLTP applications 26, 34, 40, 92
 distributed 28
 local update 30
 using request functions 30
 optimistic concurrency 6
 origin queue ID 98, 158
origin_qid, distribute command tag 160
origin_time, distribute command tag 160
 outbound message overhead 133
 outbound queue size
 calculating 140
 example calculation 146
 outbound queue volume
 calculating 139, 140
 example calculation 145
 outbound transaction rate 137

P

parameter width 134
 partial update 169
 partitions 9
 pending table
 with request functions 72
 pending updates table 30
 permissions 22
 connect source 23
 create object 23
 primary object 23
 sa 22
 personalized views
 creating xii
 pessimistic concurrency control 6
 primary data
 centralized 34
 client updates 12, 22
 maintaining 33
 and RepAgents 19
 updating from remote sites 33
 primary database
 mirroring 92
 primary fragment 29
primary object permission 23
 products for non-Sybase databases ??–102

publication 23
 publication subscriptions
 definition 67
 publications 66, 71
 definition of 67
 described 4
 procedure for creating 67
 publish-and-subscribe model
 described 4
purge, distribute subcommand 175

Q

queue disk usage
 calculating 141

R

recovering primary databases
 from dumps 95
 recovery mode 95
 re-creating subscriptions 94
 redistributed corporate rollup model 55, 56, 57
 example 57
 remote OLTP using request functions 30
 remote procedure call 20
 REP_SSL feature 24
 RepAgent
 described 11, 98
 role in replication system 19
 sample session 175
 See also Replication Agent
 RepAgent options
 for_recovery 156
 send_maint_xacts_to_replicate 56, 57, 98, 155
 send_warm_standby_xacts 60, 98, 156, 157
 repfunc keyword 171
 replicated functions
 described 4
 introduction to 4
 used for 5
 replicated stored procedures 171
 replicated table, modifying 22
 replicating data

- advantages 2
- replicating master database. *See* master database
 - replication
- replication
 - basic concepts 99
- Replication Agent
 - communication 102
 - described 11
 - for non-Sybase databases 98
 - overview 97
 - role in replication system 19
 - tasks 98
 - transaction log 101
- Replication Agent components
 - Log Reader
 - Log Transfer Interface (LTI)
 - Log Transfer Manager
- Replication Command Language. *See* RCL 9
- replication definitions
 - described 4
- replication management solutions
 - three-tier 12
 - two-tier 12
- Replication Server
 - application types 25–31
 - backup and recovery 89–95
 - described 8
 - fault tolerance 16
 - login names 21
 - LTL compatibility 154
 - non-ASE data servers, and 103
 - reducing load 16
- Replication Server application types 25–31
 - decision-support applications 26–28
 - distributed OLTP applications 28–30
 - remote OLTP using request functions 30
 - warm standby applications 31
- Replication Server System Database. *See* RSSD
- replication system 24
 - components 7
 - diagram 8
- replication, master database. *See* master database
 - replication
- replication_role permission 60
- request functions 71, 76
 - definition 5
 - with pending table 72
- restoring
 - coordinated dump 95
 - dumps 94
- resume connection** command 61
- retry_log**, error action 108
- retry_stop**, error action 108
- revoke** command 21, 22
- RM
 - described 11
- RMS
 - described 11
 - three-tier management solution 11, 13
- rollback transaction**, distribute subcommand 162
- rollback**, distribute subcommand 162
- routes
 - definition 14
 - hierarchical configuration 16
 - star configuration 16
- Routes and connections
 - diagram 15
- routes and connections 14
- row width changed
 - in calculating message size 134
- rs_datarow_for_writetext** function 106, 166
- rs_db2_function_string_class** function-string class 106
- rs_default_function_string_class** function-string class 106
- rs_delete** function 106, 164
- rs_get_lastcommit** function 110
- rs_get_textptr** function 106
- rs_init** configuration utility
 - creating connections 14
 - recording ID Server login name 10
 - recording Replication Server login name 21
- rs_insert** function 106, 164
- rs_lastcommit** table 108
- rs_marker function**, RepAgent processing of 173
- rs_select** function 107
- rs_select_with_lock** function 107
- rs_subcmp** command 90, 94, 118, 119, 121, 122
 - character sets and 119
 - sort orders and 119
- rs_textptr_init** function 106
- rs_truncate** function 171

- rs_update** function 106, 164
 - rs_update_lastcommit** stored procedure 109
 - rs_updatetext** 169
 - example 170
 - limitations 171
 - parameters 169
 - syntax 169
 - rs_writetext** function 106, 166, 167
 - RSSD
 - described 9
 - disk requirements 128
 - Replication Agent accessing 101
- S**
- sa** permission 22
 - Sample RepAgent session 175
 - save interval 93, 140
 - secure socket layers 24
 - security
 - network-based 23
 - Replication Server 21
 - send_maint_xacts_to_replicate** RepAgent option 56, 57, 98, 155
 - send_warm_standby_xacts** RepAgent option 60, 98, 156, 157
 - sendallxacts**, connect source keyword 155
 - sort orders
 - changing 121
 - configuring 116
 - Unicode 120, 121
 - sp_config_rep_agent** stored procedure 60
 - sp_reptostandby** stored procedure 59
 - sp_setreproc** stored procedure 44, 78
 - sp_setreptable** stored procedure 41, 48, 53
 - sqlddl append**, distribute subcommand 173
 - stable queues 9
 - mirroring 92
 - standby
 - applications 30
 - database 31
 - standby_only**, distribute command tag 161
 - star configuration 16
 - stop_replication**, error action 108
 - stored procedures
 - example for publications 68
 - example used with pending table 74
 - example, creating at primary and replicate sites 45
 - message location 124
 - rs_update_lastcommit** 109
 - sp_config_rep_agent** 60
 - sp_reptostandby** 59
 - sp_setreproc** 44, 78
 - sp_setreptable** 41, 48, 53
 - upper-level 78
 - with delete clauses 80
 - with insert clauses 78
 - with update clauses 81
 - subscription method
 - bulk materialization 47
 - no-materialization 46
 - subscription migration 77
 - subscriptions
 - character sets and 116, 120
 - creating for a function replication definition 46
 - described 4
 - primary fragments 48
 - sort orders and 116, 120
 - suspend connection** command 108
 - switch active** command 61
 - switching active and standby databases 61
 - Sybase Enterprise Connect Data Access 18
 - symmetric multiprocessor 151
 - enabling 152
 - syntax conventions
 - identifiers xv
 - syntax statements, conventions xiv
- T**
- table replication definitions 41
 - table volume
 - calculating 135
 - example calculations 144
 - table.function_name 165
 - text** datatype 106, 150, 151, 163, 166
 - text or image data
 - modifications to 166, 168
 - three-tier management solution 13
 - RMS 11, 13

- token header
 - for text or image data 168
- total disk space
 - example calculations 147
- tpinit
 - value for text_status 167
- tpnull
 - value for text_status 167
- tran_id**, distribute command tag 160
- transaction
 - calculating volume 136
 - duration 138
- transaction log 97, 101
 - mirroring 92
- transaction-control, distribute subcommands 161
- transactions
 - failed 6
 - for multiple databases 7
 - high value 32
 - management 5
- truncate table or partition 171
- two-tier management solution 12

U

- unichar** datatype 114, 115
- Unicode character sets
 - supported 114
- Unicode sort order 120
- unitext** datatype 114, 115
- univarchar** datatype 114, 115
- updates
 - calculating message size for 132
- upgrade locator 156
- upper-level stored procedures 78
- user defined datatypes (UDD) 18
- user documentation, for Replication Server x
- user-defined function
 - mapping to a replication definition 172
- UTF-16 character set 115
- UTF-8 character set 114

V

- validate publication** command 67, 71
- validate subscription** command 46, 47
- varbinary(36)** datatype 174
- varchar(255)** datatype 173
- varchar(30)** datatype 162, 174
- version-controlled updates 36

W

- WAN
 - described 1
 - reducing volume with routes 15
 - using for primary data maintenance 35
- warm standby applications 58, 63
 - comparison with data mirroring 90
 - example 59
 - overview 31
 - procedure for setting up 59
- warmstdb**, connect source keyword 156
- warn**, error action 108
- wide-area network. See WAN

Y

- yielding clause 165

Z

- zerolen
 - value for text_status 167

