



SQL Remote™

Published: October 2006

Copyright and trademarks

Copyright © 2006 iAnywhere Solutions, Inc. Portions copyright © 2006 Sybase, Inc. All rights reserved.

iAnywhere Solutions, Inc. is a subsidiary of Sybase, Inc.

iAnywhere grants you permission to use this document for your own informational, educational, and other non-commercial purposes; provided that (1) you include this and all other copyright and proprietary notices in the document in all copies; (2) you do not attempt to "pass-off" the document as your own; and (3) you do not modify the document. You may not publish or distribute the document or any portion thereof without the express prior written consent of iAnywhere.

This document is not a commitment on the part of iAnywhere to do or refrain from any activity, and iAnywhere may change the content of this document at its sole discretion without notice. Except as otherwise provided in a written agreement between you and iAnywhere, this document is provided "as is", and iAnywhere assumes no liability for its use or any inaccuracies it may contain.

iAnywhere®, Sybase®, and the marks listed at <http://www.iAnywhere.com/trademarks> are trademarks of Sybase, Inc. or its subsidiaries. ® 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.

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

Contents

About This Manual	vii
SQL Anywhere documentation	viii
Documentation conventions	xi
Finding out more and providing feedback	xv
I. Introduction to SQL Remote	1
Welcome to SQL Remote	3
About SQL Remote	4
About this manual	5
SQL Remote Concepts	7
SQL Remote components	8
Publications and subscriptions	10
SQL Remote features	12
Typical SQL Remote setups	13
II. Replication Design for SQL Remote	15
Principles of SQL Remote Design	17
Design overview	18
How statements are replicated	19
How data types are replicated	23
Who gets what?	25
Replication errors and conflicts	26
SQL Remote Design	29
Design overview	30
Publishing data	31
Publication design	39
Partitioning tables that do not contain the subscription expression	42
Sharing rows among several subscriptions	48
Managing conflicts	55
Ensuring unique primary keys	63
Creating subscriptions	71

III. SQL Remote Administration	73
Deploying and Synchronizing Databases	75
Deployment overview	76
Test before deployment	77
Synchronizing databases	79
Using the extraction utility	81
Synchronizing data over a message system	86
SQL Remote Administration	87
Management overview	88
Managing SQL Remote permissions	89
Using message types	96
Running the Message Agent	108
Tuning Message Agent performance	112
Encoding and compressing messages	118
The message tracking system	120
Administering SQL Remote	123
Running the Message Agent	124
Error reporting and handling	126
Transaction log and backup management	130
Using passthrough mode	141
IV. Reference	145
Utilities and Options Reference	147
Message Agent	148
Database Extraction utility	156
SQL Remote options	163
SQL Remote event-hook procedures	167
System Objects for SQL Remote	171
SQL Remote system tables	172
SQL Remote SQL Statements	173
SQL Remote statements	174
V. Appendices	175

Supported Platforms and Message Links	177
Supported message systems	178
Supported operating systems	179
Index	181

About This Manual

Subject

This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.

Audience

This book is for users of SQL Anywhere who want to add SQL Remote replication to their information systems.

Before you begin

☞ For a comparison of SQL Remote with other SQL Anywhere replication technologies, see “[Overview of Data Exchange Technologies](#)” [*SQL Anywhere 10 - Introduction*].

SQL Anywhere documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- ◆ **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- ◆ **SQL Anywhere 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software.
- ◆ **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, security, backup procedures, security, and replication with Replication Server, as well as administration utilities and options.
- ◆ **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- ◆ **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- ◆ **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.
- ◆ **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- ◆ **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- ◆ **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink applications.
- ◆ **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- ◆ **MobiLink - Server-Initiated Synchronization** This manual describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.

- ◆ **QAnywhere** This manual describes QAnywhere, which defines a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.
- ◆ **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- ◆ **SQL Anywhere 10 - Context-Sensitive Help** This manual provides context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- ◆ **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- ◆ **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- ◆ **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- ◆ **UltraLite - M-Business Anywhere Programming** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.
- ◆ **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- ◆ **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

To access the online documentation on Windows operating systems, choose Start ► Programs ► SQL Anywhere 10 ► Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

To access the online documentation on Unix operating systems, see the HTML documentation under your SQL Anywhere installation or on your installation CD.

- ◆ **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

On Windows, the PDF books are accessible from the online books via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are accessible on your installation CD.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- ◆ **Keywords** All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

```
ALTER TABLE [ owner.]table-name
```

- ◆ **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

```
ALTER TABLE [ owner.]table-name
```

- ◆ **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

```
ADD column-definition [ column-constraint, ... ]
```

One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- ◆ **Optional portions** Optional portions of a statement are enclosed by square brackets.

```
RELEASE SAVEPOINT [ savepoint-name ]
```

These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- ◆ **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

```
[ ASC | DESC ]
```

For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- ◆ **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

```
[ QUOTES { ON | OFF } ]
```

If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

File name conventions

The documentation generally adopts Windows conventions when describing operating-system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- ◆ **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

```
MobiLink\redirector
```

On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

```
MobiLink/redirector
```

- ◆ **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

- ◆ **install-dir** The installation process allows you to choose where to install SQL Anywhere, and the documentation refers to this location using the convention *install-dir*.

After installation is complete, the environment variable `SQLANY10` specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). `SQLANYSH10` specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

For more information on the default location of *install-dir*, by operating system, see “[File Locations and Installation Settings](#)” [*SQL Anywhere Server - Database Administration*].

- ◆ **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

After installation is complete, the environment variable `SQLANYXSAMP10` specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ► SQL Anywhere 10 ► Sample Applications and Projects opens a Windows Explorer window in this directory.

For more information on the default location of *samples-dir*, by operating system, see “[The samples directory](#)” [*SQL Anywhere Server - Database Administration*].

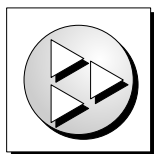
- ◆ **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *\$envvar* or *\${envvar}*.

Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

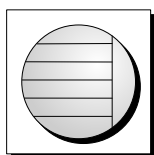
Graphic icons

The following icons are used in this documentation.

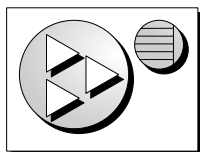
- ◆ A client application.



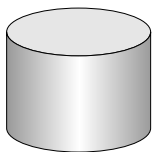
- ◆ A database server, such as SQL Anywhere.



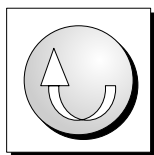
- ◆ An UltraLite application.



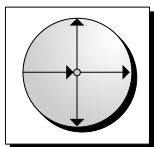
- ◆ A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- ◆ Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.



- ◆ A Sybase Replication Server



- ◆ A programming interface.



Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at <http://www.ianywhere.com/developer/>.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by entering **dbeng10 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- ◆ [sybase.public.sqlanywhere.general](#)
- ◆ [sybase.public.sqlanywhere.linux](#)
- ◆ [sybase.public.sqlanywhere.mobilink](#)
- ◆ [sybase.public.sqlanywhere.product_futures_discussion](#)
- ◆ [sybase.public.sqlanywhere.replication](#)
- ◆ [sybase.public.sqlanywhere.ultralite](#)
- ◆ [ianywhere.public.sqlanywhere.qanywhere](#)

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

Part I. Introduction to SQL Remote

This part describes the concepts, architecture, and features of SQL Remote.

CHAPTER 1

Welcome to SQL Remote

Contents

About SQL Remote 4
About this manual 5

About this chapter

This chapter introduces SQL Remote and the documentation.

About SQL Remote

SQL Remote is a data-replication technology designed for two-way replication between a consolidated data server and large numbers of remote databases, typically including many mobile databases.

SQL Remote replication is message based, and requires no direct server-to-server connection. An occasional dialup or email link is sufficient.

Administration and resource requirements at the remote sites are minimal. The time lag between the consolidated and remote databases is configurable, and can range from minutes to hours or days.

In a SQL Remote installation, you must have properly licensed SQL Remote software at each participating database.

☞ For a detailed introduction to SQL Remote concepts and features, see [“SQL Remote Concepts” on page 7](#).

☞ For a list of supported operating systems and message links, see [Appendix “Supported Platforms and Message Links” on page 177](#).

About this manual

This manual describes how to design, build, and maintain SQL Remote installations.

The manual includes the following parts.

- ◆ **Introduction to SQL Remote** Replication concepts and features of SQL Remote.
- ◆ **Replication Design for SQL Remote** Designing SQL Remote installations.
- ◆ **SQL Remote Administration** Deploying SQL Remote databases and administering a running SQL Remote setup.
- ◆ **Reference** SQL Remote commands, system tables, and other reference material.

CHAPTER 2

SQL Remote Concepts

Contents

SQL Remote components	8
Publications and subscriptions	10
SQL Remote features	12
Typical SQL Remote setups	13

About this chapter

This chapter introduces the concepts, design goals, and features of SQL Remote.

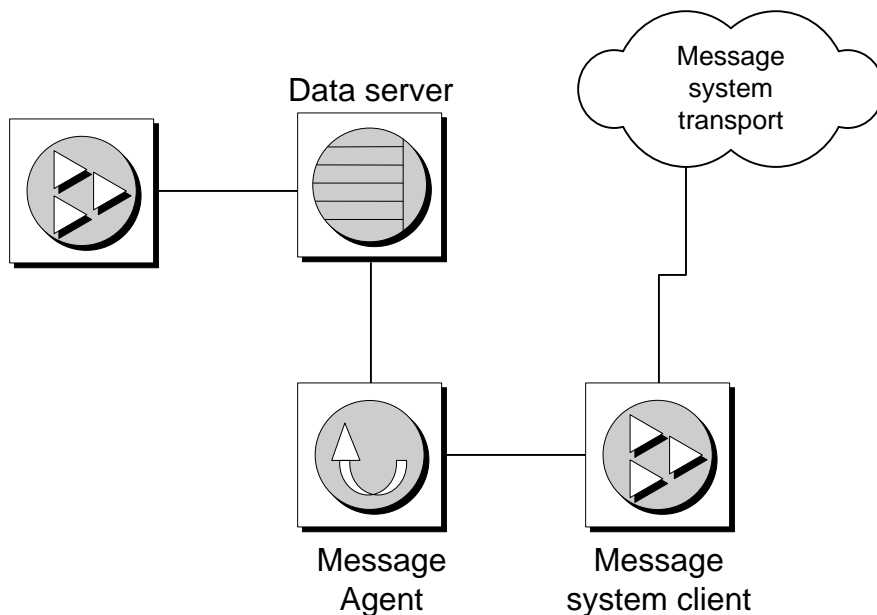
SQL Remote components

The following components are required for SQL Remote:

- ◆ **Data server** A SQL Anywhere database is required at each remote site and the consolidated database to maintain the data.
- ◆ **Message Agent** A SQL Remote Message Agent is required at the consolidated site and at each remote site to send and receive SQL Remote messages.

The Message Agent connects to the data server by a client/server connection. It may run on the same machine as the data server or on a different machine.

- ◆ **Database extraction utility** The extraction utility can be used to prepare remote databases from a consolidated database, during development and testing, and also at deployment time.
- ◆ **Message system client software** SQL Remote uses existing message systems to transport replication messages. A file-sharing "message system" is provided, which does not require client software. Each computer involved in SQL Remote replication using a message system other than file sharing must have that message system installed.
- ◆ **Client applications** The applications that work with SQL Remote databases are standard client/server database applications.



The data server

The data server must be a SQL Anywhere server.

Client applications

The client application can use ODBC, embedded SQL, or a variety of other programming interfaces.

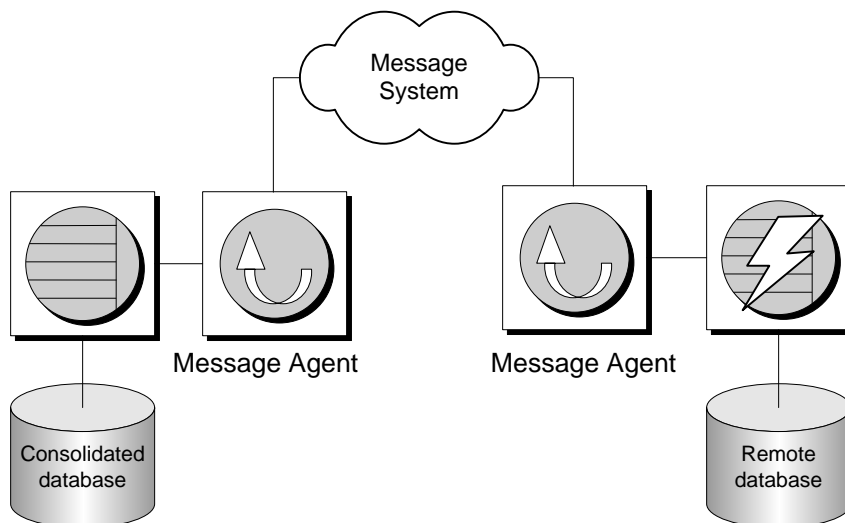
☞ See “[SQL Anywhere Data Access Programming Interfaces](#)” [[SQL Anywhere Server - Programming](#)].

Client applications do not have to know if they are using a consolidated or remote database. From the client application perspective, there is no difference.

Message Agent

The SQL Remote **Message Agent** sends and receives replication messages. It is a client application that sends and receives messages from database to database. The Message Agent must be installed at both the consolidated and at the remote sites.

The Message Agent is a program called *dbremote.exe* on PC operating systems, and *dbremote* on Unix.



Message system client

If you are using a shared file message system, no message system client is needed.

If you are using an email or other message system, you must have a message system for that client in order to send and receive messages.

Publications and subscriptions

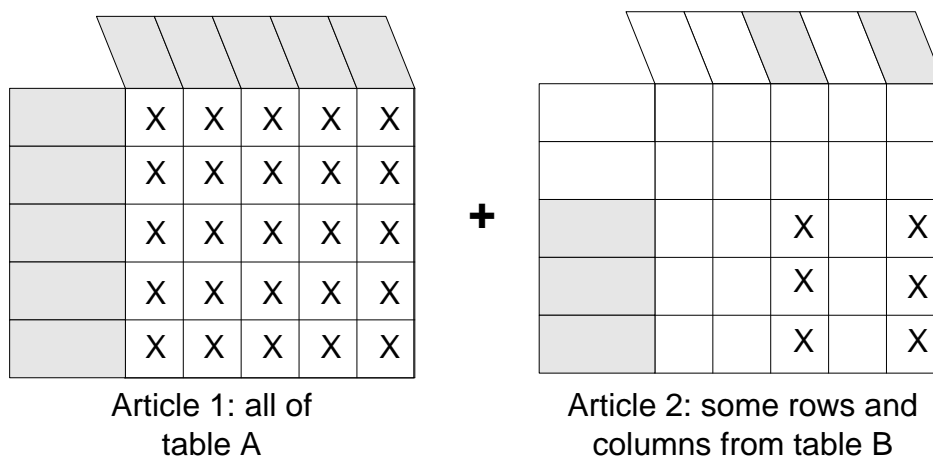
The data that is replicated by SQL Remote is arranged in **publications**. Each database that shares information in a publication must have a **subscription** to the publication.

Data is organized into publications

The publication is a database object describing data to be replicated. Remote users of the database who want to receive a publication do so by subscribing to a publication.

A publication may include data from several database tables. Each table's contribution to a publication is called an article. Each article may consist of a whole table, or a subset of the rows and columns in a table.

A two-table synchronization definition



Periodically, the changes made to each publication in a database are replicated to all subscribers to that publication. These replications are called publication updates.

Messages are always sent both ways

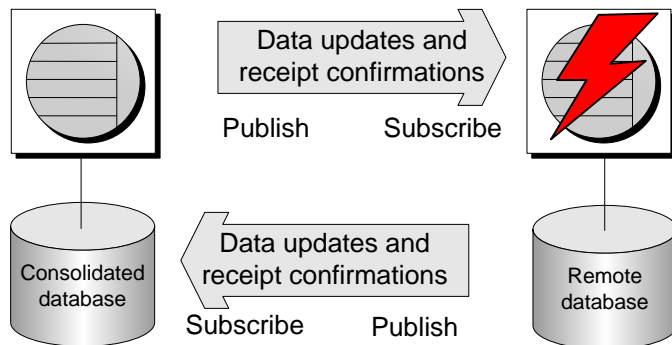
Remote databases subscribe to publications on the consolidated database so that they can receive data from the consolidated database. To do this, a subscription is created at the consolidated database, identifying the subscriber by name and by the publication they are to receive.

SQL Remote always involves messages being sent two ways. The consolidated database sends messages containing publication updates to remote databases, and remote databases also send messages to the consolidated database.

For example, if data in a publication at a consolidated database is updated, those updates are sent to the remote databases. And even if the data is never updated at the remote database, confirmation messages must still be sent back to the consolidated database to keep track of the status of the replication.

Both databases subscribe

Messages must be sent both ways, so not only does a remote database subscribe to a publication created at the consolidated database, but the consolidated database must subscribe to a corresponding publication created at the remote database.



When remote database users modify their own copies of the data, their changes are replicated to the consolidated database. When the messages containing the changes are applied at the consolidated database the changes become part of the consolidated database's publication, and are included in the next round of updates to all remote sites (except the one it came from). In this way, replication from remote site to remote site takes place via the consolidated database.

Setting up the remote database

When a subscription is initially set up, the two databases must be brought to a state where they both have the same set of information, ready to start replication. This setup can be done manually, but the database extraction utility automates the process. You can run the Extraction utility as a command line utility or from Sybase Central.

The appropriate publication and subscription are created automatically at remote databases when you use the SQL Remote database extraction utility to create a remote database.

SQL Remote features

The following features are key to SQL Remote's design.

Support for many subscribers SQL Remote is designed to support replication with many subscribers to a publication.

This feature is of particular importance for mobile workforce applications, which may require replication to the laptop computers of hundreds or thousands of sales representatives from a single office database.

Transaction log-based replication SQL Remote replication is based on the transaction log. This enables it to replicate only changes to data, rather than all data, in each update. Also, log-based replication has performance advantages over other replication systems.

The transaction log is the repository of all changes made to a database. SQL Remote replicates changes made to databases as recorded in the transaction log. Periodically, all committed transactions in the consolidated database transaction log belonging to any publication are sent to remote databases. At remote sites, all committed transactions in the transaction log are periodically submitted to the consolidated database.

By replicating only committed transactions, SQL Remote ensures proper transaction atomicity throughout the replication setup and maintains a consistency among the databases involved in the replication, albeit with some time lag while the data is replicated.

Central administration SQL Remote is designed to be centrally administered, at the consolidated database. This is particularly important for mobile workforce applications, where laptop users should not have to carry out database administration tasks. It is also important in replication involving small offices that have servers but little in the way of administration resources.

Administration tasks include setting up and maintaining publications, remote users, and subscriptions, as well as correcting errors and conflicts if they occur.

Economical resource requirements The only software required to run SQL Remote in addition to your SQL Anywhere DBMS is the Message Agent and a message system. If you use the shared file link, no message system software is required as long as each remote user ID has access to the directory where the message files are stored.

Memory and disk space requirements have been kept moderate for all components of the replication system, so that you do not have to invest in extra hardware to run SQL Remote.

Multi-platform support SQL Remote is provided on a number of operating systems and message links.

 For a list of supported environments, see [Appendix “Supported Platforms and Message Links” on page 177](#).

Typical SQL Remote setups

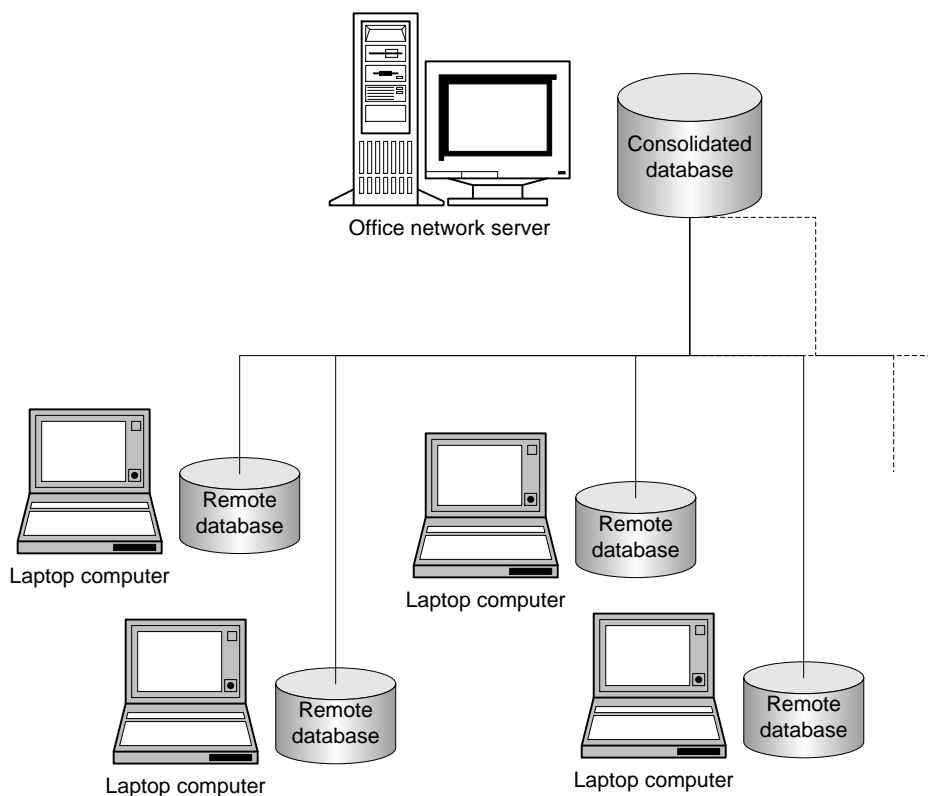
While SQL Remote can provide replication services in many different environments, its features are designed with the following characteristics in mind:

- ◆ SQL Remote should be a solution even when no administration load can be assigned to the remote databases, as in mobile workforce applications.
- ◆ Data communication among the sites may be occasional and indirect: it need not be permanent and direct.
- ◆ Memory and resource requirements at remote sites are assumed to be at a premium.

The following examples show some typical SQL Remote setups.

Server-to-laptop replication for mobile workforces

SQL Remote provides two-way replication between a database on an office network and personal databases on the laptop computers of sales representatives. Such a setup may use an email system as a message transport.



The office server may be running a server to manage the company database. The Message Agent at the company database runs as a client application for that server.

At the laptop computers each sales representative has a SQL Anywhere personal server to manage their own data.

While away from the office, a sales representative can make a single phone call from their laptop to carry out the following functions:

- ◆ Collect new email.
- ◆ Send any email messages they have written.
- ◆ Collect publication updates from the office server.
- ◆ Submit any local updates, such as new orders, to the office server.

The updates may include, for example, new specials on the products the sales representative handles, or new pricing and inventory information. These are read by the Message Agent on the laptop and applied to the sales rep's database automatically, without requiring any additional action on the sales representative's part.

The new orders recorded by the sales representative are also automatically submitted to the office without any extra action on the part of the sales representative.

Server-to-server replication among offices

SQL Remote provides two-way replication between database servers at sales offices or outlets and a central company office, without requiring database administration experience at each sales office beyond the initial setup and that required to maintain the server.

SQL Remote is not designed for instantaneous data availability at each site. For example, it may use an email system to carry the replication. Alternatively, an occasional dial-up system and file transfer software can be used to implement a FILE message system.

SQL Remote is easy to configure to allow each office to receive their own set of data. Tables that are of office interest only (staff records, perhaps, if the office is a franchise) may be kept private in the same database as the replicated data.

Layers can be added to SQL Remote hierarchies: for example, each sales office server could act as a consolidated database, supporting remote subscribers who work from that office.

Part II. Replication Design for SQL Remote

This part describes replication design issues for SQL Remote.

CHAPTER 3

Principles of SQL Remote Design

Contents

Design overview 18

How statements are replicated 19

How data types are replicated 23

Who gets what? 25

Replication errors and conflicts 26

About this chapter

This chapter describes general issues and principles for designing a SQL Remote application.

☞ For system-specific details, see [“SQL Remote Design” on page 29](#).

Design overview

This chapter describes general publication design issues that you must address when designing a SQL Remote application. It also describes how SQL Remote replicates data.

Design at the consolidated database

Like all SQL Remote administrative tasks, design is carried out by a database administrator or system administrator at the consolidated database.

The database administrator should perform all SQL Remote configuration tasks.

Using compatible sort orders and character sets

The SQL Remote Message Agent does not perform any character set conversions.

The character set and collation used by the SQL Anywhere consolidated database must be the same as the remote databases. For information about supported character sets, see [“International Languages and Character Sets” \[SQL Anywhere Server - Database Administration\]](#).

How statements are replicated

SQL Remote replication is based on the transaction log, enabling it to replicate only changes to data, rather than all data, in each update. When SQL Remote replicates data, it replicates SQL statements that modify data.

Only committed transactions are replicated

SQL Remote only replicates statements in committed transactions. This is done to ensure proper transaction atomicity throughout the replication setup and to maintain consistency among the databases involved in the replication, albeit with some time lag while the data is replicated.

Primary keys

When an UPDATE or a DELETE is replicated, SQL Remote uses the primary key columns to uniquely identify the row being updated or deleted. All tables being replicated must have a declared primary key or uniqueness constraint. A unique index is not sufficient. The columns of the primary key are used in the WHERE clause of replicated updates and deletes. If a table has no primary key, the WHERE clause refers to all columns in the table.

An UPDATE is not always an UPDATE

When a simple INSERT statement is entered at one database, it is sent to other databases in the SQL Remote setup as an INSERT statement. However, not all statements are replicated exactly as they are entered by the client application. This section describes how SQL Remote replicates SQL statements. It is important to understand this material if you are to design a robust SQL Remote installation.

The Message Agent is the component that carries out the replication of statements.

Replication of inserts and deletes

INSERT and DELETE statements are the simplest replication case. SQL Remote takes each INSERT or DELETE operation from the transaction log, and sends it to all sites that subscribe to the row being inserted or deleted.

If only a subset of the columns in the table is subscribed to, the INSERT statements sent to subscribers contains only those columns.

The Message Agent ensures that statements are not replicated to the user that initially entered them.

Replication of updates

UPDATE statements are not replicated exactly as the client application enters them. This section describes two ways in which the replicated UPDATE statement may differ from the entered UPDATE statement.

UPDATE statements replicated as INSERTS or DELETES

If an UPDATE statement has the effect of removing a row from a given remote user's subscription, it is sent to that user as a DELETE statement. If an UPDATE statement has the effect of adding a row to a given remote user's subscription, it is sent to that user as an INSERT statement.

The figure illustrates a publication, where each subscriber subscribes by their name:

Consolidated			Ann		Marc	
ID	Rep	Dept	ID	Rep	ID	Rep
1	Ann	101	1	Ann	2	Marc
2	Marc	101			3	Marc
3	Marc	101				

Consolidated			Ann		Marc	
ID	Rep	Dept	ID	Rep	ID	Rep
1	Ann	101	1	Ann	2	Marc
2	Marc	101	3	Ann	3	Marc
3	Ann	101				

An UPDATE that changes the **Rep** value of a row from Marc to Ann is replicated to Marc as a DELETE statement, and to Ann as an INSERT statement.

This reassignment of rows among subscribers is sometimes called **territory realignment**, because it is a common feature of sales force automation applications, where customers are periodically reassigned among representatives.

UPDATE conflict detection

An UPDATE statement changes the value of one or more rows from some existing value to a new value. The rows altered depend on the WHERE clause of the UPDATE statement.

When SQL Remote replicates an UPDATE statement, it does so as a set of single-row updates. These single-row statements can fail for one of the following reasons:

- ◆ **The row to be updated does not exist** Each row is identified by its primary key values, and if a primary key has been altered by some other user, the row to be updated is not found.

In this case, the UPDATE does not update anything.

- ◆ **The row to be updated differs in one or more of its columns** If one of the values expected to be present has been changed by some other user, an **update conflict** occurs.

At remote databases, the update takes place regardless of the values in the row.

At the consolidated database, SQL Remote allows **conflict resolution** operations to take place. Conflict resolution operations are held in a trigger or stored procedure, and run automatically when a conflict is detected. The conflict resolution trigger runs before the update, and the update proceeds when the trigger is finished.

- ◆ **A table without a primary key or uniqueness constraint refers to all columns in the WHERE clause of replicated updates** When two users update the same row, replicated updates will not

update anything and databases will become inconsistent. All replicated tables should have a primary key or uniqueness constraint and the columns in the constraint should never be updated.

Replication of procedures

Any replication system is faced with a choice between two options when replicating a stored procedure call:

- ◆ **Replicate the procedure call** A corresponding procedure is executed at the replicate site, or
- ◆ **Replicate the procedure actions** The individual actions (INSERTs, UPDATEs, DELETEs and so on) of the procedure are replicated.

SQL Remote replicates procedures by replicating the actions of a procedure. The procedure call is not replicated.

Replication of triggers

By default, the Message Agent does not replicate actions performed by triggers; it is assumed that the trigger is defined remotely. This avoids permissions issues and the possibility of each action occurring twice. There are some exceptions to this rule:

- ◆ **Conflict resolution trigger actions** The actions carried out by conflict resolution, or RESOLVE UPDATE, triggers *are* replicated from a consolidated database to all remote databases, including the one that sent the message causing the conflict.
- ◆ **Replication of BEFORE triggers** Some BEFORE triggers can produce undesirable results when using SQL Remote, and so BEFORE trigger actions that modify the row being updated *are* replicated, before UPDATE actions.

You must be aware of this behavior when designing your installation. For example, a BEFORE UPDATE that bumps a counter column in the row to keep track of the number of times a row is updated would double count if replicated, as the BEFORE UPDATE trigger will fire when the UPDATE is replicated. To prevent this problem, you must ensure that, at the subscriber database, the trigger is not present or does not carry out the replicated action. Also, a BEFORE UPDATE that sets a column to the time of the last update will get the time the UPDATE is replicated as well.

An option to replicate trigger actions

The Message Agent has an option that causes it to replicate all trigger actions when sending messages. This is the dbremote -t option.

If you use this option, you must ensure that the trigger actions are not carried out twice at remote databases, once by the trigger being fired at the remote site, and once by the explicit application of the replicated actions from the consolidated database.

To ensure that trigger actions are not carried out twice, you can wrap an `IF CURRENT REMOTE USER IS NULL ... END IF` statement around the body of the triggers or you can set the SQL Anywhere `fire_triggers` option to Off for the Message Agent user ID.

Replication of data definition statements

Data definition statements (`CREATE`, `ALTER`, `DROP`, and others that modify database objects) are not replicated by SQL Remote unless they are entered while in passthrough mode.

☞ For information about passthrough mode, see [“Using passthrough mode” on page 141](#).

How data types are replicated

Long binary or character data, and datetime data, need special consideration.

Replication of BLOBs

BLOBs are LONG VARCHAR, LONG BINARY, TEXT, and IMAGE data types: values that are longer than 256 characters.

SQL Remote includes a special method for replicating BLOBs between databases.

The Message Agent uses a variable in place of the value in the INSERT or UPDATE statement that is being replicated. The value of the variable is built up by a sequence of statements of the form

```
SET vble = vble || 'more_stuff'
```

This makes the size of the SQL statements involving long values smaller, so that they fit within a single message. The SET statements are separate SQL statements, so that the BLOB is effectively split over several SQL Remote messages.

Using the verify_threshold option to minimize message size

The verify_threshold database option can prevent long values from being verified (in the VERIFY clause of a replicated UPDATE). The default value for the option is 1000. If the data type of a column is longer than the threshold, old values for the column are not verified when an UPDATE is replicated. This keeps the size of SQL Remote messages down, but has the disadvantage that conflicting updates of long values are not detected.

There is a technique allowing detection of conflicts when verify_threshold is being used to reduce the size of messages. Whenever a "BLOB" is updated, a last_modified column in the same table should also be updated. Conflicts can then be detected because the old value of the last_modified column is verified.

Using a work table to avoid redundant updates

Repeated updates to a BLOB should be done in a "work" table, and the final version should be assigned to the replicated table. For example, if a document in progress is updated 20 times throughout the day and the Message Agent is run once at the end of the day, all 20 updates are replicated. If the document is 200 KB in length, this causes 4 MB of messages to be sent.

The better solution is to have a **document_in_progress** table. When the user is done revising a document, the application moves it from the **document_in_progress** table to the replicated table. The results in a single update (200 kb of messages).

Controlling replication of BLOBs

The SQL Anywhere blob_threshold option allows further control over the replication of long values. Any value longer than the blob_threshold option is replicated as a BLOB. That is, it is broken into pieces and replicated in chunks, before being reconstituted by using a SQL variable and concatenating the pieces at the recipient site.

Replication of dates and times

When date or time columns are replicated, the Message Agent uses the setting of the `sr_date_format`, `sr_time_format`, and `sr_timestamp_format` database options to format the date.

For example, the following option setting instructs the Message Agent to send a date of May 2, 1987 as 1987-05-02.

```
SET OPTION sr_date_format = 'yyyy-mm-dd'
```

☞ For more information, see [“SQL Remote options” on page 163](#).

The following points may be useful when replicating dates and times:

- ◆ The time, date, and timestamp formats must be consistent throughout the installation.
- ◆ Ensure that the order of year, month, and day used for the date and timestamp formats matches the setting of the `date_order` database option.

You can change the `date_order` option for the duration of each connection.

Who gets what?

Each time a row in a table is inserted, deleted, or updated, a message has to be sent to those subscribed to the row. In addition, an update may cause the subscription expression to change, so that the statement is sent to some subscribers as a delete, some as an update, and some as an insert.

☞ For details of what statements get sent to which subscribers, see [“How statements are replicated” on page 19](#). For details on subscriptions, see the following two chapters.

This section describes how SQL Remote sends the right operations to the right recipients.

The task of determining who gets what is divided between the database server and the Message Agent. The engine handles those aspects that are to do with publications, while the Message Agent handles aspects to do with subscriptions.

SQL Anywhere actions

SQL Anywhere evaluates the subscription expression for each update made to a table that is part of a publication. It adds the value of the expression to the log, both before and after the update.

For a table that is part of more than one publication, the subscription expression is evaluated before and after the update for each publication.

The addition of information to the log can affect performance in the following cases:

- ◆ **Expensive expressions** When a subscription expression is expensive to evaluate, it can affect performance.
- ◆ **Many publications** When a table belongs to many publications, many expressions must be evaluated. In contrast, the number of *subscriptions* is irrelevant.
- ◆ **Many-valued expressions** Some expressions are many-valued. This can lead to much additional information in the transaction log, with a corresponding effect on performance.

Message Agent actions

The Message Agent reads the evaluated subscription expressions or subscription column entries from the transaction log, and matches the before and after values against the subscription value for each subscriber to the publication. In this way, the Message Agent can send the correct operations to each subscriber.

While large numbers of subscribers do not have any impact on server performance, they can impact Message Agent performance. Both the work in matching subscription values against large numbers of subscription values, and the work in sending the messages, can be demanding.

Replication errors and conflicts

SQL Remote is designed to allow databases to be updated at many different sites. Careful design is required to avoid replication errors, especially if the database has a complicated structure. This section describes the kinds of errors and conflict that can occur in a replication setup; subsequent sections describe how you can design your publications to avoid errors and manage conflicts.

Delivery errors not discussed here

This section does not discuss issues related to message delivery failures. For information on delivery errors and how they are handled, see [“The message tracking system”](#) on page 120

Replication errors

Replication errors fall into the following categories:

- ◆ **Duplicate primary key errors** Two users INSERT a row using the same primary key values, or one user updates a primary key and a second user inserts a primary key of the new value. The second operation to reach a given database in the replication system fails because it would produce a duplicate primary key.
- ◆ **Row not found errors** A user DELETES a row (that is, the row with a given primary key value). A second user UPDATES or DELETES the same row at another site.

In this case, the second statement fails, as the row is not found.

- ◆ **Referential integrity errors** If a column containing a foreign key is included in a publication, but the associated primary key is not included, the extraction utility leaves the foreign key definition out of the remote database so that INSERTS at the remote database will not fail.

This can be solved by including proper defaults into the table definitions.

Also, referential integrity errors can occur when a primary table has a SUBSCRIBE BY expression and the associated foreign table does not: rows from the foreign table may be replicated, but the rows from the primary table may be excluded from the publication.

Replication conflicts

Replication conflicts are different from errors. Properly handled, conflicts are not a problem in SQL Remote.

- ◆ **Conflicts** A user updates a row. A second user updates the same row at another site. The second user's operation succeeds, and SQL Remote allows a trigger to be fired to resolve these conflicts in a way that makes sense for the data being changed.

Conflicts will occur in many installations. SQL Remote allows appropriate resolution of conflicts as part of the regular operation of a SQL Remote setup, using triggers and procedures.

For information about how SQL Remote handles conflicts as they occur, see the following chapters.

Tracking SQL errors

SQL errors in replication must be designed out of your setup. SQL Remote includes an option to help you track errors in SQL statements, but this option is not intended to resolve such errors.

By setting the `replication_error` option, you can specify a stored procedure to be called by the Message Agent when a SQL error occurs. By default no procedure is called.

◆ To set the `replication_error` option

- Issue the following statement:

```
SET OPTION
remote-user.replication_error
= 'procedure-name'
```

where *remote-user* is the user ID on the Message Agent command line and *procedure-name* is the procedure called when a SQL error is detected.

Replication error procedure requirements

The replication error procedure must have a single argument of type CHAR, VARCHAR, or LONG VARCHAR. The procedure is called once with the SQL error message and once with the SQL statement that causes the error.

CHAPTER 4

SQL Remote Design

Contents

Design overview	30
Publishing data	31
Publication design	39
Partitioning tables that do not contain the subscription expression	42
Sharing rows among several subscriptions	48
Managing conflicts	55
Ensuring unique primary keys	63
Creating subscriptions	71

About this chapter

This chapter describes how to design a SQL Remote installation when the consolidated database is a SQL Anywhere database.

Design overview

Designing a SQL Remote installation includes the following tasks:

- ◆ **Designing publications** The publications determine what information is shared among which databases.
- ◆ **Designing subscriptions** The subscriptions determine what information each user receives.
- ◆ **Implementing the design** Creating publications and subscriptions for all users in the system.

All administration is at the consolidated database

Like all SQL Remote administrative tasks, design is carried out by a database administrator or system administrator at the consolidated database.

The SQL Anywhere Database Administrator should perform all SQL Remote configuration tasks.

Publishing data

This section describes how to create simple publications consisting of whole tables, or of column-wise subsets of tables; these tables are also called articles. You can perform these tasks using Sybase Central or with the CREATE PUBLICATION statement in Interactive SQL.

All publications in Sybase Central appear in the Publications folder. Any articles you create for a publication appear on the Articles tab in the right pane when a publication is selected.

Each publication can contain one or more entire tables, but partial tables are also permitted. A table can be subdivided by columns, rows, or both.

Publishing whole tables

The simplest publication you can make consists of a single article, which consists of all rows and columns of one or more tables. These tables must already exist.

◆ To publish one or more entire tables (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. In the left pane, select the Publications folder.
3. From the File menu, choose New ► Publication.

The Create Publication wizard appears.

4. Type a name for the publication. Click Next.
5. On the Tables tab, select a table from the list of Available tables. Click Add. The table appears in the list of Selected Tables on the right.
6. Optionally, you may add additional tables. The order of the tables is not important.
7. Click Finish.

◆ To publish one or more entire tables (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE PUBLICATION statement that specifies the name of the new publication and the table you want to publish.

Example

- ◆ The following statement creates a publication that publishes the whole Customers table:

```
CREATE PUBLICATION PubCustomers (  
    TABLE Customers  
)
```

- ◆ The following statement creates a publication including all columns and rows in each of a set of tables:

```
CREATE PUBLICATION PubSales (  
    TABLE Customers,  
    TABLE SalesOrders,  
    TABLE SalesOrderItems,  
    TABLE Products  
)
```

☞ For more information, see the [“CREATE PUBLICATION statement \[MobiLink\] \[SQL Remote\]” \[SQL Anywhere Server - SQL Reference\]](#).

Publishing only some columns in a table

You can create a publication that contains all the rows, but only some of the columns, of a table from Sybase Central or by listing the columns in the CREATE PUBLICATION statement.

◆ To publish only some columns in a table (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. In the left pane, select the Publications folder.
3. From the File menu, choose New ► Publication.
The Create Publication wizard appears.
4. Type a name for the new publication. Click Next.
5. On the Tables tab, select a table from the list of Available tables. Click Add. The table is added to the list of Selected Tables on the right.
6. On the Columns tab, double-click the table's icon to expand the list of Available Columns. Select each column you want to publish and click Add. The selected columns appear on the right in the Selected Columns list.
7. Click Finish.

◆ To publish only some columns in a table (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE PUBLICATION statement that specifies the publication name and the table name. List the published columns in parenthesis following the table name.

Example

- ◆ The following statement creates a publication that publishes all rows of the ID, CompanyName, and City columns of the Customers table:

```
CREATE PUBLICATION PubCustomers (  
    TABLE Customers (  
        ID,  
        CompanyName,  
        City )  
)
```


☞ For more information, see the “[CREATE PUBLICATION statement \[MobiLink\] \[SQL Remote\]](#)” [*SQL Anywhere Server - SQL Reference*].

Publishing only some rows in a table

You can create a publication that contains all the columns, but only some of the rows, of a table from Sybase Central. In either case, you do so by writing a search condition that matches only the rows you want to publish.

Sybase Central and the SQL language provide two ways of publishing only some of the rows in a table; however, only one way is compatible with MobiLink.

- ◆ **WHERE clause** You can use a WHERE clause to include a subset of rows in an article. All subscribers to the publication containing this article receive the rows that satisfy the WHERE clause.
- ◆ **Subscription expression** You can use a subscription expression to include a different set of rows in different subscriptions to publications containing the article.

You can combine a WHERE clause and a subscription expression in an article. You can specify them in Sybase Central or in a CREATE PUBLICATION statement.

Use the Subscription expression when different subscribers to a publication are to receive different rows from a table. The Subscription expression is the most powerful method of partitioning tables.

Use the WHERE clause to exclude the same set of rows from all subscriptions to a publication.

Publishing only some rows using a WHERE clause

You can specify a WHERE clause to include in the publication only the rows that satisfy the WHERE conditions.

◆ To create a publication using a WHERE clause (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. In the left pane, select the Publications folder.
3. From the File menu, choose New ► Publication.
The Create Publication wizard appears.
4. Type a name for the new publication. Click Next.
5. On the Tables tab, select a table from the list of Available tables. Click Add. The table is added to the list of Selected Tables on the right.
6. On the WHERE Clauses tab, select the table then type the search condition in the lower box.
7. Click Finish.

◆ To create a publication using a WHERE clause (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE PUBLICATION statement that includes the rows you wish to include in the publication and a WHERE condition.

Examples

- ◆ The following statement creates a publication that publishes the ID, CompanyName, City, State, and Country columns of the Customers table, for the customers marked as active in the status column.

```
CREATE PUBLICATION PubCustomers (  
    TABLE Customers (  
        ID,  
        CompanyName,  
        City,  
        State,  
        Country )  
    WHERE Status = 'active'  
)
```

In this case, the status column is not published. All unpublished rows must have a default value. Otherwise, an error occurs when rows are downloaded for insert from the consolidated database.

- ◆ The following is a single-article publication sending relevant order information to Samuel Singer, a sales rep:

```
CREATE PUBLICATION PubOrdersSamuelSinger (  
    TABLE SalesOrders WHERE SalesRepresentative = 856  
)
```

 For more information, see the “CREATE PUBLICATION statement [MobiLink] [SQL Remote]” [SQL Anywhere Server - SQL Reference].

SUBSCRIBE BY

The create publication statement also allows a SUBSCRIBE BY clause. This clause can also be used to selectively publish rows in SQL Remote. However, it is ignored during MobiLink synchronization.

Publishing only some rows using a subscription expression

You can specify a subscription expression to include a different set of rows in different subscriptions to publications containing the article.

For example, in a mobile workforce situation, a sales publication may be wanted where each sales rep subscribes to their own sales orders, enabling them to update their sales orders locally and replicate the sales to the consolidated database.

Using the WHERE clause model, a separate publication for each sales rep would be needed: the following publication is for sales rep Samuel Singer: each of the other sales reps would need a similar publication.

```
CREATE PUBLICATION PubOrdersSamuelSinger (  
    TABLE SalesOrders
```

```

        WHERE SalesRepresentative = 856
    )

```

To address the needs of setups requiring large numbers of different subscriptions, SQL Remote allows a **subscription expression** to be associated with an article. Subscriptions receive rows depending on the value of a supplied expression.

Benefits of subscription expressions

Publications using a subscription expression are more compact, easier to understand, and provide better performance than maintaining several WHERE clause publications. The database server must add information to the transaction log, and scan the transaction log to send messages, in direct proportion to the number of publications. The subscription expression allows many different subscriptions to be associated with a single publication, whereas the WHERE clause does not.

◆ To create an article using a subscription expression (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. In the left pane, select the Publications folder.
3. From the File menu, choose New ► Publication.
The Create Publication wizard appears.
4. Type a name for the publication and click Next.
5. On the Specify SUBSCRIBE BY Restrictions page, enter the subscription expression.
6. Follow the remaining instructions in the wizard.

◆ To create an article using a subscription expression (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE PUBLICATION statement that includes the expression you wish to use as a match in the subscription expression.

Examples

- ◆ The following statement creates a publication that publishes the ID, CompanyName, City, State, and Country columns of the Customers table, and which matches the rows with subscribers according to the value of the State column:

```

CREATE PUBLICATION PubCustomers (
    TABLE Customers (
        ID,
        CompanyName,
        City,
        State,
        Country )
    SUBSCRIBE BY State
)

```

- ◆ The following statements subscribe two employees to the publication: Ann Taylor receives the customers in Georgia (GA), and Sam Singer receives the customers in Massachusetts (MA).

```
CREATE SUBSCRIPTION
TO PubCustomers ('GA')
FOR Ann_Taylor ;
CREATE SUBSCRIPTION
TO PubCustomers ('MA')
FOR Sam_Singer
```

Users can subscribe to more than one publication, and can have more than one subscription to a single publication.

See also

- ◆ [“CREATE PUBLICATION statement \[MobiLink\] \[SQL Remote\]” \[SQL Anywhere Server - SQL Reference\]](#)
- ◆ [“Partitioning tables that do not contain the subscription expression” on page 42](#)
- ◆ [“Creating subscriptions” on page 71](#)
- ◆ [“Publishing only some rows using a WHERE clause” on page 33](#)
- ◆ [“Altering existing publications” on page 36](#)

Altering existing publications

After you have created a publication, you can alter it by adding, modifying, or deleting articles, or by renaming the publication. If an article is modified, the entire specification of the modified article must be entered.

You can perform these tasks using Sybase Central or with the ALTER PUBLICATION statement in Interactive SQL.

◆ To modify the properties of existing publications or articles (Sybase Central)

1. Connect to the database as a user who owns the publication or as a user with DBA authority.
2. Right-click the publication or article and choose Properties from the popup menu.
3. Configure the desired properties.

◆ To add articles (Sybase Central)

1. Connect to the database as a user who owns the publication or as a user with DBA authority.
2. In the left pane, open the Publications folder.
3. Select the publication you want to add an article to.
4. From the File menu, choose New ► Article.
The Create Article wizard appears.
5. In the Create Article wizard, do the following:
 - ◆ Choose a table and click Next.

- ◆ Choose the columns for the article. Click Next.
 - ◆ Enter a WHERE clause (if desired). Click Next.
 - ◆ Create a SUBSCRIBE BY restriction (if desired).
6. Click Finish to create the article.

◆ To remove articles (Sybase Central)

1. Connect to the database as a user who owns the publication or as a user with DBA authority.
2. Open the Publications folder.
3. Select the publication you want to remove an article from.
4. Right-click the article you want to delete and choose Delete from the popup menu.

◆ To modify an existing publication (SQL)

1. Connect to the database as a user who owns the publication or as a user with DBA authority.
2. Execute an ALTER PUBLICATION statement.

Example

- ◆ The following statement adds the Customers table to the PubContacts publication.

```
ALTER PUBLICATION PubContacts (  
    ADD TABLE Customers  
)
```

See also

- ◆ [“ALTER PUBLICATION statement \[MobiLink\] \[SQL Remote\]” \[SQL Anywhere Server - SQL Reference\]](#)
- ◆ [“Publishing only some rows using a WHERE clause” on page 33](#)
- ◆ [“Publishing only some rows using a subscription expression” on page 34](#)

Dropping publications

You can drop a publication using either Sybase Central or the DROP PUBLICATION statement. If you drop a publication, all subscriptions to that publication are automatically deleted as well.

You must have DBA authority to drop a publication.

◆ To delete a publication (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. Open the Publications folder.
3. Right-click the desired publication and choose Delete from the popup menu.

◆ To delete a publication (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a DROP PUBLICATION statement.

Example

The following statement drops the publication named PubOrders.

```
DROP PUBLICATION PubOrders
```

☞ See also the “[DROP PUBLICATION statement \[MobiLink\] \[SQL Remote\]](#)” [*SQL Anywhere Server - SQL Reference*].

Notes on publications

- ◆ The different publication types described above can be combined. A single publication can publish a subset of columns from a set of tables and use a WHERE clause to select a set of rows to be replicated.
- ◆ DBA authority is required to create and drop publications.
- ◆ Publications can be altered only by the DBA or the publication's owner.
- ◆ Altering publications in a running SQL Remote setup is likely to cause replication errors and can lead to loss of data in the replication system unless carried out with care.
- ◆ Views cannot be included in publications.
- ◆ Stored procedures cannot be included in publications. For a discussion of how SQL Remote replicates procedures and triggers, see “[Replication of procedures](#)” on page 21.

Publication design

Once you understand how to create simple publications, you must think about proper publication design. Sound design is an important part of building a successful SQL Remote installation. This section helps set out the principles of sound design as they apply to SQL Remote for SQL Anywhere.

Design issues overview

Each subscription must be a complete relational database

A remote database shares with the consolidated database the information in their subscriptions. The subscription is both a subset of the relational database held at the consolidated site, and also a complete relational database at the remote site. The information in the subscription is therefore subject to the same rules as any other relational database:

- ◆ **Foreign key relationships must be valid** For every entry in a foreign key, a corresponding primary key entry must exist in the database.

The database extraction utility ensures that the CREATE TABLE statements for remote databases do not have foreign keys defined to tables that do not exist remotely.

- ◆ **Primary key uniqueness must be maintained** There is no way of checking what new rows have been entered at other sites, but not yet replicated. The design must prevent users at different sites adding rows with identical primary key values, as this would lead to conflicts when the rows are replicated to the consolidated database.

Transaction integrity must be maintained in the absence of locking

The data in the dispersed database (which consists of the consolidated database and all remote databases) must maintain its integrity in the face of updates at all sites, even though there is no system-wide locking mechanism for any particular row.

- ◆ **Locking conflicts must be prevented or resolved** In a SQL Remote installation, there is no method for locking rows across all databases to prevent different users from altering the rows at the same time. Such conflicts must be prevented by designing them out of the system or must be resolved in an appropriate manner at the consolidated database.

These key features of relational databases must be incorporated into the design of your publications and subscriptions. This section describes principles and techniques for sound design.

Conditions for valid articles

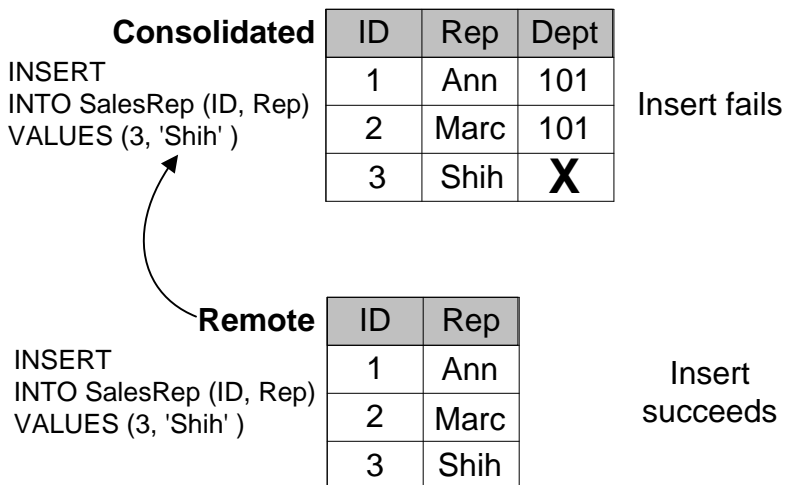
All columns in the primary key must be included in the article.

Supporting INSERTS at remote databases

For INSERT statements at a remote database to replicate correctly to the consolidated database, you can exclude from an article only columns that can be left out of a valid INSERT statement. These are:

- ◆ Columns that allow NULL.
- ◆ Columns that have defaults.

If you exclude any column that does not satisfy one of these requirements, INSERT statements carried out at a remote database will fail when replicated to the consolidated database.



Using BEFORE triggers as an alternative
 An exception to this case is when the consolidated database is a SQL Anywhere database, and a BEFORE trigger has been written to maintain the columns that are not included in the INSERT statement.

Design tips for performance

This section presents a checklist for designing high performance SQL Remote installations.

- ◆ **Keep the number of publications small** In particular, try not to reference the same table in many different publications.

The work the database server needs to do is proportional to the number of publications. Keeping the number low and making effective use of subscriptions lightens the load on the database server.

When operations occur on a table, the database server and the Message Agent must do some work for each publication that contains the table. Having one publication for each remote user will drastically increase the load on the database server. It is much better to have a few publications that use SUBSCRIBE BY and have subscriptions for each remote user. The database server does no additional work when more subscriptions are added for a publication. The Message Agent is designed to work efficiently with a large number of subscriptions.

- ◆ **Group publications logically** For example, if there is a table that every remote user requires, such as a price list table, make a separate publication for that table. Make one publication for each table where the data can be partitioned by a column value.
- ◆ **Use subscriptions effectively** When remote users receive similar subsets of the consolidated database, always use publications that incorporate SUBSCRIBE BY expressions. Do not create a separate publication for each remote user.
- ◆ **Pay attention to Update Publication Triggers** In particular:
 - ◆ Use the NEW / OLD SUBSCRIBE BY syntax.
 - ◆ Tune the SELECT statements to ensure they are accessing the database efficiently.
- ◆ **Monitor the transaction log size** The larger the transaction log, the longer it takes the Message Agent to scan it. Rename the log regularly and use the delete_old_logs option.

Partitioning tables that do not contain the subscription expression

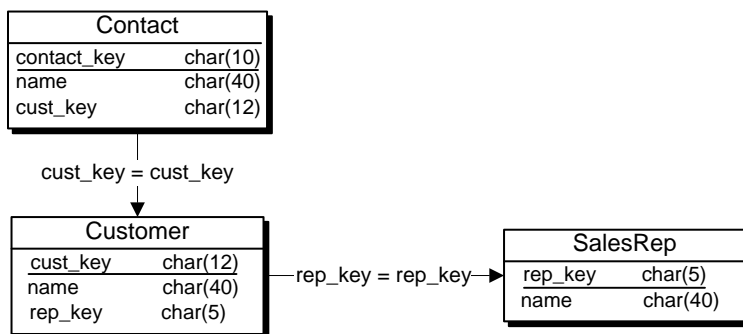
In many cases, the rows of a table need to be partitioned even when the subscription expression does not exist in the table.

The Contacts example

The Contacts database illustrates why and how to partition tables that do not contain the subscription expression.

Example

Here is a simple database that illustrates the problem.



Each sales representative sells to several customers. At some customers there is a single contact, while other customers have several contacts.

The tables in the database

The three tables are described in more detail as follows:

Table	Description
SalesReps	<p>All sales representatives that work for the company. The SalesReps table has the following columns:</p> <ul style="list-style-type: none"> ◆ rep_key An identifier for each sales representative. This is the primary key. ◆ name The name of each sales representative. <p>The SQL statement creating this table is as follows:</p> <pre> CREATE TABLE SalesReps (Rep_key CHAR(12) NOT NULL, Name CHAR(40) NOT NULL, PRIMARY KEY (rep_key)) </pre>

Table	Description
Customers	<p>All customers that do business with the company. The Customers table includes the following columns:</p> <ul style="list-style-type: none"> ◆ cust_key An identifier for each customer. This is the primary key. ◆ name The name of each customer. ◆ rep_key An identifier for the sales representative in a sales relationship. This is a foreign key to the SalesReps table. <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE Customers (Cust_key CHAR(12) NOT NULL, Name CHAR(40) NOT NULL, Rep_key CHAR(12) NOT NULL, FOREIGN KEY REFERENCES SalesReps, PRIMARY KEY (cust_key))</pre>
Contacts	<p>All individual contacts that do business with the company. Each contact belongs to a single customer. The Contacts table includes the following columns:</p> <ul style="list-style-type: none"> ◆ contact_key An identifier for each contact. This is the primary key. ◆ name The name of each contact. ◆ cust_key An identifier for the customer to which the contact belongs. This is a foreign key to the Customers table. <p>The SQL statement creating this table is:</p> <pre>CREATE TABLE Contacts (Contacts_key CHAR(12) NOT NULL, Name CHAR(40) NOT NULL, Cust_key CHAR(12) NOT NULL, FOREIGN KEY REFERENCES Customers, PRIMARY KEY (contact_key))</pre>

Replication goals

The goals of the design are to provide each sales representative with the following information:

- ◆ The complete **SalesReps** table.
- ◆ Those customers assigned to them, from the **Customers** table.
- ◆ Those contacts belonging to the relevant customers, from the **Contacts** table.

Partitioning the Customers table in the Contacts example

The **Customers** table can be partitioned using the **rep_key** value as a subscription expression. A publication that includes the **SalesReps** and **Customers** tables would be as follows:

```
CREATE PUBLICATION SalesRepData (  
    TABLE SalesReps  
    TABLE Customers SUBSCRIBE BY rep_key  
)
```

Partitioning the **Contacts** table in the **Contacts** example

The **Contacts** table must also be partitioned among the sales representatives, but contains no reference to the sales representative **rep_key** value. How can the Message Agent match a subscription value against rows of this table, when **rep_key** is not present in the table?

To solve this problem, you can use a subquery in the **Contacts** article that evaluates to the **rep_key** column of the **Customers** table. The publication then looks like this:

```
CREATE PUBLICATION SalesRepData (  
    TABLE SalesReps  
    TABLE Customers  
        SUBSCRIBE BY rep_key  
    TABLE Contacts  
        SUBSCRIBE BY (SELECT rep_key  
            FROM Customers  
            WHERE Contacts.cust_key = Customers.cust_key )  
)
```

The WHERE clause in the subscription expression ensures that the subquery returns only a single value, as only one row in the **Customers** table has the **cust_key** value in the current row of the **Contacts** table.

Territory realignment in the **Contacts** example

In **territory realignment**, rows are reassigned among subscribers. In the present case, territory realignment is the reassignment of rows in the **Customers** table, and by implication also the **Contacts** table, among the Sales Reps.

When a customer is reassigned to a new sales rep, the **Customers** table is updated. The UPDATE is replicated as an INSERT or a or a DELETE to the old and new sales representatives, respectively, so that the customer row is properly transferred to the new sales representative.

☞ For information on the way in which SQL Anywhere and SQL Remote work together to handle this situation, see [“Who gets what?” on page 25](#).

When a customer is reassigned, the **Contacts** table is unaffected. There are no changes to the **Contacts** table, and consequently no entries in the transaction log pertaining to the **Contacts** table. In the absence of this information, SQL Remote cannot reassign the rows of the **Contacts** table along with the **Customers**.

This failure will cause referential integrity problems: the **Contacts** table at the remote database of the old sales representative contains a **cust_key** value for which there is no longer a **Customers**.

Use triggers to maintain **Contacts**

The solution is to use a trigger containing a special form of UPDATE statement, which does not make any change to the database tables, but which does make an entry in the transaction log. This log entry contains

the before and after values of the subscription expression, and so is of the proper form for the Message Agent to replicate the rows properly.

The trigger must be fired BEFORE operations on the row. In this way, the BEFORE value can be evaluated and placed in the log. Also, the trigger must be fired FOR EACH ROW rather than for each statement, and the information provided by the trigger must be the new subscription expression. The Message Agent can use this information to determine which subscribers receive which rows.

Trigger definition

The trigger definition is as follows:

```
CREATE TRIGGER UpdateCustomer
BEFORE UPDATE ON Customers
REFERENCING NEW AS NewRow
      OLD as OldRow
FOR EACH ROW
BEGIN
    // determine the new subscription expression
    // for the Customers table
    UPDATE Contacts
    PUBLICATION SalesRepData
    OLD SUBSCRIBE BY ( OldRow.rep_key )
    NEW SUBSCRIBE BY ( NewRow.rep_key )
    WHERE cust_key = NewRow.cust_key;
END;
```

A special UPDATE statement for publications

The UPDATE statement in this trigger is of the following special form:

```
UPDATE table-name
PUBLICATION publication-name
{ SUBSCRIBE BY subscription-expression |
  OLD SUBSCRIBE BY old-subscription-expression
  NEW SUBSCRIBE BY new-subscription-expression }
WHERE search-condition
```

- ◆ Here is what the UPDATE statement clauses mean:
- ◆ The *table-name* indicates the table that must be modified at the remote databases.
- ◆ The *publication-name* indicates the publication for which subscriptions must be changed.
- ◆ The value of *subscription-expression* is used by the Message Agent to determine both new and existing recipients of the rows. Alternatively, you can provide both OLD and NEW subscription expressions.
- ◆ The WHERE clause specifies which rows are to be transferred between subscribed databases.

Notes on the trigger

- ◆ If the trigger uses the following syntax:

```
UPDATE table-name
PUBLICATION pub-name
      SUBSCRIBE BY sub-expression
WHERE search-condition
```

the trigger must be a BEFORE trigger. In this case, a BEFORE UPDATE trigger. In other contexts, BEFORE DELETE and BEFORE INSERT are necessary.

- ◆ If the trigger uses the alternate syntax:

```
UPDATE table-name
PUBLICATION publication-name
  OLD SUBSCRIBE BY old-subscription-expression
  NEW SUBSCRIBE BY new-subscription-expression }
WHERE search-condition
```

The trigger can be a BEFORE or AFTER trigger.

- ◆ The UPDATE statement lists the publication and table that is affected. The WHERE clause in the statement describes the rows that are affected. No changes are made to the data in the table itself by this UPDATE, it makes entries in the transaction log.
- ◆ The subscription expression in this example returns a single value. Subqueries returning multiple values can also be used. The value of the subscription expression must be the value after the UPDATE.

In this case, the only subscriber to the row is the new sales representative. In [“Sharing rows among several subscriptions” on page 48](#), there are existing as well as new subscribers.

Information in the transaction log

Understanding the information in the transaction log helps in designing efficient publications.

- ◆ Assume the following data:
 - ◆ SalesReps table

rep_key	Name
rep1	Ann
rep2	Marc

- ◆ Customers table

cust_key	name	rep_key
cust1	Sybase	rep1
cust2	SQL Anywhere	rep2

- ◆ Contacts table

contact_key	name	cust_key
contact1	David	cust1
contact2	Stefanie	cust2

- ◆ Now apply the following territory realignment Update statement

```
UPDATE Customers
SET rep_key = 'rep2'
WHERE cust_key = 'cust1'
```

The transaction log would contain two entries arising from this statement: one for the BEFORE trigger on the Contacts table, and one for the actual UPDATE to the Customers table.

```
SalesRepData - Publication Name
rep1 - BEFORE list
rep2 - AFTER list
UPDATE Contacts
SET contact_key = 'contact1',
   name = 'David',
   cust_key = 'cust1'
WHERE contact_key = 'contact1'
SalesRepData - Publication Name
rep1 - BEFORE list
rep2 - AFTER list
UPDATE Customers
SET rep_key = 'rep2'
WHERE cust_key = 'cust1'
```

The Message Agent scans the log for these tags. Based on this information it can determine which remote users get an INSERT, UPDATE or DELETE.

In this case, the BEFORE list was **rep1** and the AFTER list is **rep2**. If the before and after list values are different, the rows affected by the UPDATE statement have "moved" from one subscriber value to another. This means the Message Agent will send a DELETE to all remote users who subscribed by the value **rep1** for the Customers record **cust1** and send an INSERT to all remote users who subscribed by the value **rep2**.

If the BEFORE and AFTER lists are identical, the remote user already has the row and an UPDATE will be sent.

Sharing rows among several subscriptions

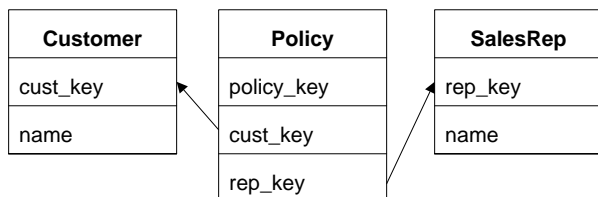
There are cases where a row may need to be included in several subscriptions, such as in a many-to-many relationship. This section uses a case study to illustrate how to handle this situation.

The Policy example

The Policy database illustrates why and how to partition tables when there is a many-to-many relationship in the database.

Example database

Here is a simple database that illustrates the problem.



Each sales representative sells to several customers, and some customers deal with more than one sales representative. In this case, the relationship between **Customers** and **SalesReps** is thus a many-to-many relationship.

The tables in the database

The three tables are described in more detail as follows:

Table	Description
SalesReps	<p>All sales representatives that work for the company. The SalesReps table has the following columns:</p> <ul style="list-style-type: none"> ♦ rep_key An identifier for each sales representative. This is the primary key. ♦ name The name of each sales representative. <p>The SQL statement creating this table is as follows:</p> <pre> CREATE TABLE SalesReps (Rep_key CHAR(12) NOT NULL, Name CHAR(40) NOT NULL, PRIMARY KEY (rep_key)); </pre>

Table	Description
Customers	<p>All customers that do business with the company. The Customers table includes the following columns:</p> <ul style="list-style-type: none"> ◆ cust_key A primary key column containing an identifier for each customer ◆ name A column containing the name of each customer <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE Customers (Cust_key CHAR(12) NOT NULL, Name CHAR(40) NOT NULL, PRIMARY KEY (cust_key));</pre>
Policy	<p>A three-column table that maintains the many-to-many relationship between customers and sales representatives. The Policy table has the following columns:</p> <ul style="list-style-type: none"> ◆ policy_key A primary key column containing an identifier for the sales relationship. ◆ cust_key A column containing an identifier for the customer representative in a sales relationship. ◆ rep_key A column containing an identifier for the sales representative in a sales relationship. <p>The SQL statement creating this table is as follows.</p> <pre>CREATE TABLE Policy (policy_key CHAR(12) NOT NULL, cust_key CHAR(12) NOT NULL, rep_key CHAR(12) NOT NULL, FOREIGN KEY (cust_key) REFERENCES Customers (cust_key) FOREIGN KEY (rep_key) REFERENCES SalesReps (rep_key), PRIMARY KEY (policy_key));</pre>

Replication goals

The goals of the replication design are to provide each sales representative with the following information:

- ◆ The entire **SalesReps** table.
- ◆ Those rows from the **Policy** table that include sales relationships involving the sales rep subscribed to the data.
- ◆ Those rows from the **Customers** table listing customers that deal with the sales rep subscribed to the data.

New problems

The many-to-many relationship between customers and sales representatives introduces new challenges in maintaining a proper sharing of information:

- ◆ There is a table (in this case the Customers table) that has no reference to the sales representative value that is used in the subscriptions to partition the data.

Again, this problem is addressed by using a subquery in the publication.

- ◆ Each row in the **Customers** table may be related to many rows in the **SalesReps** table, and shared with many sales representatives databases.

Put another way, the rows of the **Contacts** table in [“Partitioning tables that do not contain the subscription expression” on page 42](#) were partitioned into disjoint sets by the publication. In the present example there are overlapping subscriptions.

To meet the replication goals, you need one publication and a set of subscriptions. In this case, you can use two triggers to handle the transfer of customers from one sales representative to another.

The publication

A single publication provides the basis for the data sharing:

```
CREATE PUBLICATION SalesRepData (  
    TABLE SalesReps,  
    TABLE Policy SUBSCRIBE BY rep_key,  
    TABLE Customers SUBSCRIBE BY (  
        SELECT rep_key FROM Policy  
        WHERE Policy.cust_key =  
            Customers.cust_key  
    ),  
);
```

The subscription statements are exactly as in the previous example.

How the publication works

The publication includes part or all of each of the three tables. To understand how the publication works, it helps to look at each article in turn:

- ◆ **SalesReps table** There are no qualifiers to this article, so the entire **SalesReps** table is included in the publication.

```
... TABLE SalesReps,  
...
```

- ◆ **Policy table** This article uses a subscription expression to specify a column used to partition the data among the sales reps:

```
... TABLE Policy  
    SUBSCRIBE BY rep_key,  
...
```

The subscription expression ensures that each sales rep receives only those rows of the table for which the value of the **rep_key** column matches the value provided in the subscription.

The **Policy** table partitioning is **disjoint**: there are no rows that are shared with more than one subscriber.

Customers table A subscription expression with a subquery is used to define the partition. The article is defined as follows:

```
... TABLE Customers SUBSCRIBE BY (  
    SELECT rep_key  
    FROM Policy  
    WHERE Policy.cust_key =  
           Customers.cust_key  
    ),  
...
```

The **Customers** partitioning is **non-disjoint**: some rows are shared with more than one subscriber.

Multiple-valued subqueries in publications

The subquery in the **Customers** article returns a single column (**rep_key**) in its result set, but may return multiple rows, corresponding to all those sales representatives that deal with the particular customer. When a subscription expression has multiple values, the row is replicated to all subscribers whose subscription matches any of the values. It is this ability to have multiple-valued subscription expressions that allows non-disjoint partitionings of a table.

Territory realignment with a many-to-many relationship

The problem of territory realignment (reassigning rows among subscribers) requires special attention, just as in the section [“Territory realignment in the Contacts example” on page 44](#).

You need to write triggers to maintain proper data throughout the installation when territory realignment (reassignment of rows among subscribers) is allowed.

How customers are transferred

This example requires that a customer transfer be achieved by deleting and inserting rows in the **Policy** table.

To cancel a sales relationship between a customer and a sales representative, a row in the **Policy** table is deleted. In this case, the **Policy** table change is properly replicated to the sales representative, and the row no longer appears in their database. However, no change has been made to the **Customers** table, and so no changes to the **Customers** table are replicated to the subscriber.

In the absence of triggers, this would leave the subscriber with incorrect data in their **Customers** table. The same kind of problem arises when a new row is added to the **Policy** table.

Using Triggers to solve the problem

The solution is to write triggers that are fired by changes to the **Policy** table, which include a special syntax of the UPDATE statement. The special UPDATE statement makes no changes to the database tables, but does make an entry in the transaction log that SQL Remote uses to maintain data in subscriber databases.

A BEFORE INSERT trigger

Here is a trigger that tracks INSERTS into the **Policy** table, and ensures that remote databases contain the proper data.

```
CREATE TRIGGER InsPolicy  
BEFORE INSERT ON Policy
```

```
REFERENCING NEW AS NewRow
FOR EACH ROW
BEGIN
    UPDATE Customers
    PUBLICATION SalesRepData
    SUBSCRIBE BY (
        SELECT rep_key
        FROM Policy
        WHERE cust_key = NewRow.cust_key
        UNION ALL
        SELECT NewRow.rep_key
    )
    WHERE cust_key = NewRow.cust_key;
END;
```

A BEFORE DELETE trigger

Here is a corresponding trigger that tracks DELETES from the **Policy** table:

```
CREATE TRIGGER DelPolicy
BEFORE DELETE ON Policy
REFERENCING OLD AS OldRow
FOR EACH ROW
BEGIN
    UPDATE Customers
    PUBLICATION SalesRepData
    SUBSCRIBE BY (
        SELECT rep_key
        FROM Policy
        WHERE cust_key = OldRow.cust_key
        AND Policy_key <> OldRow.Policy_key
    )
    WHERE cust_key = OldRow.cust_key;
END;
```

Some of the features of the trigger are the same as in the previous section. The major new features are that the INSERT trigger contains a subquery, and that this subquery can be multi-valued.

Multiple-valued subqueries

The subquery in the BEFORE INSERT trigger is a UNION expression, and can be multi-valued:

```
...
SELECT rep_key
FROM Policy
WHERE cust_key = NewRow.cust_key
UNION ALL
SELECT NewRow.rep_key
...
```

- ◆ The second part of the UNION is the **rep_key** value for the new sales representative dealing with the customer, taken from the INSERT statement.
- ◆ The first part of the UNION is the set of existing sales representatives dealing with the customer, taken from the Policy table.

This illustrates the point that the result set of the subscription query must be all those sales representatives receiving the row, not just the new sales representatives.

The subquery in the BEFORE DELETE trigger is multi-valued:

```

...
SELECT rep_key
FROM Policy
WHERE cust_key = OldRow.cust_key
AND rep_key <> OldRow.rep_key
...

```

- ◆ The subquery takes **rep_key** values from the **Policy** table. The values include the primary key values of all those sales reps who deal with the customer being transferred (WHERE **cust_key** = **OldRow.cust_key**), with the exception of the one being deleted (AND **rep_key** <> **OldRow.rep_key**).

This again emphasizes that the result set of the subscription query must be all those values matched by sales representatives receiving the row following the DELETE.

Notes

- ◆ Data in the **Customers** table is not identified with an individual subscriber (by a primary key value, for example) and is shared among more than one subscriber. This allows the possibility of the data being updated in more than one remote site between replication messages, which could lead to replication conflicts. You can address this issue either by permissions (allowing only certain users the right to update the Customers table, for example) or by adding RESOLVE UPDATE triggers to the database to handle the conflicts programmatically.
- ◆ UPDATES on the Policy table have not been described here. They should either be prevented, or a BEFORE UPDATE trigger is required that combines features of the BEFORE INSERT and BEFORE DELETE triggers shown in the example.

Using the `subscribe_by_remote` option with many-to-many relationships

When the `subscribe_by_remote` option is On, operations from remote databases on rows with a `subscribe` by value of NULL or an empty string will assume the remote user is subscribed to the row. By default, the `subscribe_by_remote` option is set to On. In most cases, this setting is the desired setting.

The `subscribe_by_remote` option solves a problem that otherwise would arise with some publications, including the Policy example. This section describes the problem, and how the option automatically avoids it.

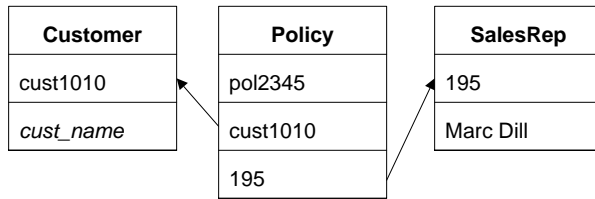
The publication uses a subquery for the **Customers** table subscription expression, because each Customers may belong to several Sales Reps:

```

CREATE PUBLICATION SalesRepData (
    TABLE SalesReps,
    TABLE Policy SUBSCRIBE BY rep_key,
    TABLE Customers SUBSCRIBE BY (
        SELECT rep_key FROM Policy
        WHERE Policy.cust_key =
            Customers.cust_key
    ),
);

```

Marc Dill is a Sales Rep who has just arranged a policy with a new customer. He inserts a new **Customers** row and also inserts a row in the **Policy** table to assign the new Customers to himself.



As the INSERT of the Customers row is carried out by the Message Agent at the consolidated database, SQL Anywhere records the subscription value in the transaction log, at the time of the INSERT.

Later, when the Message Agent scans the log, it builds a list of subscribers from the subscription expression, and Marc Dill is not on the list, as the row in the Policy table assigning the customer to him has not yet been applied. If `subscribe_by_remote` were set to Off, the result would be that the new Customers is sent back to Marc Dill as a DELETE operation.

As long as `subscribe_by_remote` is set to On, the Message Agent assumes the row belongs to the Sales Rep that inserted it, the INSERT is not replicated back to Marc Dill, and the replication system is intact.

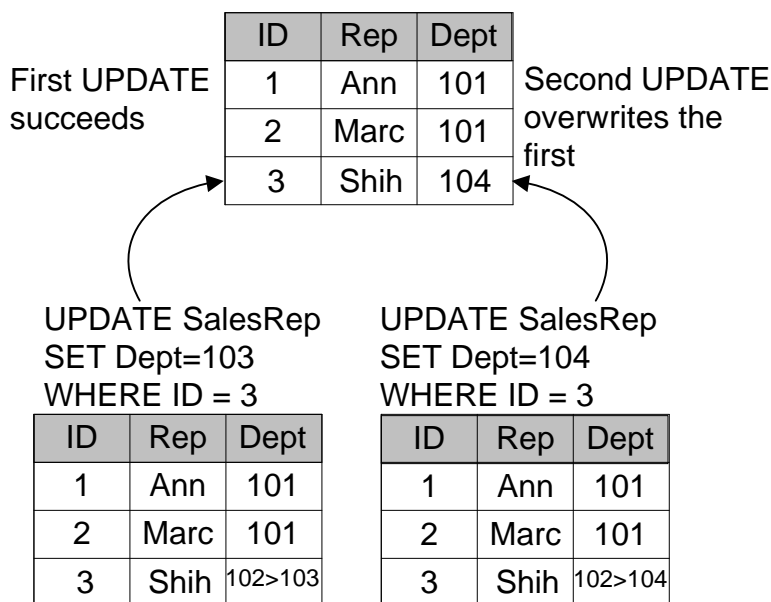
If `subscribe_by_remote` is set to Off, you must ensure that the Policy row is inserted before the Customers row, with the referential integrity violation avoided by postponing checking to the end of the transaction.

Managing conflicts

An UPDATE conflict occurs when the following sequence of events takes place:

1. User 1 updates a row at remote site 1.
2. User 2 updates the same row at remote site 2.
3. The update from User 1 is replicated to the consolidated database.
4. The update from User 2 is replicated to the consolidated database.

When the SQL Remote Message Agent replicates UPDATE statements, it does so as a separate UPDATE for each row. Also, the message contains the old row values for comparison. When the update from user 2 arrives at the consolidated database, the values in the row are not those recorded in the message.



Default conflict resolution

By default, the UPDATE still proceeds, so that the User 2 update (the last to reach the consolidated database) becomes the value in the consolidated database, and is replicated to all other databases subscribed to that row.

In general, the default method of conflict resolution is that the most recent operation (in this case that from User 2) succeeds, and no report is made of the conflict. The update from User 1 is lost. SQL Remote also allows custom conflict resolution, using a trigger to resolve conflicts in a way that makes sense for the data being changed.

Conflict resolution does not apply to primary key updates

UPDATE conflicts do *not* apply to primary key updates. You should not update primary keys in a SQL Remote installation. Primary key conflicts must be excluded from the installation by proper design.

This section describes how you can build conflict resolution into your SQL Remote installation at the consolidated database.

How SQL Remote handles conflicts

When a conflict is detected

SQL Remote replication messages include UPDATE statements as a set of single row updates, each with a VERIFY clause that includes values prior to updating.

An UPDATE conflict is detected by the database server as a failure of the VERIFY clause values to match the rows in the database.

Conflicts are detected and resolved by the Message Agent, but only at a consolidated database. When an UPDATE conflict is detected in a message from a remote database, the Message Agent causes the database server to take two actions:

1. Any conflict resolution (RESOLVE UPDATE) triggers are fired.
2. The UPDATE is applied.

UPDATE statements are applied even if the VERIFY clause values do not match, whether or not there is a RESOLVE UPDATE trigger.

Conflict resolution can take several forms. For example,

- ◆ In some applications, resolution could mean reporting the conflict into a table.
- ◆ You may wish to keep updates made at the consolidated database in preference to those made at remote sites.
- ◆ Conflict resolution can be more sophisticated, for example in resolving inventory numbers in the face of goods deliveries and orders.

Implementing conflict resolution

This section describes what you need to do to implement custom conflict resolution in SQL Remote.

SQL Remote allows you to define **conflict resolution triggers** to handle UPDATE conflicts. Conflict resolution triggers are fired only at a consolidated database, when messages are applied by a remote user. When an UPDATE conflict is detected at a consolidated database, the following sequence of events takes place.

1. Any conflict resolution triggers defined for the operation are fired.

2. The UPDATE takes place.
3. Any actions of the trigger, as well as the UPDATE, are replicated to all remote databases, including the sender of the message that triggered the conflict.

In general, SQL Remote for SQL Anywhere does not replicate the actions of triggers: the trigger is assumed to be present at the remote database. Conflict resolution triggers are fired only at consolidated databases, and so their actions are replicated to remote databases.

4. At remote databases, no RESOLVE UPDATE triggers are fired when a message from a consolidated database contains an UPDATE conflict.
5. The UPDATE is carried out at the remote databases.

At the end of the process, the data is consistent throughout the setup.

UPDATE conflicts cannot happen where data is shared for reading, but each row (as identified by its primary key) is updated at only one site. They only occur when data is being updated at more than one site.

Using conflict resolution triggers

This section describes how to use RESOLVE UPDATE, or **conflict resolution** triggers.

UPDATE statements with a VERIFY clause

Conflict resolution triggers are fired by the failure of values in the VERIFY clause of an UPDATE statement to match the values in the database before the update. An UPDATE statement with a VERIFY clause takes the following form:

```
UPDATE table-list
SET column-name = expression, ...
[ VERIFY (column-name, ...)
  VALUES ( expression, ...) ]
[ WHERE search-condition ]
```

The VERIFY clause can be used only if *table-list* consists of a single table. It compares the values of specified columns to a set of expected values, which are the values that were present in the publisher database when the UPDATE statement was applied there. When the VERIFY clause is specified, only one table can be updated at a time.

The VERIFY clause is useful only for single-row updates. However, multi-row update statements entered at a database are replicated as a set of single-row updates by the Message Agent, so this imposes no constraints on client applications.

Conflict resolution trigger syntax

The syntax for a RESOLVE UPDATE trigger is as follows:

```
CREATE TRIGGER trigger-name
RESOLVE UPDATE
OF column-name ON table-name
[ REFERENCING [ OLD AS old_val ]
  [ NEW AS new_val ]
```

```
[ REMOTE AS remote_val ] ]  
FOR EACH ROW  
BEGIN  
...  
END
```

RESOLVE UPDATE triggers fire before each row is updated. The REFERENCING clause allows access to the values in the row of the table to be updated (OLD), to the values the row is to be updated to (NEW) and to the rows that should be present according to the VERIFY clause (REMOTE). Only columns present in the VERIFY clause can be referenced in the REMOTE AS clause; other columns produce a "column not found" error.

Using the VERIFY_ALL_COLUMNS option

The database option `verify_all_columns` is Off by default. If it is set to On, all columns are verified on replicated updates, and a RESOLVE UPDATE trigger is fired whenever any column is different. If it is set to Off, only those columns that are updated are checked.

Setting this option to On makes messages bigger, because more information is sent for each UPDATE.

If this option is set at the consolidated database before remote databases are extracted, it will be set at the remote databases also.

You can set the `verify_all_columns` option either for the PUBLIC group or just for the user contained in the Message Agent connection string.

Using the CURRENT_REMOTE_USER special constant

The CURRENT_REMOTE_USER special constant holds the user ID of the remote user sending the message. This can be used in RESOLVE UPDATE triggers that place reports of conflicts into a table, to identify the user producing a conflict.

Conflict resolution examples

This section describes some ways of using RESOLVE UPDATE triggers to handle conflicts.

Resolving date conflicts

Suppose a table in a contact management system has a column holding the most recent contact with each customer.

One representative talks with a customer on a Friday, but does not upload his changes to the consolidated database until the next Monday. Meanwhile, a second representative meets the customer on the Saturday, and updates the changes that evening.

There is no conflict when the Saturday UPDATE is replicated to the consolidated database, but when the Monday UPDATE arrives it finds the row already changed.

By default, the Monday UPDATE would proceed, leaving the column with the incorrect information that the most recent contact occurred on Friday.

Update conflicts on this column should be resolved by inserting the most recent date in the row.

Implementing the solution

The following RESOLVE UPDATE trigger chooses the most recent of the two new values and enters it in the database.

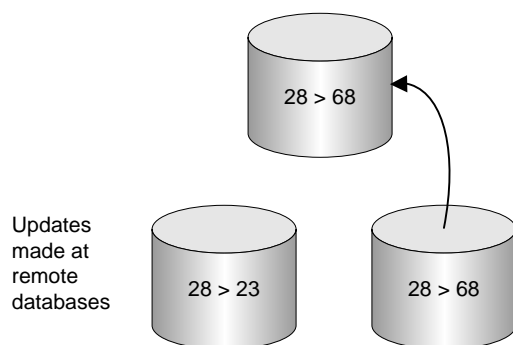
```
CREATE TRIGGER contact_date RESOLVE UPDATE
ON Contacts
REFERENCING OLD AS old_name
NEW AS new_name
FOR EACH ROW
BEGIN
    IF new_name.contact_date <
        old_name.contact_date THEN
        SET new_name.contact_date
            = old_name.contact_date
    END IF
END
```

If the value being updated is later than the value that would replace it, the new value is reset to leave the entry unchanged.

Resolving inventory conflicts

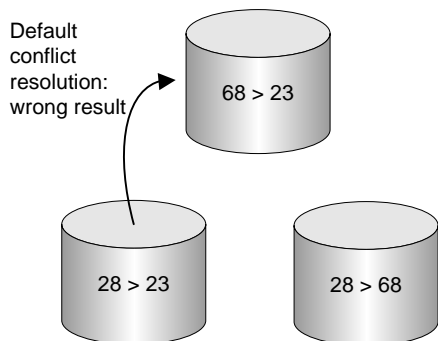
Consider a warehouse system for a manufacturer of sporting goods. There is a table of product information, with a **Quantity** column holding the number of each product left in stock. An update to this column will typically deplete the quantity in stock or, if a new shipment is brought in, add to it.

A sales representative at a remote database enters an order, depleting the stock of small tank top tee shirts by five, from 28 to 23, and enters this in on her database. Meanwhile, before this update is replicated to the consolidated database, a new shipment of tee shirts comes in, and the warehouse enters the shipment, adding 40 to the **Quantity** column to make it 68.

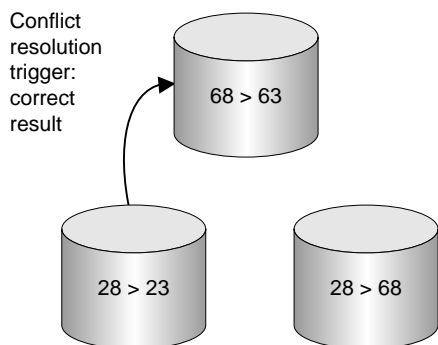


The warehouse entry gets added to the database: the **Quantity** column now shows there are 68 small tank-top tee shirts in stock. When the update from the sales representative arrives, it causes a conflict—SQL Anywhere detects that the update is from 28 to 23, but that the current value of the column is 68.

By default, the most recent UPDATE succeeds, and the inventory level is set to the incorrect value of 23.



In this case the conflict should be resolved by summing the changes to the inventory column to produce the final result, so that a final value of 63 is placed into the database.



Implementing the solution

A suitable RESOLVE UPDATE trigger for this situation would add the increments from the two updates. For example,

```
CREATE TRIGGER resolve_quantity
RESOLVE UPDATE OF Quantity
ON "DBA".Products
REFERENCING OLD AS old_name
NEW AS new_name
REMOTE AS remote_name
FOR EACH ROW
BEGIN
    SET new_name.Quantity =      new_name.Quantity
                                + old_name.Quantity
                                - remote_name.Quantity
END
```

This trigger adds the difference between the old value in the consolidated database (68) and the old value in the remote database when the original UPDATE was executed (28) to the new value being sent, before the UPDATE is implemented. Thus, **new_name.Quantity** becomes 63 (= 23 + 68 - 28), and this value is entered into the **Quantity** column.

Consistency is maintained at the remote database as follows:

1. The original remote UPDATE changed the value from 28 to 23.

2. The warehouse's entry is replicated to the remote database, but fails as the old value is not what was expected.
3. The changes made by the RESOLVE UPDATE trigger are replicated to the remote database.

Reporting conflicts

In some cases, you may not want to alter the default way in which SQL Remote resolves conflicts; you may just want to report the conflicts by storing them in a table. In this way, you can look at the conflict table to see what, if any, conflicts have occurred, and if necessary take action to resolve the conflicts.

Designing to avoid referential integrity errors

The tables in a relational database are related through foreign key references. The referential integrity constraints applied as a consequence of these references ensure that the database remains consistent. If you wish to replicate only a part of a database, there are potential problems with the referential integrity of the replicated database.

By paying attention to referential integrity issues while designing publications you can avoid these problems. This section describes some of the more common integrity problems and suggests ways to avoid them.

Unreplicated referenced table errors

The **sales** publication described in [“Publishing whole tables” on page 31](#) includes the **SalesOrders** table:

```
CREATE PUBLICATION PubSales (  
    TABLE Customers,  
    TABLE SalesOrders,  
    TABLE SalesOrderItems,  
    TABLE Products  
)
```

The **SalesOrders** table has a foreign key to the **Employees** table. The ID of the sales rep is a foreign key in the **SalesOrders** table referencing the primary key of the **Employees** table. However, the **Employees** table is not included in the publication.

If the publication is created in this manner, new sales orders would fail to replicate unless the remote database has the foreign key reference removed from the **SalesOrders** table.

If you use the extraction utility to create the remote databases, the foreign key reference is automatically excluded from the remote database, and this problem is avoided. However, there is no constraint in the database to prevent an invalid value from being inserted into the **SalesRepresentative** column of the **SalesOrders** table, and if this happens the INSERT will fail at the consolidated database. To avoid this problem, you can include the **Employees** table (or at least its primary key) in the publication.

Designing triggers to avoid errors

Actions performed by triggers are not replicated: triggers that exist at one database in a SQL Remote setup are assumed by the replication procedure to exist at other databases in the setup. When an action that fires

a trigger at the consolidated database is replicated at the replicate site, the trigger is automatically fired. By default, the database extraction utility extracts the trigger definitions, so that they are in place at the remote database also.

If a publication includes only a subset of a database, a trigger at the consolidated database may refer to tables or rows that are present at the consolidated database, but not at the remote databases. You can design your triggers to avoid such errors by making actions of the trigger conditional using an IF statement. The following list suggests some ways in which triggers can be designed to work on consolidated and remote databases.

- ◆ Have actions of the trigger be conditional on the value of CURRENT PUBLISHER. In this case, the trigger would not execute certain actions at the remote database.
- ◆ Have actions of the trigger be conditional on the **object_id** function not returning NULL. The **object_id** function takes a table or other object as argument, and returns the ID number of that object or NULL if the object does not exist.
- ◆ Have actions of the trigger be conditional on a SELECT statement which determines if rows exist.

The RESOLVE UPDATE trigger is a special trigger type for the resolution of UPDATE conflicts, and is discussed in the section [“Conflict resolution examples” on page 58](#). The actions of RESOLVE UPDATE triggers are replicated to remote databases, including the database that caused the conflict.

Ensuring unique primary keys

Primary key values must be unique. When all users are connected to the same database, there is no problem keeping unique values. If a user tries to re-use a value, the INSERT statement fails.

The situation is different in a replication system because users are connected to many databases. A potential problem arises when two users, connected to different databases, insert a row using the same primary key value. Each of their statements succeeds because the value is unique in each database.

However, problems arise in a replication system when two users, connected to separate databases, INSERT a row using the same primary key value. The second INSERT to reach a given database in the replication system fails. As SQL Remote is a replication system for occasionally connected users, there can be no locking mechanism across all databases in the installation. It is necessary to design your SQL Remote installation so that primary key duplication errors do not occur.

For primary key errors to be designed out of SQL Remote installations, the primary keys of tables that may be modified at more than one site must be guaranteed unique. There are several ways of achieving this goal. This chapter describes two general, economical, and reliable methods.

1. Using the default global autoincrement feature of SQL Anywhere.
2. Using the primary key pools to maintain a list of unused, unique primary key values at each site.

You can use these techniques either separately or together to avoid duplicate values.

Using global autoincrement default column values

In SQL Anywhere, you can set the default column value to be GLOBAL AUTOINCREMENT. You can use this default for any column in which you want to maintain unique values, but it is particularly useful for primary keys. This feature is intended to simplify the task of generating unique values in setups where data is being replicated among multiple databases, typically by MobiLink synchronization.

When you specify default global autoincrement, the domain of values for that column is partitioned. Each partition contains the same number of values. For example, if you set the partition size for an integer column in a database to 1000, one partition extends from 1001 to 2000, the next from 2001 to 3000, and so on.

You assign each copy of the database a unique global database identification number. SQL Anywhere supplies default values in a database only from the partition uniquely identified by that database's number. For example, if you assigned the database in the above example the identity number 10, the default values in that database would be chosen in the range 10001–11000. Another copy of the database, assigned the identification number 11, would supply default value for the same column in the range 11001–12000.

Declaring default global autoincrement

You can set default values in your database by selecting the column properties in Sybase Central, or by including the DEFAULT GLOBAL AUTOINCREMENT phrase in a TABLE or ALTER TABLE statement.

Optionally, the partition size can be specified in parentheses immediately following the `AUTOINCREMENT` keyword. The partition size may be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.

For columns of type `INT` or `UNSIGNED INT`, the default partition size is $2^{16} = 65536$; for columns of other types the default partition size is $2^{32} = 4294967296$. Since these defaults may be inappropriate, especially if our column is not of type `INT` or `BIGINT`, it is best to specify the partition size explicitly.

For example, the following statement creates a simple table with two columns: an integer that holds a customer identification number and a character string that holds the customer's name.

```
CREATE TABLE Customers (  
  ID INT DEFAULT GLOBAL AUTOINCREMENT (5000),  
  name VARCHAR(128) NOT NULL,  
  PRIMARY KEY (ID)  
)
```

In the above example, the chosen partition size is 5000.

☞ For more information on `GLOBAL AUTOINCREMENT`, see [“CREATE TABLE statement” \[SQL Anywhere Server - SQL Reference\]](#).

Setting the `global_database_id` value

When deploying an application, you must assign a different identification number to each database. You can accomplish the task of creating and distributing the identification numbers by a variety of means. One method is to place the values in a table and download the correct row to each database based on some other unique property, such as user name.

◆ To set the global database identification number

- You set the identification number of a database by setting the value of the public option `global_database_id`. The identification number must be a non-negative integer.

For example, the following statement sets the database identification number to 20.

```
SET OPTION PUBLIC.global_database_id = 20
```

If the partition size for a particular column is 5000, default values for this database are selected from the range 100001–105000.

Setting unique database identification numbers when extracting databases

If you use the extraction utility to create your remote databases, you can write a stored procedure to automate the task. If you create a stored procedure named `sp_hook_dbxtract_begin`, it is called automatically by the extraction utility. Before the procedure is called, the extraction utility creates a temporary table named `#hook_dict`, with the following contents:

name	value
extracted_db_global_id	user ID being extracted

If you write your `sp_hook_dbxtract_begin` procedure to modify the value column of the row, that value is used as the `global_database_id` option of the extracted database, and marks the beginning of the range of primary key values for GLOBAL DEFAULT AUTOINCREMENT values.

Example

Consider extracting a database for remote user **user2** with a **user_id** of 101. If you do not define an `sp_hook_dbxtract_begin` procedure, the extracted database will have **global_database_id** set to **101**.

If you define a `sp_hook_dbxtract_begin` procedure, but it does not modify any rows in the `#hook_dict` then the option will still be set to **101**.

If you set up the database as follows:

```
set option "PUBLIC"."global_database_id" = '1';
create table extract_id ( next_id integer not null ) ;
insert into extract_id values( 1 );
create procedure sp_hook_dbxtract_begin
as
  declare @next_id integer
  update extract_id set next_id = next_id + 1000
  select @next_id = (next_id )
  from extract_id
  commit
  update #hook_dict
  set value = @next_id
  where name = 'extracted_db_global_id'
```

Then each extracted or re-extracted database will get a different **global_database_id**. The first starts at 1001, the next at 2001, and so on.

To assist in debugging procedure hooks, `dbxtract` outputs the following when it is set to operate in verbose mode:

- ◆ the procedure hooks found
- ◆ the contents of `#hook_dict` before the procedure hook is called
- ◆ the contents of `#hook_dict` after the procedure hook is called.

How default values are chosen

The public option **global_database_id** in each database must be set to a unique, non-negative integer. The range of default values for a particular database is $pn + 1$ to $p(n + 1)$, where p is the partition size and n is the value of the public option **global_database_id**. For example, if the partition size is 1000 and **global_database_id** is set to 3, then the range is from 3001 to 4000.

If **global_database_id** is set to a non-negative integer, SQL Anywhere chooses default values by applying the following rules:

- ◆ If the column contains no values in the current partition, the first default value is $pn + 1$.
- ◆ If the column contains values in the current partition, but all are less than $p(n + 1)$, the next default value will be one greater than the previous maximum value in this range.
- ◆ Default column values are not affected by values in the column outside of the current partition; that is, by numbers less than $pn + 1$ or greater than $p(n + 1)$. Such values may be present if they have been replicated from another database via MobiLink synchronization.

If the public option **global_database_id** is set to the default value of 2147483647, a null value is inserted into the column. Should null values not be permitted, the attempt to insert the row causes an error. This situation arises, for example, if the column is contained in the table's primary key.

Because the public option **global_database_id** cannot be set to negative values, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.

Null default values are also generated when the supply of values within the partition has been exhausted. In this case, a new value of **global_database_id** should be assigned to the database to allow default values to be chosen from another partition. Attempting to insert the null value causes an error if the column does not permit nulls. To detect that the supply of unused values is low and handle this condition, create an event of type **GlobalAutoincrement**.

Should the values in a particular partition become exhausted, you can assign a new database ID to that database. You can assign new database ID numbers in any convenient manner. However, one possible technique is to maintain a pool of unused database ID values. This pool is maintained in the same manner as a pool of primary keys.

You can set an event handler to automatically notify the database administrator (or carry out some other action) when the partition is nearly exhausted. For more information, see [“Defining trigger conditions for events” \[SQL Anywhere Server - Database Administration\]](#).

☞ For more information, see [“global_database_id option \[database\]” \[SQL Anywhere Server - Database Administration\]](#).

☞ For further information on pools, see [“Using primary key pools” on page 66](#).

Using primary key pools

The **primary key pool** is a table that holds a set of primary key values for each database in the SQL Remote installation. Each remote user receives their own set of primary key values. When a remote user inserts a new row into a table, they use a stored procedure to select a valid primary key from the pool. The pool is maintained by periodically running a procedure at the consolidated database that replenishes the supply.

The method is described using a simple example database consisting of sales representatives and their customers. The tables are much simpler than you would use in a real database; this allows us to focus just on those issues important for replication.

The primary key pool technique requires the following components:

- ◆ **Key pool table** A table to hold valid primary key values for each database in the installation.
- ◆ **Replenishment procedure** A stored procedure keeps the key pool table filled.
- ◆ **Sharing of key pools** Each database in the installation must subscribe to its own set of valid values from the key pool table.
- ◆ **Data entry procedures** New rows are entered using a stored procedure that picks the next valid primary key value from the pool and delete that value from the key pool.

The primary key pool table

The pool of primary keys is held in a separate table. The following CREATE TABLE statement creates a primary key pool table:

```
CREATE TABLE KeyPool (
    table_name VARCHAR(40) NOT NULL,
    value INTEGER NOT NULL,
    location CHAR(12) NOT NULL,
    PRIMARY KEY (table_name, value),
);
```

The columns of this table have the following meanings:

Column	Description
table_name	Holds the names of tables for which primary key pools must be maintained. In our simple example, if new sales representatives were to be added only at the consolidated database, only the Customers table needs a primary key pool and this column is redundant. It is included to show a general solution.
value	Holds a list of primary key values. Each value is unique for each table listed in table_name .
location	An identifier for the recipient. In some setups, this could be the same as the rep_key value of the SalesReps table. In other setups, there will be users other than sales representatives and the two identifiers should be distinct.

For performance reasons, you may wish to create an index on the table:

```
CREATE INDEX KeyPoolLocation
ON KeyPool (table_name, location, value);
```

Replicating the primary key pool

You can either incorporate the key pool into an existing publication or share it as a separate publication. In this example, you create a separate publication for the primary key pool.

◆ To replicate the primary key pool (SQL)

1. Create a publication for the primary key pool data.

```
CREATE PUBLICATION KeyPoolData (
    TABLE KeyPool SUBSCRIBE BY location
);
```

2. Create subscriptions for each remote database to the KeyPoolData publication.

```
CREATE SUBSCRIPTION
TO KeyPoolData( 'user1' )
FOR user1;
CREATE SUBSCRIPTION
TO KeyPoolData( 'user2' )
FOR user2;
...
```

The subscription argument is the location identifier.

In some circumstances it makes sense to add the KeyPool table to an existing publication and use the same argument to subscribe to each publication. This example keeps the location and rep_key values distinct to provide a more general solution.

See also

- ◆ “CREATE PUBLICATION statement [MobiLink] [SQL Remote]” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “CREATE SUBSCRIPTION statement [SQL Remote]” [[SQL Anywhere Server - SQL Reference](#)]

Replenishing the key pool

Every time a user adds a new customer, their pool of available primary keys is depleted by one. The primary key pool table needs to be periodically replenished at the consolidated database using a procedure such as the following:

```
CREATE PROCEDURE ReplenishPool()
BEGIN
    FOR EachTable AS TableCursor
    CURSOR FOR
        SELECT table_name
        AS CurrTable, max(value) as MaxValue
        FROM KeyPool
        GROUP BY table_name
    DO
        FOR EachRep AS RepCursor
        CURSOR FOR
            SELECT location
            AS CurrRep, count(*) as NumValues
            FROM KeyPool
            WHERE table_name = CurrTable
            GROUP BY location
        DO
            // make sure there are 100 values.
            // Fit the top-up value to your
            // requirements
            WHILE NumValues < 100 LOOP
                SET MaxValue = MaxValue + 1;
                SET NumValues = NumValues + 1;
                INSERT INTO KeyPool
                (table_name, location, value)
                VALUES
                (CurrTable, CurrRep, MaxValue);
```

```

        END LOOP;
    END FOR;
END FOR;
END;
```

This procedure fills the pool for each user up to 100 values. The value you need depends on how often users are inserting rows into the tables in the database.

The **ReplenishPool** procedure must be run periodically at the consolidated database to refill the pool of primary key values in the **KeyPool** table.

The **ReplenishPool** procedure requires at least one primary key value to exist for each subscriber, so that it can find the maximum value and add one to generate the next set. To initially fill the pool you can insert a single value for each user, and then call **ReplenishPool** to fill up the rest. The following example illustrates this for three remote users and a single consolidated user named **Office**:

```

INSERT INTO KeyPool VALUES( 'Customers', 40, 'user1' );
INSERT INTO KeyPool VALUES( 'Customers', 41, 'user2' );
INSERT INTO KeyPool VALUES( 'Customers', 42, 'user3' );
INSERT INTO KeyPool VALUES( 'Customers', 43, 'Office' );
CALL ReplenishPool();
```

Cannot use a trigger to replenish the key pool

You cannot use a trigger to replenish the key pool, as trigger actions are not replicated.

Using primary keys from the key pool

When a sales representative wants to add a new customer to the Customers table, the primary key value to be inserted is obtained using a stored procedure. This example shows a stored procedure to supply the primary key value, and also illustrates a stored procedure to carry out the INSERT.

The procedure takes advantage of the fact that the Sales Rep identifier is the CURRENT PUBLISHER of the remote database.

- ◆ **NewKey procedure** The **NewKey** procedure supplies an integer value from the key pool and deletes the value from the pool.

```

CREATE PROCEDURE NewKey(
    IN @table_name VARCHAR(40),
    OUT @value INTEGER )
BEGIN
    DECLARE NumValues INTEGER;

    SELECT count(*), min(value)
    INTO NumValues, @value
    FROM KeyPool
    WHERE table_name = @table_name
    AND location = CURRENT PUBLISHER;
    IF NumValues > 1 THEN
        DELETE FROM KeyPool
        WHERE table_name = @table_name
        AND value = @value;
    ELSE
        // Never take the last value, because
        // ReplenishPool will not work.
```

```
        // The key pool should be kept large enough
        // that this never happens.
        SET @value = NULL;
    END IF;
END;
```

- ◆ **NewCustomers procedure** The **NewCustomers** procedure inserts a new customer into the table, using the value obtained by **NewKey** to construct the primary key.

```
CREATE PROCEDURE NewCustomers(
    IN customer_name CHAR( 40 ) )
BEGIN
    DECLARE new_cust_key INTEGER ;
    CALL NewKey( 'Customers', new_cust_key );
    INSERT
    INTO Customers (
        cust_key,
        name,
        location
    )
    VALUES (
        'Customers ' ||
        CONVERT (CHAR(3), new_cust_key),
        customer_name,
        CURRENT PUBLISHER
    );
);
END
```

You may want to enhance this procedure by testing the **new_cust_key** value obtained from **NewKey** to check that it is not NULL, and preventing the insert if it is NULL.

Creating subscriptions

To subscribe to a publication, each subscriber must be granted REMOTE permissions and a subscription must also be created for that user. The details of the subscription are different depending on whether or not the publication uses a subscription expression.

Working with subscriptions in Sybase Central

◆ To create and manage subscriptions in Sybase Central

1. In the left pane, open the Publications folder.
2. Select the desired publication. You can perform the following tasks in Sybase Central:
3. In the right pane, click the SQL Remote Subscriptions tab. You can configure the appropriate settings as follows:
 - ◆ To subscribe a remote user to the publication, from the File menu choose New ► SQL Remote Subscription and follow the instructions in the SQL Remote Subscription Creation wizard.
 - ◆ To unsubscribe a remote user, right-click the user in the Subscribers list and choose Delete from the popup menu.
 - ◆ To manually start, stop, or synchronize subscriptions, select the user in the Subscribers list and choose Properties from the popup menu.

Click the Advanced tab. On this tab, click Start Now to start subscriptions, Stop Now to stop subscriptions, or Synchronize Now to synchronize subscriptions.

The subscriptions are affected as soon as you click the button. Subsequently clicking Cancel on the property sheet does *not* cancel your start/stop/synchronize action.

Subscriptions with no subscription expression

To subscribe a user to a publication, if that publication has no subscription expression, you need the following information:

- ◆ **User ID** The user who is being subscribed to the publication. This user must have been granted remote permissions.
- ◆ **Publication name** The name of the publication to which the user is being subscribed.

The following statement creates a subscription for a user ID **SamS** to the **PubOrdersSamuelSinger** publication, which was created using a WHERE clause:

```
CREATE SUBSCRIPTION  
TO PubOrdersSamuelSinger  
FOR SamS
```

Subscriptions with a subscription expression

To subscribe a user to a publication, if that publication does have a subscription expression, you need the following information:

- ◆ **User ID** The user who is being subscribed to the publication. This user must have been granted remote permissions.
- ◆ **Publication name** The name of the publication to which the user is being subscribed.
- ◆ **Subscription value** The value that is to be tested against the subscription expression of the publication. For example, if a publication has the name of a column containing an employee ID as a subscription expression, the value of the employee ID of the subscribing user must be provided in the subscription. The subscription value is always a string.

The following statement creates a subscription for Samuel Singer (user ID **SamS**, employee ID 856) to the PubOrders publication, defined with a subscription expression **SalesRepresentative**, requesting the rows for Samuel Singer's own sales:

```
CREATE SUBSCRIPTION
TO PubOrders ( '856' )
FOR SamS
```

Starting a subscription

In order to receive and apply updates properly, each subscriber needs to have an initial copy of the data. The synchronization process is discussed in [“Synchronizing databases” on page 79](#).

☞ For more information, see [“CREATE SUBSCRIPTION statement \[SQL Remote\]” \[SQL Anywhere Server - SQL Reference\]](#).

Part III. SQL Remote Administration

This part describes deployment and administration issues for SQL Remote.

CHAPTER 5

Deploying and Synchronizing Databases

Contents

Deployment overview 76

Test before deployment 77

Synchronizing databases 79

Using the extraction utility 81

Synchronizing data over a message system 86

About this chapter

This chapter describes the steps you need to take to deploy and synchronize a SQL Remote replication installation.

Deployment overview

When you have completed the design phase of a SQL Remote system, the next step is to create and deploy the remote databases and applications.

Deployment tasks

In some cases, deployment is a major undertaking. For example, if you have a large number of remote users in a sales force automation system, deployment involves the following steps:

1. Building a SQL Anywhere database for each remote user, with their own initial copy of the data.
2. Installing the database, together with the SQL Anywhere database server, the SQL Remote Message Agent, and client application, on each user's machine.
3. Ensuring that the system is properly configured, with correct user names, Message Agent connection strings, permissions, and so on.

In the case of large-scale deployments, remote sites are most commonly SQL Anywhere databases, and this chapter focuses on this case.

Topics covered

This chapter covers the following topics:

- ◆ **Creating remote databases** Before you can deploy a SQL Remote system, you must create a remote database for each remote site.

Most of the description focuses on creating remote SQL Anywhere databases.

- ◆ **Synchronizing data** Synchronization of a database is the setting up of the initial copy of data in the remote database.

Test before deployment

Thorough testing of your SQL Remote system should be carried out before deployment, especially if you have a large number of remote sites.

When you are in the design and setup phase, you can alter many facets of the SQL Remote setup. Altering publications, message types, writing triggers to resolve update conflicts are all easy to do.

Once you have deployed a SQL Remote application, the situation is different. A SQL Remote setup can be seen as a single **dispersed database**, spread out over many sites, maintaining a loose form of consistency. The data may never be in exactly the same state in all databases in the setup at once, but all data changes are replicated as complete transactions around the system over time. Consistency is built in to a SQL Remote setup through careful publication design, and through the reconciliation of UPDATE conflicts as they occur.

Upgrading and resynchronization

Once a SQL Remote setup is deployed and is running, it is not easy to tinker with. An upgrade to a SQL Remote installation needs to be carried out with the same care as an initial deployment. This applies also to upgrading maintenance releases of the SQL Anywhere database software. Any such software upgrade needs to be tested for compatibility before deployment.

Making changes to a database schema at one database within the system can cause failures because of incompatible database objects. The passthrough mode does allow schema changes to be sent to some or all databases in a SQL Remote setup, but must still be used with care and planning.

The loose consistency in the dispersed database means that updates are always in progress: you cannot generally stop changes being made to all databases, make some changes to the database schema, and restart.

Without careful planning, changes to a database schema will produce errors throughout the installation, and will require all subscriptions to be stopped and resynchronized. Resynchronization involves loading new copies of the data in each remote database, and for more than a few subscribers is a time-consuming process involving work interruptions and possible loss of data.

Changes to avoid on a running system

The following are examples of changes that should not be made to a deployed and running SQL Remote setup. From the list, you will see that there is a class of changes that are **permissive**, and these are generally permissible, while other changes are **restrictive**, and must be avoided.

The following changes must be avoided, except under the conditions stated:

- ◆ Change the publisher for the consolidated database.
- ◆ Make restrictive changes to tables, such as dropping a column or altering a column to not allow NULL values. Changes that include the column or including NULL entries may already be being sent in messages around the SQL Remote setup, and will fail.
- ◆ Alter a publication. Publication definitions must be maintained at both local and remote sites, and changes that rely on the old publication definition may already be being sent in messages around the SQL Remote setup.

You can make permissive changes, such as adding a new table or column, as long as you use passthrough to ensure that the new table or column exists in the remote database and in the publication at the remote database.

- ◆ Drop a subscription. This can be done only if you use passthrough deletes to remove the data at the remote site.
- ◆ Unload and reload a SQL Anywhere database.

If a SQL Anywhere database is participating in replication, it cannot be unloaded and reloaded without re-synchronizing the database. Replication is based on the transaction log, and when a database is unloaded and reloaded, the old transaction log is no longer available. For this reason, good backup practices are especially important when participating in replication.

Synchronizing databases

What is synchronization?

SQL Remote replication is carried out using the information in the transaction log, but there are two circumstances where SQL Remote deletes all existing rows from those tables of a remote database that form part of a publication, and copies the publication's entire contents from the consolidated database to the remote site. This process is called **synchronization**.

When to synchronize

Synchronization is used under the following circumstances:

- ◆ When a subscription is created at a consolidated database a synchronization is carried out, so that the remote database starts off with a database in the same state as the consolidated database.
- ◆ If a remote database gets corrupt or gets out of step with the consolidated database, and cannot be repaired using SQL passthrough mode, synchronization forces the remote site database back in step with the consolidated site.

How to synchronize

Synchronizing a remote database can be done in the following ways:

- ◆ **Use the database extraction utility** This utility creates a schema for a remote SQL Anywhere database, and synchronizes the remote database. This is generally the recommended procedure.
- ◆ **Manual synchronization** Synchronize the remote database manually by loading from files, using the PowerBuilder pipeline, or some other tool.
- ◆ **Synchronize over the message system** Synchronize the remote database via the message system using the SYNCHRONIZE SUBSCRIPTION statement .

Caution

Do not execute SYNCHRONIZE SUBSCRIPTION at a remote database.

Mixed operating systems and database extraction

In many installations, the consolidated server will be running on a different operating system than the remote databases.

SQL Anywhere databases can be copied from one file or operating system to another. This allows you flexibility in how you carry out your initial synchronization of databases.

Notes on synchronization and extraction

- ◆ Extracting large numbers of subscriptions, or synchronizing subscriptions to large, frequently-used tables, can slow down database access for other users. You may want to extract such subscriptions when

the database is not in heavy use. This happens automatically if you use a SEND AT clause with a quiet time specified.

- ◆ Synchronization applies to an entire subscription. There is currently no straightforward way of synchronizing a single table.

Using the extraction utility

The extraction utility is an aid to creating remote SQL Anywhere databases.

Running the extraction utility

The extraction utility can be accessed in the following ways:

- ◆ From Sybase Central, if your consolidated database is SQL Anywhere.
- ◆ As a command line utility. This is the *dbxtract* utility.

Caution

Do not run the Message Agent while running the extraction utility. The results are unpredictable.

Creating a database from the reload files

The command line utility unloads a database schema and data suitable for building a remote SQL Anywhere database for a named subscriber. It produces a SQL command file with default name *reload.sql* and a set of data files. You can use these files to create a remote SQL Anywhere database.

Editing of reload.sql may be needed

The database extraction utility is intended to assist in preparing remote databases, but is not intended as a black box solution for all circumstances. You should edit the *reload.sql* command file as needed when creating remote databases.

◆ To create a remote database from the reload file

1. Create a SQL Anywhere database using one of the following:
 - ◆ the Sybase Central Create Database wizard (from the Tools menu, choose SQL Anywhere 10 ► Create Database)
 - ◆ the *dbinit* utility
2. Connect to the database from the Interactive SQL utility, and run the *reload.sql* command file. The following statement entered in the SQL Statements pane runs the *reload.sql* command file:

```
read path\reload.sql
```

where *path* is the path of the reload command file.

When used from Sybase Central, the extraction utility carries out the database unloading task, in the same way that *dbxtract* does, and then takes the additional step of creating the new database.

The extraction utility does not use a message system. The reload file (*dbxtract*) or database (from Sybase Central) is created in a directory accessible from the current machine. Synchronizing many subscriptions

over a message link can produce heavy message traffic and, if the message system is not completely reliable, it may take some time for all the messages to be properly received at the remote sites.

Before extracting a database

You must complete the following tasks before using the extraction utility at a consolidated database.

- ◆ Create message types for replication.
- ◆ Add a publisher user ID to the database.
- ◆ Add remote users to the database.
- ◆ Add the publication to the database.
- ◆ Created a subscription for the remote users.
- ◆ If you need to specify message link parameters, you must have set them. See [“Setting message type control parameters” on page 99](#).

When you use the extraction utility to create a remote database, the user for which you are creating the database receives the same permissions they have in the consolidated database. Further, if the user is a member of any groups on the consolidated database, those group IDs are created in the remote database with the permissions they have in the consolidated database.

Using the extraction utility from Sybase Central

This section describes how to extract a database for a remote user from the current consolidated database. This section applies only to SQL Anywhere consolidated databases.

When you complete the Extract Database wizard, it does the following on your machine:

- ◆ Creates the remote database
- ◆ Extracts (unloads) the relevant structures and/or data from the consolidated database to files
- ◆ Loads those files into the newly created remote database

◆ To extract a database for a remote user (Sybase Central)

1. From the Tools menu, choose SQL Anywhere 10 ► Extract Database.
2. Follow the instructions in the wizard.

Notes

- ◆ You can also access this wizard by clicking Tools ► SQL Anywhere 10 ► Extract Database.
- ◆ You can also invoke the extraction wizard for a particular database or for a particular remote user—Sybase Central automatically fills in the appropriate entries in the wizard.

- ◆ The extraction wizard always extracts (synchronizes) the remote database using the WITH SYNCHRONIZATION option. In those rare cases where you don't want to use this option, you must use the dbextract utility instead.
- ◆ Only tables for users selected in the Filter Objects by Owner dialog appear in the Extract Database wizard. If you want to view tables belonging to a particular database user, right-click the database you are unloading, choose Filter Objects by Owner from the popup menu, and then select the desired user in the resulting dialog.

See also

- ◆ For information about the extraction utility options, available as command line options or as choices presented by the extraction wizard, see [“Extraction utility” on page 157](#).

Designing an efficient extraction procedure

It is very inefficient to create a large number of remote databases by running the extraction utility for each one. You can make the process much more efficient. This section describes one way of making the process more efficient.

There are several potential causes of inefficiency in a large-scale extraction process:

- ◆ The extraction utility extracts one database at a time, including the schema and data for each user. Commonly, many users share a common schema, and only the data differs. The brute force method of running the extraction utility for each user repeats large amounts of work unnecessarily. Extracting schema and data separately can help with this problem.
- ◆ Running from Sybase Central, the extraction utility creates a new database for each user. If subscribers share a common schema, you could create a single database, with schema but no data, and copy the file.
- ◆ By default, the extraction utility runs at isolation level zero. If you are extracting a database from an active server, you should run it at isolation level 3 (see [“Extraction utility” on page 157](#)) to ensure that data in the extracted database is consistent with data on the server.

Running at isolation level 3 may hamper others' turnaround time on the server because of the large number of locks required. It is recommended that you run the extraction utility when the server is not busy, or run it against a copy of the database.

An efficient approach to extracting many databases

One approach that avoids these problems is as follows:

1. Make a copy of the consolidated database, and at the same time start the subscriptions from the live database. Messages will now start being sent to subscribers, even though they have no database and will not receive them yet.

To start several subscriptions within a single transaction, use the REMOTE RESET statement.

2. Extract the remote databases from the copy of the database. As the database is a copy, there are no locking and concurrency problems. For a large number of remote databases, this process may take several days.

3. As each remote database is created, it is out of date, but its user can receive and apply messages that have been being sent from the live consolidated database, to bring themselves up to date.

This solution interferes with the production database only during the first step. The copy must be made at isolation level three if the database is in use, and uses large numbers of locks. Also, the subscriptions must be started at the same time that the copy is made. Any operations that take place between the copy and the starting of the subscriptions would be lost, and could lead to errors at remote databases.

Extracting groups

If the remote user is a group user ID, the extraction utility extracts all the user IDs of members of that group. You can use this feature to all multiple users on each remote database, using different user IDs, without requiring a custom extraction process.

When a database is extracted for a user, all message link parameters for that user and the groups of which the user is a member are extracted.

Limits to using the extraction utility

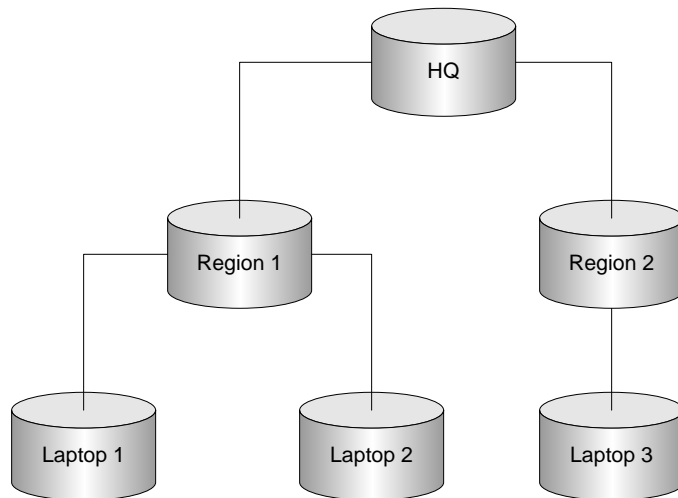
While the extraction utility is the recommended way of creating and synchronizing remote databases from a consolidated databases, there are some circumstances where it cannot be used, and you must synchronize remote databases manually. This section describes some of those cases.

- ◆ **Additional tables at the remote database** Remote databases can have tables not present at their consolidated database as long as these tables do not take part in replication. Of course, the extraction utility cannot extract such tables from a consolidated database.
- ◆ **Extracting procedures and views** By default, the extraction utility extracts all stored procedures and views from the database. While some of these views and procedures are likely to be required at the remote site, others may not be required—they may refer only to parts of the database that are not included in the remote site.

After running the extraction utility, you should edit the reload script and remove unnecessary views and procedures.

- ◆ **Using the extraction utility in multi-tiered setups** To understand the role of the extraction utility in multi-tiered arrangements, consider a three-tiered SQL Remote setup.

This setup is illustrated in the following diagram.



From the consolidated database at the top level, you can use the extraction utility to create the second-level databases. You can then add remote users to these second-level databases, and use the extraction utility from each second-level database to create the remote databases. However, if you have to re-extract the second-level databases from the top-level consolidated database, you will delete the remote users that were created, along with their subscriptions and permissions, and will have to rebuild those users. The exception is if you resynchronize data only, in which case you can use the extraction utility to replace the data in the database, without replacing the schema.

Synchronizing data over a message system

Creating subscriptions

Creating a subscription defines the data to be received. It does not synchronize a subscription (provide an initial copy of the data) or start (exchange messages) a subscription.

Synchronizing subscriptions

Synchronizing a subscription causes the Message Agent to send a copy of all rows in the subscription to the subscriber. It assumes that an appropriate database schema is in place. Subscriptions are synchronized using the `SYNCHRONIZE SUBSCRIPTION` statement.

When synchronization messages are received at a subscriber database, the Message Agent replaces the current contents of the database with the new copy. Any data at the subscriber that is part of the subscription, and which has not been replicated to the consolidated database, is lost. Once synchronization is complete, the subscription is started by the Message Agent using the `START SUBSCRIPTION` statement.

Large volume of messages may result

Synchronizing databases over a message system may lead to large volumes of messages. In many cases, it is preferable to use the extraction process to synchronize a database locally without placing this burden on the message system.

Synchronizing subscriptions during operation

If a remote database becomes out of step with the consolidated database, and cannot be brought back in step using the SQL passthrough capabilities of SQL Remote, synchronizing the subscription forces the remote database into step with the consolidated database by copying the rows of the subscription from the consolidated database over the contents at the remote database.

Data loss on synchronization

Any data in the remote database that is part of the subscription, but which has not been replicated to the consolidated database, is lost when the subscription is synchronized. You may want to unload or back up the remote database using Sybase Central or, for SQL Anywhere, the `dbunload` utility before synchronizing the database.

CHAPTER 6

SQL Remote Administration

Contents

Management overview	88
Managing SQL Remote permissions	89
Using message types	96
Running the Message Agent	108
Tuning Message Agent performance	112
Encoding and compressing messages	118
The message tracking system	120

About this chapter

This chapter describes general issues and principles for administering a running SQL Remote installation.

 For system-specific details, see [“Administering SQL Remote”](#) on page 123.

Management overview

This chapter describes administration issues for SQL Remote installations.

Administration of a deployed and running SQL Remote setup is carried out at a consolidated database.

- ◆ **Permissions** As a SQL Remote installation includes many different physical databases, a consistent scheme for users having permissions on remote and consolidated databases is necessary. A section of this chapter describes the considerations you need to make when assigning users permissions.
- ◆ **Configuring message systems** Each message system that is used in a SQL Remote installation has control parameters and other settings that must be set up. These settings are discussed in this chapter.
- ◆ **The Message Agent** The Message Agent is responsible for sending and receiving messages.
- ◆ **Message tracking** Administering a SQL Remote installation means managing large numbers of messages being handed back and forth among many databases. A section on the SQL Remote message tracking system is included to help you understand what the messages contain, when they are sent, how they are applied, and so on.
- ◆ **Log management** SQL Remote obtains the data to send from the transaction log. Consequently, proper management of the transaction log, and proper backup procedures, are essential for a smoothly running SQL Remote installation. While many details depend on the server you are running, the generic issues are discussed in this chapter.
- ◆ **Passthrough mode** This is a method for directly intervening at a remote site from a consolidated database. This method is discussed in this chapter.

Managing SQL Remote permissions

Users of a database involved in SQL Remote replication are identified by one of the following sets of permissions:

- ◆ **PUBLISH** A single user ID in a database is identified as the publisher for that database. All outgoing SQL Remote messages, including both publication updates and receipt confirmations, are identified by the publisher user ID. Every database in a SQL Remote setup must have a single publisher user ID, as every database in a SQL Remote setup sends messages.
- ◆ **REMOTE** All recipients of messages from the current database, or senders of messages to the current database, who are immediately lower on the SQL Remote hierarchy than the current database must be granted REMOTE permissions.
- ◆ **CONSOLIDATE** At most one user ID may be granted CONSOLIDATE permissions in a database. CONSOLIDATE permissions identifies a database immediately above the current database in a SQL Remote setup. Each database can have only one consolidated database directly above it.

Information about these permissions are held in the SQL Remote system tables, and are independent of other database permissions.

Granting and revoking PUBLISH permissions

When a database sends a message, a user ID representing that database is included with the message to identify its source to the recipient. This user ID is the **publisher** user ID of the database. A database can have only one publisher. You can find out who the publisher of a SQL Anywhere database is at any time in Sybase Central by opening the Users & Groups folder.

A publisher is required even for read-only remote databases within a replication system, as even these databases send confirmations to the consolidated database to maintain information about the status of the replication. The GRANT PUBLISH statement for remote SQL Anywhere databases is carried out automatically by the database extraction utility.

Granting and revoking PUBLISH permissions from Sybase Central

You can grant PUBLISH permissions on a SQL Anywhere database from Sybase Central. You must connect to the database as a user with full system or database administrator permissions.

◆ To create a new user as the publisher (Sybase Central)

1. In the left pane, select the Users & Groups folder.
2. From the File menu, choose New ► User.
The Create User wizard appears.
3. Follow the instructions in the wizard. Ensure that the user has a password and is granted Remote DBA authority; this enables the user ID to run the Message Agent.
4. Click Finish to create the user.

5. In the Users & Groups folder, right-click the user you just created and choose Change to Publisher from the popup menu.

◆ To make an existing user the publisher (Sybase Central)

- In the Users & Groups folder, right-click a user and choose Change to Publisher from the popup menu.

You can also revoke PUBLISH permissions from Sybase Central.

◆ To revoke PUBLISH permissions (Sybase Central)

- In the Users & Groups folder, right-click the user who has granted PUBLISH permissions and choose Revoke Publisher from the popup menu.

Granting and revoking PUBLISH permissions

For SQL Anywhere, PUBLISH permissions are granted using the GRANT PUBLISH statement:

```
GRANT PUBLISH TO userid ;
```

The *userid* is a user with CONNECT permissions on the current database. For example, the following statement grants PUBLISH permissions to user **S_Beaulieu**:

```
GRANT PUBLISH TO S_Beaulieu
```

The REVOKE PUBLISH statement revokes the PUBLISH permissions from the current publisher:

```
REVOKE PUBLISH FROM userid
```

Notes on PUBLISH permissions

- ◆ To see the publisher user ID outside Sybase Central, use the CURRENT PUBLISHER special constant. The following statement retrieves the **publisher** user ID:

```
SELECT CURRENT PUBLISHER
```

- ◆ If PUBLISH permissions is granted to a user ID with GROUP permissions, it is not inherited by members of the group.
- ◆ PUBLISH permissions carry no authority except to identify the publisher in outgoing messages.
- ◆ For messages sent from the current database to be received and processed by a recipient, the publisher user ID must have REMOTE or CONSOLIDATE permissions on the receiving database.
- ◆ The publisher user ID for a database cannot also have REMOTE or CONSOLIDATE permissions on that database. This would identify them as both the sender of outgoing messages and a recipient of such messages.
- ◆ Changing the user ID of a publisher at a remote database will cause serious problems for any subscriptions that database is involved in, including loss of information. You should not change a remote database publisher user ID unless you are prepared to resynchronize the remote user from scratch.
- ◆ Changing the user ID of a publisher at a consolidated database while a SQL Remote setup is operating will cause serious problems, including loss of information. You should not change the consolidated

database publisher user ID unless you are prepared to close down the SQL Remote setup and resynchronize all remote users.

Granting and revoking REMOTE and CONSOLIDATE permissions

REMOTE and CONSOLIDATE permissions are very similar. Each database receiving messages from the current database must have an associated user ID on the current database that is granted one of REMOTE or CONSOLIDATE permissions. This user ID represents the receiving database in the current database.

The consolidated user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. There can be only one consolidated user per remote database.

The remote user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user.

Databases directly below the current database on a SQL Remote hierarchy are granted REMOTE permissions, and the at most one database above the current database in the hierarchy is granted CONSOLIDATE permissions.

Setting REMOTE and CONSOLIDATE permissions

The GRANT REMOTE and GRANT CONSOLIDATE statements identify the message system and address to which replication messages must be sent.

CONSOLIDATE permissions must be granted even from read-only remote databases to the consolidated database, as receipt confirmations are sent back from the remote databases to the consolidated database. The GRANT CONSOLIDATE statement at remote SQL Anywhere databases is executed automatically by the database extraction utility.

Granting REMOTE permissions

Each remote database must be represented by a single user ID in the consolidated database. This user ID must be granted REMOTE permissions to identify their user ID and address as a subscriber to publications.

Granting REMOTE permissions accomplishes several tasks:

- ◆ It identifies a user ID as a remote user.
- ◆ It specifies a message type to use for exchanging messages with this user ID.
- ◆ It provides an address to where messages are to be sent.
- ◆ It indicates how often messages should be sent to the remote user.

Granting REMOTE permissions is also referred to as adding a remote user to the database.

Sybase Central example

You can add a remote user to a database using Sybase Central. Remote users and groups appear in two locations in Sybase Central: in the Users & Groups folder, and in the SQL Remote Users folder. This section applies only to SQL Anywhere databases.

By default, remote users are created with remote DBA authority. Since the message agent for access to the remote database requires this authority, you shouldn't revoke it.

You cannot create a new remote user until at least one message type is defined in the database.

While you can grant remote permissions to a group, those remote permissions do *not* automatically apply to users in the group (unlike table permissions, for example). To do this, you must explicitly grant remote permissions to each user in the group. Otherwise, remote groups behave exactly like remote users (and are categorized as remote users).

◆ To add a new user to the database as a remote user (Sybase Central)

1. In the left pane, select the SQL Remote Users folder.
2. From the File menu, choose New ► SQL Remote User.

The Create a New Remote User wizard appears.

3. Follow the instructions in the wizard.

◆ To make an existing user remote (Sybase Central)

1. Open the Users & Groups folder.
2. Right-click the user you want to make remote and choose Change to Remote User from the popup menu.
3. In the resulting dialog, select the message type from the list, enter an address, choose the frequency of sending messages, and click OK to make the user a remote user.

This user now appears in both the Users & Groups folder and the SQL Remote Users folder.

Example

The following statement grants remote permissions to user **S_Beaulieu**, with the following options:

- ◆ Use an SMTP email system
- ◆ Send messages to email address **s_beaulieu@acme.com**:
- ◆ Send message daily, at 10 p.m.

```
GRANT REMOTE TO S_Beaulieu
TYPE smtp
ADDRESS 's_beaulieu@acme.com'
SEND AT '22:00'
```

Selecting a send frequency

There are three alternatives for the setting the frequency with which messages are sent. The alternatives are:

- ◆ **SEND EVERY** A frequency can be specified in hours, minutes, and seconds in the format 'HH:MM:SS'.

When any user with SEND EVERY set is sent messages, all users with the same frequency are sent messages also. For example, all remote users who receive updates every twelve hours are sent updates at the same times, rather than being staggered. This reduces the number of times the SQL Anywhere transaction log has to be processed. You should use as few unique frequencies as possible.

- ◆ **SEND AT** A time of day, in hours and minutes.

Updates are started daily at the specified time. It is more efficient to use as few distinct times as possible than to stagger the sending times. Also, choosing times when the database is not busy minimizes interference with other users.

- ◆ **Default setting (no SEND clause)** If any user has no SEND AT or SEND EVERY clause, the Message Agent sends messages every time it is run, and then stops: it runs in batch mode.

Setting the send frequency in Sybase Central

In Sybase Central, you can specify the send frequency in the following ways:

- ◆ When you make an existing user or group remote. For more information, see [“Granting REMOTE permissions” on page 91](#).
- ◆ On the SQL Remote tab of the property sheet of a remote user or group. You can access the property sheet by right-clicking the remote user or group and choosing Properties from the popup menu.

Granting CONSOLIDATE permissions

In the remote database, the publish and subscribe user IDs are inverted compared to the consolidated database. The subscriber (remote user) in the consolidated database becomes the publisher in the remote database. The publisher of the consolidated database becomes a subscriber to publications from the remote database, and is granted CONSOLIDATE permissions.

At each remote database, the consolidated database must be granted CONSOLIDATE permissions. When you produce a remote database by running the database extraction utility, the GRANT CONSOLIDATE statement is executed automatically at the remote database.

Example

The following SQL Anywhere statement grants CONSOLIDATE permissions to the **hq_user** user ID, using the VIM email system:

```
GRANT CONSOLIDATE TO hq_user
TYPE vim
ADDRESS 'hq_address'
```

There is no SEND clause in this statement, so the default is used and messages will be sent to the consolidated database every time the Message Agent is run.

Revoking REMOTE and CONSOLIDATE permissions

A user can be removed from a SQL Remote installation by revoking their REMOTE permissions. When you revoke remote permissions from a user or group, you revert that user or group to a normal user/group. You also automatically unsubscribe that user or group from all publications.

Revoking permissions from Sybase Central

You can revoke REMOTE permissions on SQL Anywhere databases from Sybase Central.

◆ To revoke REMOTE permissions (Sybase Central)

1. Open either the Users & Groups folder or the SQL Remote Users folder.
2. Right-click the remote user or group and choose Revoke Remote from the popup menu.

Revoking permissions

REMOTE and CONSOLIDATE permissions can be revoked from a user using the REVOKE statement. The following statement revokes REMOTE permission from user **S_Beaulieu**.

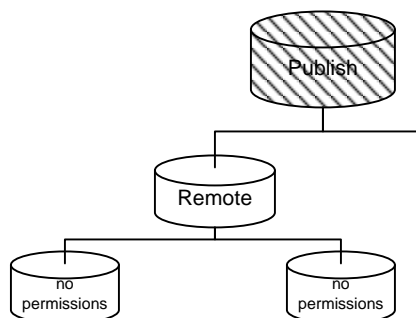
```
REVOKE REMOTE FROM S_Beaulieu
```

DBA authority is required to revoke REMOTE or CONSOLIDATE access.

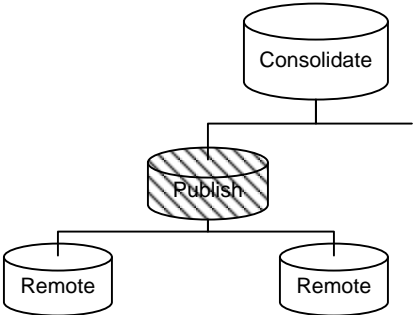
Assigning permissions in multi-tier installations

Special considerations are needed for assigning permissions in multi-tier installations. The permissions in a three-level SQL Remote setup are summarized in the following diagrams. In each diagram one database is shaded; the diagram shows the permissions that need to be granted in that database for the user ID representing each of the other databases. The phrase "No permissions" means that the database is not granted any permissions in the shaded database.

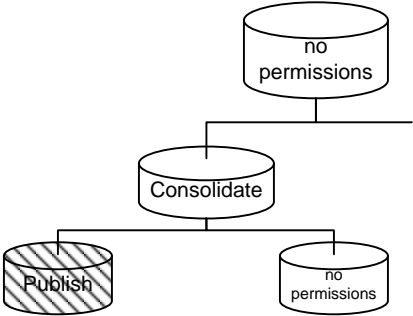
The following picture shows SQL Remote permissions, as granted at the consolidated site of a three-tier installation.



The following picture shows SQL Remote permissions, as granted at an internal site of a three-tier installation.



The following picture shows SQL Remote permissions, as granted at an internal site of a three-tier installation.



Granting the appropriate PUBLISH and CONSOLIDATE permissions at remote databases is done automatically by the database extraction utility.

Using message types


SQL Remote supports several different systems for exchanging messages. The message systems supported by SQL Remote are:

- ◆ **file** Storage of message files in directories on a shared file system for reading by other databases.
- ◆ **ftp** Storage of message files in directories accessible by a file transfer protocol (ftp) link.
- ◆ **mapi** Microsoft's messaging API (MAPI) link, used in Microsoft Mail and other electronic mail systems.
- ◆ **smtp** Internet Simple Mail Transfer Protocol (SMTP/POP), used in Internet email.
- ◆ **vim** Lotus's Vendor Independent Messaging (VIM), used in Lotus Notes and cc:Mail.

A database can exchange messages using one or more of the available message systems.

Operating system availability

Not all message systems are supported on all operating systems for which SQL Remote is available. The links are implemented as DLLs on Windows operating systems.

 For a listing of which message systems are supported on which operating system, see [Appendix "Supported Platforms and Message Links" on page 177](#).

See also

- ◆ ["The file message system" on page 100](#)
- ◆ ["The ftp message system" on page 101](#)
- ◆ ["The SMTP message system" on page 103](#)
- ◆ ["The MAPI message system" on page 105](#)
- ◆ ["The VIM message system" on page 106](#)

Working with message types

Each message type definition includes the type name (**file**, **ftp**, **smtp**, **mapi**, or **vim**) and also the address of the publisher under that message type. The publisher address at a consolidated database is used by the database extraction utility as a return address when creating remote databases. It is also used by the Message Agent to identify where to look for incoming messages for the **file** system.

The address supplied with a message type definition is closely tied to the publisher ID of the database. Valid addresses are considered in following sections.

Before you can use a message system, you must set the publisher's address.

Using Sybase Central to work with message types

You can create and alter message types in Sybase Central. Message types appear on the Message Types tab in the right pane when the SQL Remote Users folder is selected. This section applies only to SQL Anywhere databases.

You must have DBA authority to create and alter message types.

◆ To add a message type (Sybase Central)

1. Connect to a database.
2. In the left pane, open the SQL Remote Users folder for that database.
3. In the right pane, click the Message Types tab.
4. From the File menu, choose New ► Message Type.

The Create SQL Remote Message Type wizard appears.

5. In the Create SQL Remote Message Type wizard, enter a message type name. The name should correspond to a message-type DLL already installed in your SQL Anywhere directory. Click Next.
6. Enter a publisher address and click Finish to save the definition in the database.

If you want to change the publisher's address, you can do so by altering a message type. You cannot change the name of an existing message type; instead, you must delete it and create a new message type with the new name.

◆ To alter a message type (Sybase Central)

1. In the left pane, open the SQL Remote Users folder for a database.
2. In the right pane, click the Message Types tab.
3. In the right pane, right-click the message type you want to alter and choose Properties from the popup menu.
4. On the property sheet, configure the various options.

If you want to drop a message type from the installation, you can do so.

◆ To drop a message type (Sybase Central)

1. In the left pane, open the SQL Remote Users folder for a database.
2. In the right pane, click the Message Types tab.
3. In the right pane, right-click the message type you want to alter and choose Delete from the popup menu.

Creating message types for Windows CE

If you have Windows CE services installed, you have an option to set up SQL Remote for ActiveSync synchronization from Sybase Central. This sets your folder for FILE message link messages to be the ActiveSync folder. When you dock your Windows CE device to your desktop computer, ActiveSync keeps the files in your desktop computer's ActiveSync folder synchronized with those in the Windows CE ActiveSync folder. You can access the utility to set up SQL Remote ActiveSync synchronization by choosing Tools ► SQL Anywhere 10 ► Edit Windows CE Message Type.

Using commands to work with message types

◆ To create a message type (SQL)

1. Make sure you have decided on an address for the publisher under the message type.
2. Execute a CREATE REMOTE MESSAGE TYPE command.

The CREATE REMOTE MESSAGE TYPE statement has the following syntax:

```
CREATE REMOTE MESSAGE TYPE type-name  
ADDRESS address-string
```

where *type-name* is one of the message systems supported by SQL Remote, and *address-string* is the publisher's address under that message system.

If you want to change the publisher's address, you can do so by altering the message type.

◆ To alter a message type (SQL)

1. Make sure you have decided on a new address for the publisher under the message type.
2. Execute an ALTER REMOTE MESSAGE TYPE statement.

The ALTER REMOTE MESSAGE TYPE statement has the following syntax:

```
ALTER REMOTE MESSAGE TYPE type-name  
ADDRESS address-string
```

where *type-name* is one of the message systems supported by SQL Remote, and *address-string* is the publisher's address under that message system.

You can also drop message types if they are no longer used in your installation. This has the effect of removing the publisher's address from the definition.

◆ To drop a message type (SQL)

- Execute a DROP REMOTE MESSAGE TYPE statement.

The DROP REMOTE MESSAGE TYPE statement has the following syntax:

```
DROP REMOTE MESSAGE TYPE type-name
```

where *type-name* is one of the message systems supported by SQL Remote.

See also

- ◆ “CREATE REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “ALTER REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “DROP REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]

Setting message type control parameters

Each message link has several parameters that govern aspects of its behavior. The parameters differ from message system to message system, but all are managed in the same way.

When you first use the Message Agent for a particular message link, it displays a dialog box showing a set of parameters that control the behavior of the link. These parameters may be a user ID for the message system, a host name where ftp messages are held, and so on. The parameters you enter are saved by the Message Agent. You can also set these parameters explicitly.

Message link parameters stored in the database

The message control parameters are held in the database. You can set the options as follows:

◆ To set a message control parameter

- Execute the following statement:

```
SET REMOTE link-name OPTION
[username.]option-name = option-value
```

You can see the current message link parameters by querying the sys.sysremoteoptions view.

Holding the message link parameters on disk

Earlier versions of this software stored the message link parameters outside the database. You can still use this method, but storing the parameters inside the database is recommended unless you have specific reasons to choose otherwise.

The message link control parameters are stored in the following places:

- ◆ **Windows** In the registry, at the following location:

```
\\HKEY_CURRENT_USER
  \Software
    \Sybase
      \SQL Remote
```

The parameters for each message link go in a key under the SQL Remote key, with the name of the message link (4, *smtp*, and so on).

- ◆ **NetWare** You should create a file named *dbremote.ini* in the *sys:\system* directory to hold the FILE system directory setting. This file is not a Windows-format INI file: it must consist of a single line, holding only the directory name.

For example, if the directory is *user:\dbr43*, then the *dbremote.ini* file would contain the following:

```
user:\dbr43
```

- ◆ **Unix** The FILE system directory setting is held in the SQLREMOTE environment variable.

The *sqlremote* environment variable holds a path that can be used as an alternative to one of the control parameters for the file sharing system.

The parameters available for each message system are discussed in the following sections. Each section describes a single message system.

When the Message Agent loads a message link, the link uses the settings of the current publisher or, if a setting is not specified, of groups to which the publisher belongs. On Windows, the first time a version of the Message Agent is run that supports storing the message link parameters in the database, it copies the link options from the registry to the database.

The file message system

SQL Remote can be used even if you do not have a message system in place, by using the **file** message system.

Addresses in the file message system

The **file** message system is a simple file-sharing system. A **file** address for a remote user is a subdirectory into which all their messages are written. To retrieve messages from their "inbox", an application reads the messages from the directory containing the user's files. Return messages are sent to the address (written to the directory) of the consolidated database.

When running as an NT service make sure that the account under which the Message Agent is running has permissions to read and write all necessary directories. This is often a problem when accessing network drives.

Root directory for addresses

The **file** system addresses are typically subdirectories of a shared directory that is available to all SQL Remote users, whether by modem or on a local area network. Each user should have a registry entry, initialization file entry, or SQLREMOTE environment variable pointing to the shared directory.

You can also use the **file** system to put the messages in directories on the consolidated and remote machines. A simple file transfer mechanism can then be used to exchange the files periodically to effect replication.

FILE message control parameters

The FILE message system uses the following control parameters:

- ◆ **Directory** This is set to the directory under which the messages are stored. The setting is an alternative to the SQLREMOTE environment variable.

- ◆ **Debug** This is set to either YES or NO, with the default being NO. When set to YES, all file system calls made by the FILE link are displayed.
- ◆ **Encode_dll** If you have implemented a custom encoding scheme, you must set this to the full path of the custom encoding DLL that you created.

See [“Encoding and compressing messages” on page 118](#).

- ◆ **invalid_extensions** A comma-separated list of file extensions that you do not want dbremote to use when generating files in the messaging system.
- ◆ **Unlink_delay** This is the number of seconds to wait before attempting to delete a file if the previous attempt to delete the file failed. If no value is defined for unlink_delay, then the default behavior is to pause for 1 second after the first failed attempt, 2 seconds after the second failed attempt, 3 seconds after the third failed attempt, and 4 seconds after the fourth failed attempt.

On NetWare, you should create a file named *dbremote.ini* in the *sys:\system* directory to hold the directory setting.


See also

- ◆ [“The ftp message system” on page 101](#)

The ftp message system

Addresses for ftp

In the ftp message system, messages are stored in directories under a root directory on an ftp host. The ftp host and the root directory are specified by message system control parameters held in the registry or initialization file, and the address of each user is the subdirectory where their messages are held.

 For a list of operating systems for which ftp is supported, see [“Supported operating systems” on page 179](#).

FTP message control parameters

The ftp message system uses the following control parameters:

- ◆ **encode_dll** If you have implemented a custom encoding scheme, you must set this to the full path of the custom encoding DLL that you created.

See [“Encoding and compressing messages” on page 118](#).

- ◆ **host** The host name of the computer where the messages are stored. This can be a host name (such as **ftp.ianywhere.com**) or an IP address (such as 192.138.151.66).
- ◆ **user** The user name for accessing the ftp host.
- ◆ **password** The password for accessing the ftp host.
- ◆ **root_directory** The root directory within the ftp host site, under which the messages are stored.

- ◆ **port** Usually not required. This is the IP port number used for the Ftp connection.
- ◆ **debug** This is set to either YES or NO, with the default being NO. When set to YES, debugging output is displayed.
- ◆ **active_mode** This is set to either YES or NO, with the default being NO (passive mode).
- ◆ **reconnect_retries** The number of times the link should try to open a socket with the server before failing. The default value is 4. The setting of this parameter only affects reconnects. The initial connection made by the FTP link is not affected.
- ◆ **reconnect_pause** The time in seconds to pause between each connection attempt. The default setting is 30 seconds. The setting of this parameter only affects reconnects. The initial connection made by the FTP link is not affected.
- ◆ **suppress_dialogs** If set to true, the Connect dialog does not appear after failed attempts to connect to the FTP server. Instead, an error is generated.
- ◆ **invalid_extensions** A comma-separated list of file extensions that you do not want dbremote to use when generating files in the messaging system.

Troubleshooting ftp problems

Most problems with the FTP message link are network setup issues. This section contains a list of tests you can try to troubleshoot problems.

Set the DEBUG message control parameter Looking over the debug output should indicate whether you are connecting to the FTP server. If you are connecting, it will indicate which FTP commands are failing.

Ping the ftp server If the FTP link is not able to connect to the FTP server, try testing your systems network configuration. If your system has the **ping** command, try typing the following command:

```
ping ftp-server-name
```

You should see output indicating the IP address of the server and the ping (round trip) time to the server. If you cannot ping the server then you have a network configuration problem, and you should contact your network administrator.

Check that passive mode works If the FTP link is connecting to the FTP server, but is unable to open a data connection, make sure that an FTP client can use passive mode to transfer data with the server.

Passive mode is the preferred transfer mode and the default for the FTP message link. In passive mode all data transfer connections are initiated by the client, in this case the message link. In Active mode the server initiates all data connections. If your FTP server is sitting behind an incorrectly configured firewall you may not be able to use the default passive transfer mode. In this situation the firewall blocks socket connections to the FTP server on ports other than the FTP control port.

Using an FTP user program that allows you to set the transfer mode between **active** and **passive**, set the transfer mode to passive and try to upload/download a file. If the client you are using cannot transfer the file without using active mode then you should either reconfigure the firewall and FTP server to allow passive mode transfers or set the active_mode message control parameter to YES. Active mode transfers may not

work in all network configurations. For example: if your client is sitting behind an IP masquerading gateway incoming connections may fail depending on you gateway software.

Check permissions and directory structures If the FTP server is connecting and having problems getting directory listings or manipulating files; make sure your permissions are set up correctly and the directories that you need exist.

Log in to the FTP server using an FTP program. Change directories to the location stored in the root_directory parameter. If the directories you need do not show up, the root_directory control parameter may be wrong or the directories may not exist.

Test permissions by fetching a file in your message directory and uploading a file to the consolidated database directory. If you get errors your FTP server permissions are set up incorrectly.

The SMTP message system

The Simple Mail Transfer Protocol (SMTP) is used in Internet email products.

With the SMTP system, SQL Remote sends messages using Internet mail. The messages are encoded to a text format and sent in an email message to the target database. The messages are sent using an SMTP server, and retrieved from a POP server: this is the way that many email programs send and receive messages.

☞ For a list of operating systems for which SMTP is supported, see [“Supported operating systems” on page 179](#).

SMTP addresses and user IDs

To use SQL Remote and an SMTP message system, each database participating in the setup requires a SMTP address, and a POP3 user ID and password. These are distinct identifiers: the SMTP address is the destination of each message, and the POP3 user ID and password are the name and password entered by a user when they connect to their mail box.

Separate email account recommended

It is recommended that a separate POP email account be used for SQL Remote messages.

Troubleshooting

If you cannot get the SMTP Link to work try connecting to the SMTP/POP3 server from the same machine on which the Message Agent is running using the same account and password. Use an Internet email program that supports SMTP/POP3 and make sure to disable the program once the SMTP message link is working.

SMTP message control parameters

Before the Message Agent connects to the message system to send or receive messages, the user must either have a set of control parameters already set on their machine, or must fill in a window with the needed information. This information is needed only on the first connection. It is saved and used as the default entries on subsequent connects.

The SMTP message system uses the following control parameters:

- ◆ **encode_dll** If you have implemented a custom encoding scheme, you must set this to the full path of the custom encoding DLL that you created.

See [“Encoding and compressing messages” on page 118](#).

- ◆ **local_host** This is the name of the local computer. It is useful on machines where SQL Remote is unable to determine the local host name. The local host name is needed to initiate a session with any SMTP server. In most network environments, the local host name can be determined automatically and this entry is not needed.
- ◆ **TOP_supported** SQL Remote uses a POP3 command called TOP when enumerating incoming messages. The TOP command may not be supported by all POP servers. Setting this entry to NO will use the RETR command, which is less efficient but will work with all POP servers. The default is YES.
- ◆ **smtp_authenticate** Determines whether the SMTP link authenticates the user. The default value is YES. Set to NO for no SMTP authentication to be carried out.
- ◆ **smtp_userid** The user ID for SMTP authentication. By default this parameter takes the same value as the **pop3_userid** parameter. The **smtp_userid** only needs to be set if the user ID is different to that on the POP server.
- ◆ **smtp_password** The password for SMTP authentication. By default this parameter takes the same value as the **pop3_password** parameter. The **smtp_password** only needs to be set if the user ID is different to that on the POP server.
- ◆ **smtp_host** This is the name of the computer on which the SMTP server is running. It corresponds to the SMTP host field in the SMTP/POP3 login dialog.
- ◆ **pop3_host** This is the name of the computer on which the POP host is running. It is commonly the same as the SMTP host. It corresponds to the POP3 host field in the SMTP/POP3 login dialog.
- ◆ **pop3_userid** This is used to retrieve mail. The POP user ID corresponds to the user ID field in the SMTP/POP3 login dialog. You must obtain a user ID from your POP host administrator.
- ◆ **pop3_password** This is used to retrieve mail. It corresponds to the password field in the SMTP/POP3 login dialog. If all of these five fields are set, the login dialog is not displayed.
- ◆ **Debug** When set to YES, displays all SMTP and POP3 commands and responses. This is useful for troubleshooting SMTP/POP support problems. Default is NO.
- ◆ **Suppress_dialogs** If set to true, the Connect dialog does not appear after failed attempts to connect to the mail server. Instead, an error is generated.

Sharing SMTP/POP addresses

The database should have its own email account for SQL Remote messages, separate from personal email messages intended for reading. This is because many email readers will collect email in the following manner:

1. Connect to the POP Host and download all messages.
2. Delete all messages from POP Host
3. Disconnect from POP Host.
4. Read mail from the local file or from memory

This causes a problem, as the email program downloads and deletes all of the SQL Remote email messages as well as personal messages. If you are certain that your email program will not delete unread messages from the POP Host then you may share an email address with the database as long as you take care not to delete or alter the database messages.

These messages are easy to recognize, as they are filled with lines of seemingly random text.

The MAPI message system

The Message Application Programming Interface (MAPI) is used in several popular email systems, such as Microsoft Mail and later versions of Lotus cc:Mail.

☞ For a list of operating systems for which MAPI is supported, see [“Supported operating systems” on page 179](#).

MAPI addresses and user IDs

To use SQL Remote and a MAPI message system, each database participating in the setup requires a MAPI user ID and address. These are distinct identifiers: the MAPI address is the destination of each message, and the MAPI user ID is the name entered by a user when they connect to their mail box.

MAPI message and the email inbox

Although SQL Remote messages may arrive in the same mailbox as email intended for reading, they do not in general show up in your email inbox.

SQL Remote attempts to send application-defined messages, which MAPI identifies and hides when the mailbox is opened. In this way, users can use the same email address and same connection to receive their personal email and their database updates, yet the SQL Remote messages do not interfere with the mail intended for reading.

If a message is routed via the Internet, or if certain security patches are applied to the operating system, the special message type information can be lost. The message then does show up in the recipient's mailbox.

MAPI message control parameters

The MAPI message system uses the following control parameters:

- ◆ **Debug** When set to YES, displays all MAPI calls and the return codes. This is useful for troubleshooting MAPI support problems. Default is NO.
- ◆ **Encode_dll** If you have implemented a custom encoding scheme, you must set this to the full path of the custom encoding DLL that you created.

See [“Encoding and compressing messages” on page 118](#).

- ◆ **Force_Download** (default YES) controls if the MAPI_FORCE_DOWNLOAD option is set when calling MapiLogon. This might be useful when using remote mail software that dials when this option is set.
- ◆ **IPM_Receive** This can be set to YES or NO (default YES). If set to NO, the MAPI link receives IPC messages, which are not visible in the mailbox. If set to YES, the MAPI link receives IPM messages, which are visible in the mailbox, and IPC messages, which are not visible in the mailbox. It is recommended that this value remain at YES to ensure that both IPC and IPM messages are picked up by SQL Remote.
- ◆ **IPM_Send** This can be set to YES or NO (default NO). If set to YES, the MAPI link sends IPM messages, which are visible in the mailbox. If set to NO, the MAPI link attempts to send IPC messages, which are not visible in the mailbox. If a message is routed via the Internet, or if certain security patches are applied to the operating system, SQL Remote may be unable to send IPC messages.
- ◆ **Profile** Use the specified Microsoft Exchange profile. You should use this if you are running the Message Agent as a service.
- ◆ **Suppress_dialogs** If set to true, the Connect dialog does not appear after failed attempts to connect to the mail server. Instead, an error is generated.

The VIM message system

The Vendor Independent Messaging system (VIM) is used in Lotus Notes and in some releases of Lotus cc:Mail.

To use SQL Remote and a VIM message system, each database participating in the setup requires a VIM user ID and address. These are distinct identifiers: the VIM address is the destination of each message, and the VIM user ID is the name entered by a user when they connect to their mail box.

☞ For a list of operating systems for which VIM is supported, see [“Supported operating systems” on page 179](#).

VIM message control parameters

The VIM message system uses the following control parameters:

- ◆ **Path** This corresponds to the Path field in the cc:Mail login dialog. It is not applicable to and is ignored under Lotus Notes.
- ◆ **Userid** This corresponds to the User ID field in the cc:Mail login dialog.
- ◆ **Password** This corresponds to the Password field in the cc:Mail login dialog. If all of Path, Userid, and Password are set, the login dialog is not displayed.
- ◆ **Debug** When set to YES, displays all VIM calls and the return codes. This is useful for troubleshooting VIM support problems. Default is NO.
- ◆ **Encode_dll** If you have implemented a custom encoding scheme, you must set this to the full path of the custom encoding DLL that you created.

See [“Encoding and compressing messages”](#) on page 118.

- ◆ **Receive_All** When set to YES, the Message Agent checks all messages to see if they are SQL Remote messages. When set to NO (the default), the Message Agent looks only for messages of the application-defined type **SQLRemoteData**. This leads to improved performance in Notes.

Setting Receive_All to YES is useful in setups where the message type is lost, reset, or never set. This includes setups including cc:Mail messages, or over the Internet.

- ◆ **Suppress_dialogs** If set to true, the Connect dialog does not appear after failed attempts to connect to the mail server. Instead, an error is generated.
- ◆ **Send_VIM_Mail** When set to YES, the Message Agent sends messages compatible with Adaptive Server Anywhere releases before 5.5.01, and compatible with cc:Mail. If this is set to YES, you should ensure that **Receive_All** is set to YES also.

Running the Message Agent

The SQL Remote Message Agent is a key component in SQL Remote replication. The Message Agent handles both the sending and receiving of messages. It carries out the following functions:

- ◆ It processes incoming messages, and applies them in the proper order to the database.
- ◆ It scans the transaction log or stable queue at each publisher database, and translates the log entries into messages for subscribers.
- ◆ It parcels the log entries up into messages no larger than a fixed maximum size (50,000 bytes by default), and sends them to subscribers.
- ◆ It maintains the message tracking information in the system tables, and manages the guaranteed transmission mechanism.

Executable names

On Windows operating systems, the Message Agent is named *dbremote.exe*. On Unix operating systems, the name is *dbremote*.

Message Agent batch and continuous modes

The Message Agent can be run in one of two modes:

- ◆ **Batch mode** In batch mode, the Message Agent starts, receives and sends all messages that can be received and sent, and then shuts down.

Batch mode is useful at occasionally-connected remote sites, where messages can only be exchanged with the consolidated database when the connection is made: for example, when the remote site dials up to the main network.

- ◆ **Continuous mode** In continuous mode, the Message Agent periodically sends messages, at times specified in the properties of each remote user. When it is not sending messages, it receives messages as they arrive.

Continuous mode is useful at consolidated sites, where messages may be coming in and going out at any time, to spread out the workload and to ensure prompt replication.

The options available depend on the send frequency options selected for the remote users. Sending frequency options are described in [“Selecting a send frequency” on page 92](#).

- ◆ **To run the Message Agent in continuous mode**

1. Ensure that every user has a sending frequency specified. The sending frequency is specified by a `SEND AT` or `SEND EVERY` option in the `GRANT REMOTE` statement.
2. Start the Message Agent without using the `-b` option.

◆ To run the Message Agent in batch mode

- Either:
 - ◆ Have at least one remote user who has neither a SEND AT nor a SEND EVERY option in their remote properties, or
 - ◆ Start the Message Agent using the -b option.

Connections used by the Message Agent

The Message Agent uses a number of connections to the database server. These are:

- ◆ One global connection, alive all the time the Message Agent is running.
- ◆ One connection for scanning the log. This connection is alive during the scan phase only.
- ◆ One connection for executing commands from the log-scanning thread. This connection is alive during the scan phase only.
- ◆ One connection for processing synchronize subscription requests. This connection is alive during the send phase only.
- ◆ One connection for each worker thread. These connections are alive during the receive phase only.

Replication system recovery procedures

SQL Remote replication places new requirements on data recovery practices at consolidated database sites. Standard backup and recovery procedures enable recovery of data from system or media failure. In a replication installation, even if such recovery is achieved, the recovered database can be out of synch with remote databases. This can require a complete resynchronization of remote databases, which can be a formidable task if the installation involves large numbers of databases.

In short, recovery of the consolidated database from a failure at the consolidated site is only part of the task of recovering the entire replication installation.

Protection of the replication system against media failures has two aspects:

- ◆ **Backup and log management** Solid backup procedures and log management procedures for the consolidated database server are an essential part of recovery plans. Backup procedures protect against media failure on the database device. Using a transaction log mirror protects against media failure on the transaction log device.
For more information about backup and log management procedures, see [“Transaction log and backup management” on page 130](#).
- ◆ **Message Agent configuration** The Message Agent command line options provide ways for you to tune Message Agent behavior to match your backup and recovery requirements.

Message Agent configuration is discussed in the following pages.

Replicating only backed-up transactions

By default, the Message Agent processes all committed transactions. When the Message Agent is run with the -u option, only transactions that have been backed up by the database backup commands are processed.

Transaction log backup is carried out using Sybase Central or the *dbbackup* utility, or off-line copying and renaming of the log file.

By sending only backed-up transactions, the replication installation is protected against media failure on the transaction log. Maintaining a mirrored transaction log also accomplishes this goal.

The -u option provides additional protection against total site failure, if backups are carried out to another site.

Ensuring consistent Message Agent settings

Some Message Agent settings need to be the same throughout an installation, and so should be set before deployment. This section lists the settings that need to be the same.

- ◆ **Maximum message length** The maximum message length for SQL Remote messages has a default value of 50K. This is configurable, using the Message Agent -l option. However, the maximum message length must be the same for each Message Agent in the installation, and may be restricted by operating system memory allocation limits.

Received messages that are longer than the limit are deleted as corrupt messages.

For details of this setting, see [“Message Agent” on page 148](#).

The Message Agent and replication security

Messages sent by the SQL Remote Message Agent have a very simple encryption that protects against casual snooping. However, the encryption scheme is not intended to provide full protection against determined efforts to decipher them.

Troubleshooting errors at remote sites

There are obvious obstacles for an administrator who has access only to the consolidated site to troubleshoot errors that occur at remote sites. To assist with this task, you can set up SQL Remote so that portions of the output log from remote sites are delivered to the consolidated site and written to a file. This one file contains logging information from some or all sites in the system.

To set up SQL Remote to collect log information, you must configure both the remote and the consolidated sites.

- ◆ **To configure a remote database to send log information to the consolidated database**

1. Set a link option to send log information when an error is encountered.

Execute the following command against the remote database:

```
SET REMOTE link-name OPTION  
PUBLIC.output_log_send_on_error = 'Yes'
```

With this option set, any message that starts with the error indicator 'E' causes SQL Remote to send log information to the consolidated site.

☞ For more information, see “[SET REMOTE OPTION statement \[SQL Remote\]](#)” [[SQL Anywhere Server - SQL Reference](#)].

2. Set a link option to limit the amount of information sent to the consolidated site. This step is optional.

Execute the following command against the remote database:

```
SET REMOTE link-name OPTION  
PUBLIC.output_log_send_limit = 'nnn'
```

The value of this option is the number of bytes at the tail of the output log (that is, the most recent entries) which are sent to the consolidated site. You can use *nnnK* to indicate kilobytes. The default setting is '5K'.

If you supply a value that is too large to fit in the maximum message size, SQL Remote overrides the option value and sends only what will fit in the message.

You can also send log information even in the absence of errors by setting the `output_log_send_now` option to Yes. SQL Remote then sends the output log information on the next poll and resets the option to 'NO' after the log is sent.

◆ To configure a consolidated site to receive log information

- Use either the `-ro` or the `-rt` Message Agent option.

☞ For more information, see “[Message Agent](#)” on page 148.

Tuning Message Agent performance

Who needs to read this section?

If performance is not a problem at your site, you do not need to read this section.

There are several options you can use to tune the performance of the Message Agent. This section describes those options.

Sending messages and receiving messages are two separate processes. The major performance issues for these two processes are different.

- ◆ **Replication throughput** The major bottleneck for total throughput of SQL Remote sites is generally receiving messages from many remote databases and applying them to the database at the consolidated site. You can control this step by tuning the receive process of the Message Agent at the consolidated site.
- ◆ **Replication turnaround** The time lag from when data is entered at one site to when it appears at other sites is the turnaround time for replication. You can control this time lag.

Tuning throughput by controlling Message Agent threading

It is assumed in this section that you are tuning the performance of a Message Agent that is running in continuous mode at a consolidated site.

Worker threads can be used by the Message Agent to apply incoming messages from remote users. This can improve throughput by allowing messages to be applied in parallel rather than serially.

Setting the number of worker threads

The number of worker threads is set on the Message Agent command line, using the `-w` option. The default is to use no worker threads, so that all messages are applied serially. The maximum number of worker threads is 50.

Performance benefits from worker threads

The performance advantage will be most significant when the server is on a system with a striped drive array.

What messages are applied in parallel

When worker threads are being used, messages from different remote users are applied in parallel. Messages from a single remote user are applied serially. For example, ten messages from a single remote user will be applied by a single worker thread in the correct order.

Deadlock is handled by re-applying the rolled back transaction at a later time.

Reading messages from the message system is single-threaded. Messages are read and the header information is examined (to determine the remote user and the correct order of application) before passing them off to worker threads to be applied.

Building messages and sending messages is single-threaded.

Tuning throughput by caching messages

The Message Agent caches incoming messages in a configurable area of memory as it reads them.

Specifying the message cache size

The size of the message cache is specified on the Message Agent command line, using the `-m` option.

The `-m` option specifies the maximum amount of memory to be used by the Message Agent for building messages. The allowed size can be specified as n (in bytes), nK , or nM . The default is 2048K (2M).

Example

The following command line starts a Message Agent using twelve Megabytes of memory as a message cache:

```
dbremote -c "eng=..." -m 12M
```

How messages are cached

When transactions are large, or messages arrive out of order, they are stored in memory by the Message Agent until the message is to be applied. This caching of messages prevents rereading of out-of-order messages from the message system, which may lower performance on large installations. It is especially important when messages are being read over a WAN (such as Remote Access Services or POP3 through a modem). It also avoids contention between worker threads reading messages (a single threaded task) because the message contents are cached.

When the memory usage specified using the `-m` option is exceeded, messages are flushed in a least-recently-used fashion.

This option is provided primarily for customers considering a single consolidated database for thousands of remote databases.

Tuning incoming message polling

When running a Message Agent in continuous mode, typically at a consolidated database site, you can control how often it polls for incoming messages, and how "patient" it is in waiting for messages that arrive out of order before requesting that the message be resent. Tuning these aspects of the behavior can have a significant effect on performance in some circumstances.

Issues to consider

The issues to consider when tuning the message-receiving process are similar to those when tuning the message-sending process.

- ◆ **Regular messages** Your choices dictate how often the Message Agent polls for incoming messages from remote databases.
- ◆ **Resend requests** You can control how many polls to wait until an out-of-order message arrives, before requesting that it be resent.
- ◆ **Processing incoming messages** If your polling period for incoming messages is too long, compared to the frequency with which messages are arriving, you could end up with messages sitting in

the queue, waiting to be processed. If your polling period is too short, you will waste resources polling when no messages are in the queue.

☞ For more information on the message sending process, see [“Tuning the message sending process” on page 116](#).

Polling interval

By default, a Message Agent running in continuous mode polls one minute after finishing the previous poll, to see whether new messages have arrived. You can configure the polling interval using the `-rd` option.

The default polling interval from the end of one poll to the start of another is one minute. You can poll more frequently using a value in seconds, as in the following command line:

```
dbremote -rd 30s
```

Alternatively, you can poll less frequently, as in the following command line, which polls every five minutes:

```
dbremote -rd 5
```

Setting a very small interval may have some detrimental impact on overall system throughput, for the following reasons:

- ◆ Each poll of the mail server (if you are using email) places a load on your message system. Too-frequent polling may affect your message system and produce no benefits.
- ◆ If you do not modify the Message Agent patience before it assumes that an out of sequence message is lost, and requests it be sent again, you can flood your system with resend requests.

In general, you should not use a very small polling interval unless you have a specific reason for requiring a very quick response time for messages.

Setting larger intervals may provide a better overall throughput of messages in your system, at the cost of waiting somewhat longer for each message to be applied. In many SQL Remote installations, optimizing turnaround time is not the primary concern.

Requesting resends

If, when the Message Agent polls for incoming messages, one message is missing from a sequence, the Message Agent does not immediately request that the message be resent. Instead, it has a default **patience** of one poll.

If the next message expected is number 6 and message 7 is found, the Message Agent takes no action until the next poll. Then, if no new message for that user is found, it issues a resend request.

You can change the number of polls for which the Message Agent waits before sending a request using the `-rp` option. This option is often used in conjunction with the `-rd` option that sets the polling interval.

For example, if you have a very small polling interval, and a message system that does not preserve the order in which messages arrive, it may be very common for out-of-sync messages to arrive only after two

or three polls have been completed. In such a case, you should instruct the Message Agent to be more patient before sending a resend request, by increasing the `-rp` value. If you do not do this, a large number of unnecessary resend requests may be sent.

Example

Suppose there are two remote users, named **user1** and **user2**, and suppose the Message Agent command line is as follows:

```
dbremote -rd 30s -rp 3
```

In the following sequence of operations, messages are marked as *userX.n* so that **user1.5** is the sixth message from user1. The Message Agent expects messages to start at number 1 for both users.

At time 0 seconds:

1. The Message Agent reads user1.1, user2.4
2. The Message Agent applies user1.1
3. The Message Agent patience is now user1: N/A, user2: 3, as an out of sequence message has arrived from user 2.

At time 30 seconds:

1. The Message Agent reads: no new messages
2. The Message Agent applies: none
3. The Message Agent patience is now user1: N/A, user2: 2

At time 60 seconds:

1. The Message Agent reads: user1.3
2. The Message Agent applies: no new messages
3. The Message Agent patience: user1: 3, user2: 1

At time 90 seconds:

1. The Message Agent reads: user1.4
2. The Message Agent applies: none
3. The Message Agent patience user1: 3, user2: 0
4. The Message Agent issues resend to user2.

When a user receives a new message, it resets the Message Agent patience even if that message is not the one expected.

Tuning the message sending process

The turnaround time for replication is governed by how often each sites sends messages and how often each site polls for incoming messages. To achieve a small time lag between data entry and data replication, you can set a small value for the `-sd` Message Agent option, which controls the frequency for polling to see if more data needs to be sent.

Issues to consider

The issues to consider when tuning the message-sending process are similar to those when tuning the incoming-message polling frequency:

- ◆ **Regular messages** Your choices dictate how often updates are sent to remote databases.
- ◆ **Resend requests** When a remote user requests that a message be resent, the Message Agent needs to take special action that can interrupt regular message sending. You can control the urgency with which these resend requests are processed.
- ◆ **Number and size of messages** If you send messages very frequently, there is more chance of small messages being sent. Sending messages less frequently allows more instructions to be grouped in a single message. If a large number of small messages is a concern for your message system, then you may have to avoid using very small polling periods.

☞ For more information on tuning polling for the incoming-messages, see [“Tuning incoming message polling” on page 113](#).

Polling interval

You control the interval to wait between polls for more data from the transaction log to send using the `-sd` option, which has a default of one minute. The following example sets the polling interval to 30 seconds:

```
dbremote -sd 30s ...
```

Alternatively, you can poll less frequently, as in the following command line, which polls every five minutes:

```
dbremote -sd 5
```

Setting a very small interval may have some detrimental impact on overall system throughput, for the following reasons:

- ◆ Too-frequent polling produces many short messages. If the message load places a strain on your message system, throughput could be affected.

Setting larger intervals may provide a better overall throughput of messages in your system, at the cost of waiting somewhat longer for each message to be applied. In many SQL Remote installations, optimizing turnaround time is not the primary concern.

Resending messages

When a user requests that a message be resent, the message has to be retrieved from early in the transaction log. Going back in the transaction log to retrieve this message and send it causes the Message Agent to interrupt the regular sending process. If you are tuning your SQL Remote installation for optimum performance, you must balance the urgency of sending requests for resent messages with the priority of processing regular messages.

The `-ru` option controls the urgency of the resend requests. The value for the parameter is a time in minutes (or in other units if you add `s` or `h` to the end of the number), with a default of zero.

To help the Message Agent delay processing resend requests until more have arrived before interrupting the regular message sending activity, set this option to a longer time.

The following command line waits one hour until processing a resend request.

```
dbremote -ru 1h ...
```

If you do not specify the `-ru` option, then a default value is picked by the Message Agent, based on the send interval of the users that have requested that data be resent. The elapsed time between receiving a resend request for a user and rescanning the log does not exceed half of the send interval for that user.

Encoding and compressing messages

As messages pass through email and other message systems, there is a danger of them becoming corrupted. For example, some message systems use certain characters or character combinations as control characters.

Message size affects the efficiency with which messages pass through a system. Compressed messages can be processed more efficiently by a message system than uncompressed messages. On the other hand, carrying out compression can itself take a significant amount of time.

SQL Remote encoding and compression

SQL Remote has a message encoding and compression scheme built in to the Message Agent. The scheme provides the following features:

- ◆ **Compatibility** The system can be set up to be compatible with previous versions of the software.
- ◆ **Compression** You can select a level of compression for your messages.
- ◆ **Encoding** SQL Remote encodes messages to ensure that they pass through message systems uncorrupted. The encoding scheme can be customized to provide extra features.

Settings for compatibility

To be compatible with previous versions of the software, you should set the compression database option to -1 (minus one) at each database running the Version 6 software. This setting ensures that messages are sent out in a format compatible with older versions of the software.

Upgrading SQL Remote

If you upgrade the consolidated database Message Agent first, you should set its compression database option to -1. As each remote site in your replication system is upgraded to Version 6, you can change its setting of the compression database option to a value between 0 (no compression) and 9 (maximum compression). This allows you to take advantage of compression features on messages being sent to the consolidated database. Once all remote sites are upgraded, you can set the consolidated site Message Agent compression option to a value other than -1.

In addition, setting the compression option to a value other than -1 allows you to take advantage of the Version 6 message encoding improvements.

The encoding scheme

The default message-encoding behavior of SQL Remote is as follows:

- ◆ For message systems that can use binary message formats, no encoding is carried out.
- ◆ Some message systems, including SMTP, VIM, and MAPI, require text-based message formats. For these systems, an encoding DLL (*dbencod.dll*) translates messages into a text format before sending. The message format is unencoded at the receiving end using the same DLL.
- ◆ You can instruct SQL Remote to use a custom encoding scheme. The tools for building a custom encoding scheme are described in the following section.

- ◆ If the compression database option is set to -1, then a Version 5 compatible encoding is carried out for all message systems.

Creating custom encoding schemes

You can implement a custom encoding scheme by building a custom encoding DLL. You could use this DLL to apply special features required for a particular messages system, or to collect statistics, such as how many messages or how many bytes were sent to each user.

The header file *dbrmt.h*, installed into the *h* subdirectory of your installation directory, provides an application programming interface for building such a scheme.

To instruct SQL Remote to use your DLL for a particular message system, you need to set a message control parameter called `encode_dll` to a value that is the full path to the custom DLL you have created. For example:

```
SET REMOTE ftp OPTION "Public"."encode_dll" = 'c:\\sany10\\win32\\custom.dll';
```

Encoding and decoding must be compatible

If you implement a custom encoding, you must make sure that the DLL is present at the receiving end, and that the DLL is in place to decode your messages properly.

The message tracking system

SQL Remote has a message tracking system to ensure that all replicated operations are applied in the correct order, no operations are missed, and no operation is applied twice.

Message system failures may lead to replication messages not reaching their destination, or reaching it in a corrupt state. Also, messages may arrive at their destination in a different order from that in which they were sent. This section describes the SQL Remote system for detecting and correcting message system errors, and for ensuring correct application of messages.

If you are using an email message system, you should confirm that email is working properly between the two machines if SQL Remote messages are not being sent and received properly.

The SQL Remote message tracking system is based on status information maintained in the **remoteuser** SQL Remote system table. The table is maintained by the Message Agent. The Message Agent at a subscriber database sends confirmation to the publisher database to ensure that **remoteuser** is maintained properly at each end of the subscription.

The **remoteuser** table is the **sys.sysremoteuser** system table.

Status information in the remoteuser table

The **remoteuser** SQL Remote system table contains a row for each subscriber, with status information for messages sent to and received by that subscriber. At the consolidated database, **remoteuser** contains a row for each remote user. At each remote database, **remoteuser** contains a single row maintaining information for the consolidated database. (Recall that the consolidated database subscribes to publications from the remote database.)

The **remoteuser** SQL Remote system table at each end of a subscription is maintained by the Message Agent.

Tracking messages by transaction log offsets

The message-tracking status information takes the form of offsets in the transaction logs of the publisher and subscriber databases. Each COMMIT is marked in the transaction log by a well-defined offset. The order of transactions can be determined by comparing their offset values.

Message ordering

When messages are sent, they are ordered by the offset of the last COMMIT of the preceding message. If a transaction spans several messages, there is a serial number within the transaction to order the messages correctly. The default maximum message size is 50,000 bytes, but you can use the Message Agent -l option to change this setting.

Sending messages

The **log_sent** column holds the local transaction log offset for the latest message sent to the subscriber. When the Message Agent sends a message, it sets the **log_sent** value to the offset of the last COMMIT in the message. Once the message has been received and applied at the subscribed database, confirmation is

sent back to the publisher. When the publisher Message Agent receives the confirmation, it sets the **confirm_sent** column for that subscriber with the local transaction log offset. Both **log_sent** and **confirm_sent** are offsets in the local database transaction log, and **confirm_sent** cannot be a later offset than **log_sent**.

Receiving messages

When the Message Agent at a subscriber database receives and applies a replication update, it updates the **log_received** column with the offset of the last COMMIT in the message. The **log_received** column at any subscriber database therefore contains a transaction log offset in the publisher database's transaction log. After the operations have been received and applied, the Message Agent sends confirmation back to the publisher database and also sets the **confirm_received** value in the local SYSREMOTEUSER table. The **confirm_received** column at any subscriber database contains a transaction log offset in the publisher database's transaction log.

Subscriptions are two-way

SQL Remote subscriptions are two-way operations: each remote database is a subscriber to publications of the consolidated database and the consolidated database subscribes to a matching publication from each remote database. Therefore, the **remoteuser** SQL Remote system tables at the consolidated and remote database hold complementary information.

The Message Agent applies transactions and updates the **log_received** value atomically. If a message contains several transactions, and a failure occurs while a message is being applied, the **log_received** value corresponds exactly to what has been applied and committed.

Resending messages

The **remoteuser** SQL Remote table contains two other columns that handle resending messages. The **resend_count** and **receive_count** columns are retry counts that are incremented when messages get lost or deleted for some reason.

In general, the **log_send** column has the same value as the **log_sent** column. However, if the **log_send** has a value that is greater than **log_sent**, the Message Agent sends messages to the subscriber immediately on its next run.

Handling lost or corrupt messages

When messages are received at a subscriber database, the Message Agent applies them in the correct order (determined from the log offsets) and sends confirmation to the publisher. If a message is missing, the Message Agent increments the local value of **receive_count**, and requests that it be resent. Other messages present or en route are not applied.

The request from a subscriber to resend a message increments the **resend_count** value at the publisher database, and also sets the publisher's **log_send** value to the value of **confirm_sent**. This resetting of the **log_send** value causes operations to be resent.

Users cannot reset log_send

The **log_send** value cannot be reset by a user, as it is in a system table.

Message identification

Each message is identified by three values:

- ◆ Its **resend_count**.
- ◆ The transaction log offset of the last COMMIT in the previous message.
- ◆ A serial number within transactions, for transactions that span messages.

Messages with a **resend_count** value smaller than **rereceive_count** are not applied; they are deleted. This ensures that operations are not applied more than once.

CHAPTER 7

Administering SQL Remote

Contents

Running the Message Agent	124
Error reporting and handling	126
Transaction log and backup management	130
Using passthrough mode	141

About this chapter


This chapter details setup and management issues for SQL Remote administrators. SQL Remote uses SQL Anywhere as the consolidated database.

Running the Message Agent

This section describes how to run the Message Agent.

Starting the Message Agent

The Message Agent has a set of options that control its behavior. The only option that is required for the Message Agent to run is the connection parameters option (-c).


 For more information on connection parameters, see [“Connection parameters” \[SQL Anywhere Server - Database Administration\]](#).

Verbose keyword	Short form	Argument
DatabaseFile	DBF	string
DatabaseName	DBN	string
DatabaseSwitches	DBS	string
EngineName	ENG	string
Password	PWD	string
Start	Start	string
Userid	UID	string

Running the Message Agent as a service

If you are running the Message Agent in continuous mode (not batch mode) you may wish to keep the Message Agent running all the time that the server is running.

You can do this by running the Message Agent as a Windows **service**. A service can be configured to keep running even when the current user logs out, and to start as soon as the operating system is started.

 For a full description of running programs as services, see [“Running the Database Server” \[SQL Anywhere Server - Database Administration\]](#).

The Message Agent and replication security

In the tutorials in the previous chapter, the Message Agent was run using a user ID with DBA permissions. The operations in the messages are carried out from the user ID specified in the Message Agent connection string; by using the user ID **DBA**, you can be sure that the user has permissions to make all the changes.

In many situations, distributing the DBA user ID and password to all remote database users is an unacceptable practice for security and data privacy reasons. SQL Remote provides a solution that enables the Message

Agent to have full access to the database in order to make any changes contained in the messages without creating security problems.

A special permission, REMOTE DBA, has the following properties:

- ◆ **No distinct permissions when not connected from the Message Agent** A user ID granted REMOTE DBA authority has no extra privileges on any connection apart from the Message Agent. Therefore, even if the user ID and password for a REMOTE DBA user is widely distributed, there is no security problem. As long as the user ID has no permissions beyond CONNECT granted on the database, no one can use this user ID to access data in the database.
- ◆ **Full DBA permissions from the Message Agent** When connecting from the Message Agent, a user ID with REMOTE DBA authority has full DBA permissions on the database.

Using REMOTE DBA permission

A suggested practice is to grant REMOTE DBA authority at the consolidated database to the publisher and to each remote user. When the remote database is extracted, the remote user becomes the publisher of the remote database, and is granted the same permissions they were granted on the consolidated database, including the REMOTE DBA authority which enables them to use this user ID in the Message Agent connection string. Adopting this procedure means that there are no extra user IDs to administer, and each remote user needs to know only one user ID to connect to the database, whether from the Message Agent (which then has full DBA authority) or from any other client application (in which case the REMOTE DBA authority grants them no extra permissions).

Granting REMOTE DBA permission

You can grant REMOTE DBA permissions to a user ID named **dbremote** as follows:

```
GRANT REMOTE DBA
TO dbremote
IDENTIFIED BY dbremote
```

In SQL Anywhere, you can add the REMOTE DBA authority to a remote user by checking the appropriate option on the Authorities tab of the remote user's property sheet.

Error reporting and handling

This section describes how errors are reported and handled by the Message Agent.

Default error handling

The default action taken by the Message Agent when an error occurs is to record the fact in its log output. The Message Agent sends log output to a window or a log file recording its operation. By default, log output is sent to the window only; the `-o` option sends output to a log file as well.

The Message Agent may print more information in the output log than in the window. The Message Agent log includes the following:

- ◆ Listing of messages applied.
- ◆ Listing of failed SQL statements.
- ◆ Listing of other errors.

UPDATE conflicts are not errors

UPDATE conflicts are not errors, and so are not reported in the Message Agent output.

☞ For more information on the log file, see [“Message Agent” on page 148](#).

Ignoring errors

There may be exceptional cases where you wish to allow an error encountered by the Message Agent when applying SQL statements to go unreported. This may arise when you know the conditions under which the error occurs and are sure that it does not produce inconsistent data and that its consequences can safely be ignored.

To allow errors to go unreported, you can create a BEFORE trigger on the action that causes the known error. The trigger should signal the `REMOTE_STATEMENT_FAILED` SQLSTATE (5RW09) or `SQLCODE` (-288) value.

For example, if you wish to quietly fail INSERT statements on a table that fail because of a missing referenced column, you could create a BEFORE INSERT trigger that signals the `REMOTE_STATEMENT_FAILED` SQLSTATE when the referenced column does not exist. The INSERT statement fails, but the failure is not reported in the Message Agent log.

Implementing error handling procedures

SQL Remote allows you to carry out some other process in addition to logging a message if an error occurs. The `replication_error` database option allows you to specify a stored procedure to be called by the Message Agent when an error occurs. By default no procedure is called.

The procedure must have a single argument of type CHAR, VARCHAR, or LONG VARCHAR. The procedure is called twice: once with the error message and once with the SQL statement that causes the error.

While the option allows you to track and monitor errors in replication, you must still design them out of your setup: this option is not intended to resolve such errors.

For example, the procedure could insert the errors into a table with the current time and remote user ID, and this information can then replicate back to the consolidated database. An application at the consolidated database can create a report or send email to an administrator when errors show up.

☞ For information on setting the `replication_error` option, see [“SQL Remote options” on page 163](#).

Example: emailing notification of errors

You may wish to receive some notification at the consolidated database when the Message Agent encounters errors. This section demonstrates a method to send Email messages to an administrator when an error occurs.

A stored procedure

The stored procedure for this example is called `sp_LogReplicationError`, and is owned by the user `cons`. To cause this procedure to be called in the event of an error, set the `replication_error` database option using Interactive SQL or Sybase Central:

```
SET OPTION PUBLIC.replication_error =
    'cons.sp_LogReplicationError'
```

The following stored procedure implements this notification:

```
CREATE PROCEDURE cons.sp_LogReplicationError
    (IN error_text LONG VARCHAR)
BEGIN
    DECLARE current_remote_user CHAR(255);
    SET current_remote_user = CURRENT REMOTE USER;

    // Log the error
    INSERT INTO cons.replication_audit
        ( remoteuser, errormsg)
    VALUES
        ( current_remote_user, error_text);
    COMMIT WORK;

    //Now notify the DBA by email that an error has occurred
    // on the consolidated database. The email should contain the error
    // strings that the Message Agent is passing to the procedure.
    IF CURRENT PUBLISHER = 'cons' THEN
        CALL sp_notify_DBA( error_text );
    END IF
END;
```

The stored procedure calls another stored procedure to manage the sending of Email:

```
CREATE PROCEDURE sp_notify_DBA( in msg long varchar)
BEGIN
    DECLARE rc INTEGER;
    rc=call xp_startmail( mail_user='davidf' );
    //If successful logon to mail
    IF rc=0 THEN
        rc=call xp_sendmail(
```

```

        recipient='Doe, John; John, Elton',
        subject='SQL Remote Error',
        "message"=msg);
//If mail sent successfully, stop
    IF rc=0 THEN
        call xp_stopmail()
    END IF
END IF
END;

```

An audit table

An audit table could be defined as follows:

```

CREATE TABLE replication_audit (
    id          INTEGER DEFAULT AUTOINCREMENT,
    pub         CHAR(30) DEFAULT CURRENT PUBLISHER,
    remoteuser  CHAR(30),
    errormsg    LONG VARCHAR,
    timestamp   DATETIME DEFAULT CURRENT TIMESTAMP,
    PRIMARY KEY (id,pub)
);

```

The columns have the following meaning:

Column	Description
pub	Current publisher of the database (lets you know at what database it was inserted)
remoteuser	Remote user applying the message (lets you know what database it came from)
errormsg	Error message passed to the replication_error procedure

Here is a sample insert into the table from the above error:

```

INSERT INTO cons.replication_audit
    ( id,
      pub,
      remoteuser,
      errormsg,
      "timestamp")
VALUES
    ( 1,
      'cons',
      'sales',
      'primary key for table 'reptable' is not unique (-193)',
      '1997/apr/21 16:03:13.836')
COMMIT WORK

```

Since SQL Anywhere supports calling external DLLs from stored procedures you can also design a paging system, instead of using Email.

An example of an error

For example, if a row is inserted at the consolidated using the same primary key as one inserted at the remote, the Message Agent displays the following errors:

Received message from "cons" (0-0000000000-0)

SQL statement failed: (-193) primary key for table 'reptable' is not unique

```
INSERT INTO cons.reptable( id,text,last_contact )
```

```
VALUES (2,'dave','1997/apr/21 16:02:38.325')
```

```
COMMIT WORK
```

The messages that arrived in Doe, John and Elton, John's email each had a subject of SQL Remote Error:

primary key for table 'reptable' is not unique (-193)

```
INSERT INTO cons.reptable( id,text,last_contact ) VALUES
```

```
(2,'dave','1997/apr/21 16:02:52.605')
```

Transaction log and backup management

The importance of good backup practices

Replication depends on access to operations in the transaction log, and access to old transaction logs is sometimes required. This section describes how to set up backup procedures at the consolidated and remote databases to ensure proper access to old transaction logs.

It is crucial to have good backup practices at SQL Remote consolidated database sites. A lost transaction log could easily mean having to re-extract remote users. At the consolidated database site, a transaction log mirror is recommended.

☞ For information on transaction log mirrors and other backup procedure information, see [“Backup and Data Recovery” \[SQL Anywhere Server - Database Administration\]](#).

Ensuring access to old transactions

All transaction logs must be guaranteed available until they are no longer needed by the replication system.

In many setups, users of remote databases may receive updates from the office server every day or so. If some messages get lost or deleted, and have to be resent by the message-tracking system, it is possible that changes made several days ago will be required. If a remote user takes a vacation, and messages have been lost in the meantime, changes weeks old may be required. If the transaction log is backed up daily, the log with the changes will no longer be running on the server.

Because the transaction log continually grows in size, space can become a concern. You can use an event handler on transaction log size to rename the log when it reaches a given size. Then you can use the `delete_old_logs` option to clean up log files that are no longer needed.

☞ For more information about controlling transaction log size, see the [“BACKUP statement” \[SQL Anywhere Server - SQL Reference\]](#).

Setting the transaction log directory

When the Message Agent needs to scan transaction logs other than the current log, it looks through all the transaction log files kept in a designated **transaction log directory**. A setting on the Message Agent command line tells the Message Agent which directory this is.

Example

For example, the following command line tells the Message Agent to look in the directory `e:\archive` to find old transaction logs. The command must be entered all on one line.

```
dbremote -c "eng=server_name;uid=DBA;pwd=sql" e:\archive
```

Log names are not important

The Message Agent opens all the files in the transaction log directory to determine which files are logs, so the actual names of the log files are not important.

This section describes how you can set up a backup procedure to ensure that such a directory is kept in proper shape.

Backup utility options

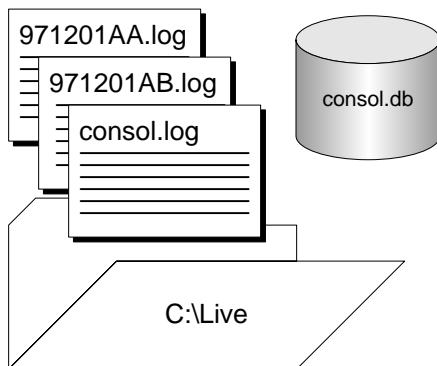
The SQL Anywhere backup utility has several options, accessible through Sybase Central wizard selections or through `dbbackup` options, that control its behavior.

This section describes two approaches to using the backup utility in SQL Remote consolidated database backups. Backups must ensure that a set of transaction logs suitable for use by the Message Agent is always available.

Using the live directory as the transaction log directory

It is recommended that you use the option to rename and restart the transaction log when backing up the consolidated database and remote database transaction logs. For the `dbbackup` utility, this is the `-r` option.

The figure below illustrates a database named `consol.db`, with a transaction log named `consol.log` in the same directory. For the sake of simplicity, this example assumes that the log is in the same directory as the database, although this would not generally be safe practice in a production environment. The directory is named `c:\live`.



A backup command line

The following command line backs up the database using the rename and restart option:

```
dbbackup -r -c "uid=DBA;pwd=sql" c:\archive
```

The connection string options would be different for each database.

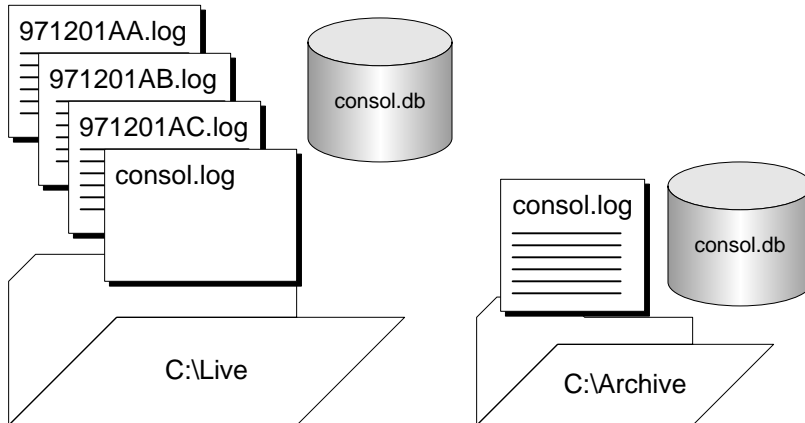
Effects of the backup

If you back up the transaction log to a directory `c:\archive` using the rename and restart option, the Backup utility carries out the following tasks:

1. Backs up the transaction log file, creating a backup file `c:\archive\consol.log`.

2. Renames the existing transaction log file to *971201xx.log*, where *xx* are sequential characters ranging from *AA* to *ZZ*.
3. Starts a new transaction log, as *consol.log*.

After several backups, the live directory contains a set of sequential transaction logs.



A Message Agent command line

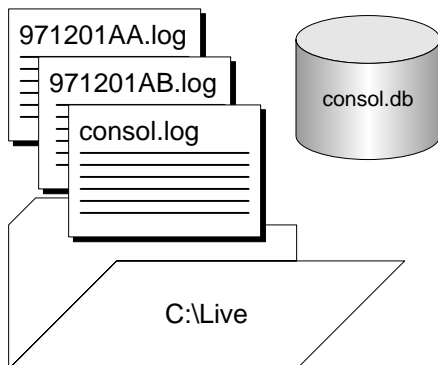
You can run the Message Agent with access to these log files using the following command line:

```
dbremote -c "dbn=hq;..." c:\live
```

Using the backup directory as the transaction log directory

An alternative procedure is to use the backup directory as the transaction log directory.

Again, the figure below illustrates a database named *consol.db*, with a transaction log named *consol.log* in the same directory. For the sake of simplicity, this example assumes that the log is in the same directory as the database, although this would not generally be safe practice in a production environment. The directory is named *c:\live*.



A backup command line

The following command line backs up the database using the rename and restart option, and also uses an option to rename the transaction log backup file:

```
dbbackup -r -n -c "uid=DBA;pwd=sql" c:\archive
```

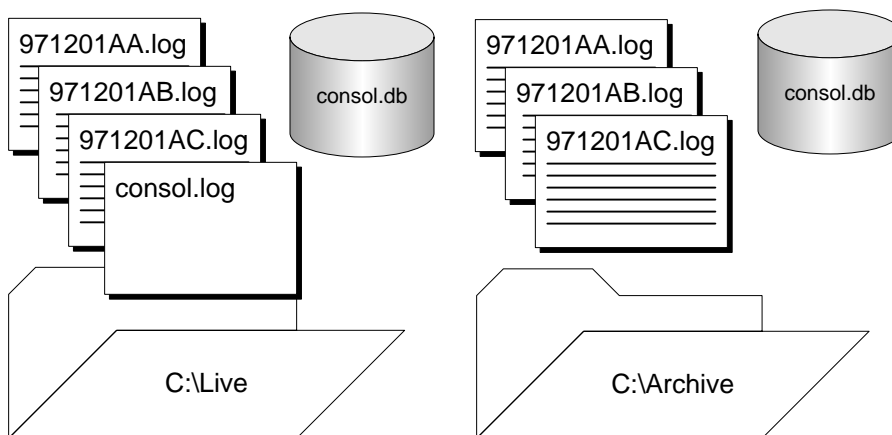
The connection string options would be different for each database.

Effects of the backup

If you back up the transaction log to a directory `c:\archive` using the rename and restart option and the log renaming option, the Backup utility carries out the following tasks:

1. Renames the existing transaction log file to `971201xx.log`, where `xx` are sequential characters ranging from `AA` to `ZZ`.
2. Backs up the transaction log file to the backup directory, creating a backup file named `971201xx.log`
3. Starts a new transaction log, as `consol.log`.

After several backups, the live directory and also the archive directory contain a set of sequential transaction logs.



A Message Agent command line

You can run the Message Agent with access to these log files using the following command line:

```
dbremote -c "dbn=hq;..." c:\archive
```

Old log names different before 8.0.1

Prior to release 8.0.1 of Adaptive Server Anywhere, the old log files were named `yymmdd01.log`, `yymmdd02.log`, and so on. The name change was introduced to allow more old logs to be stored. As the Message Agent scans all the files in the specified directory, regardless of their names, the name change should not affect existing applications.

Managing old transaction logs

All transaction logs must be guaranteed available until they are no longer needed by the replication system: at that point, they can be discarded.

The replication system no longer needs the logs when all remote databases have received and successfully applied the messages contained in the log files. Remote databases confirm the successful receipt of messages from the consolidated database, and the confirmation sets a value in the consolidated database SQL Remote tables (see “[The message tracking system](#)” on page 120). The old transaction logs at the consolidated database are no longer needed by SQL Remote when this receipt confirmation has been received from all remote databases.

Using the delete_old_logs option

You can use the delete_old_logs database option at the consolidated database to manage old transaction logs automatically.

The delete_old_logs database option is set by default to Off. If it is set to on, the old transaction logs are deleted automatically by the Message Agent when they are no longer needed. A log is no longer needed when all subscribers have confirmed receiving all changes recorded in that log file.

You can set the delete_old_logs option either for the PUBLIC group or just for the user contained in the Message Agent connection string.

Example

The following statement sets the public delete_old_logs option to delete logs that were created more than 10 days ago:

```
SET OPTION PUBLIC.delete_old_logs = '10 days'
```

Recovery from database media failure for consolidated databases

This section describes how to recover from a media failure on the database device at the consolidated database.

The procedures to follow are easiest to describe if there is only one transaction log file. While this might not be common for consolidated databases, it is described first, followed by a more common, but complicated, situation with a set of transaction log files.

Recovery with a single transaction log

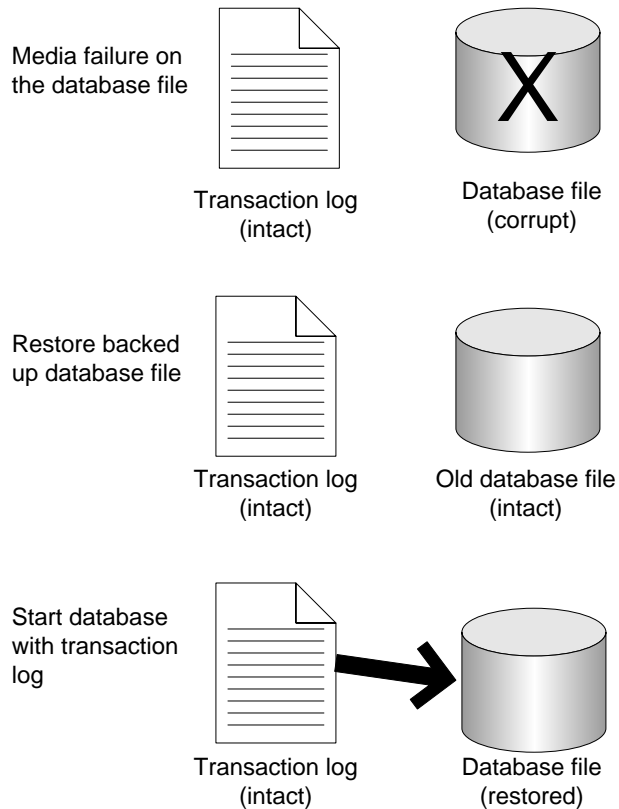
The following steps assume that there is a single transaction log file, which has existed since the database was created. They also assume that previous backups of the database file have been made and are available, for example on tape.

◆ To recover the database

1. Make a copy of the database and log file.
2. Restore the database (.db) file, *not* the log file, from tape into a temporary directory.

3. Start the database using the existing transaction log and the `-a` option, to apply the transactions and bring the database file up to date.
4. Start the database in your normal way.

Any new activity is appended to the current transaction log.



Example

This example illustrates recovery using a mirrored transaction log.

Suppose you have a consolidated database file named *consol.db* in a directory *c:\dbdir*, and a transaction log file *c:\logdir\consol.log* that is mirrored to *d:\mirmdir\consol.mlg*.

◆ To recover from media failure on the C drive

1. Back up the mirrored transaction log *d:\mirmdir\consol.mlg*.
2. Replace the failed hardware and re-install all affected software.
3. Create a temporary directory to perform the recovery in (for example, *c:\recover*).
4. Restore the most recent backup of the database file, *consol.db*, to *c:\recover\consol.db*.

5. Copy the mirror transaction log, *d:\mirdir\consol.mlg*, to the recovery directory with a *.log* extension, giving *c:\recover\consol.log*.
6. Start the database using the following command:

```
dbeng10 -a c:\recover\consol.log C:\recover\consol.db
```
7. Shut down the database server.
8. Back up the recovered database and transaction log from *c:\recover*.
9. Copy the files from *c:\recover* to the appropriate production directories:
 - ◆ Copy *c:\recover\consol.db* to *c:\dbdir\consol.db*.
 - ◆ Copy *c:\recover\consol.log* to *c:\logdir\consol.log*, and to *d:\mirdir\consol.mlg*.
10. Restart your system normally.

Recovery with multiple transaction logs

If you have a set of transaction logs, the procedure is different than if you only have an online transaction log. You can choose to apply transaction logs individually using the *-a* option, or you can allow the database server to determine the correct order of the transaction logs and apply them all using the *-ad* option.

Applying transaction logs individually

The following steps describe how to apply each transaction log to the database individually when recovering. The procedure assumes that previous backups of the database file have been made and are available, for example on tape.

◆ To recover the database using the *-a* option

1. Make a copy of the database and log file.
2. Restore the database (*.db*) file, *not* the log file, from tape into a temporary directory.
3. In the temporary directory, start the database, applying the old logs using the *-a* option, applying the named transaction logs in the correct order.
4. Start the database using the current transaction log and the *-a* option to apply the transactions and bring the database file up to date.
5. Start the database in your normal way.

Any new activity is appended to the current transaction log.

Example

Suppose you have a consolidated database file named *c:\dbdir\cons.db*. The transaction log file *c:\dbdir\cons.log* is mirrored to *d:\mirdir\cons.mlg*.

Assume that you perform full backups weekly, and you perform incremental backups daily using the following command:


```
dbbackup -c "uid=DBA;pwd=sql" -r -t e:\backdir
```

This command backs up the transaction log *cons.log* to the directory *e:\backdir*. The transaction log file is then renamed to *datexx.log*, where *date* is the current date and *xx* is the next set of letters in sequence, and a new transaction log is started. The directory *e:\backdir* is then backed up using a third-party utility.

In this scenario you would be running the Message Agent with the optional directory to point to the renamed transaction log files. The Message Agent command line would be

```
dbremote -c "uid=DBA;pwd=sql" c:\dbdir
```

On the third day following the weekly backup the database file gets corrupted because of a bad disk block.

◆ To recover from media failure on the C drive

1. Back up the mirrored transaction log *d:\mirdir\cons.mlg*.
2. Create a temporary directory to perform the recovery in. In this example, the directory is called *c:\recover*.
3. Restore the most recent backup of the database file, *cons.db* to *c:\recover\cons.db*.
4. Apply the renamed transaction logs in order, as follows

```
dbeng10 -a c:\dbdir\date00.log c:\recover\cons.db
dbeng10 -a c:\dbdir\date01.log c:\recover\cons.db
```

5. Copy the current transaction log, *c:\dbdir\cons.log* to the recovery directory, giving *c:\recover\cons.log*.
6. Start the database using the following command:

```
dbeng10 c:\recover\cons.db
```

7. Shut down the database server.
8. Back up the recovered database and transaction log from *c:\recover*.
9. Copy the files from *c:\recover* to the appropriate production directories:
 - ◆ Copy *c:\recover\cons.db* to *c:\dbdir\cons.db*.
 - ◆ Copy *c:\recover\cons.log* to *c:\dbdir\cons.log*, and to *d:\mirdir\cons.mlg*.
10. Restart your system as normal.

Applying multiple transaction logs

The following steps describe how to start the database server so it automatically applies all the transaction logs to the database in the correct order. When you specify the *-ad* option, the database server looks in the specified directory for the transaction logs for the database. It then determines the correct order to apply the logs based on the log offsets.

The following procedure assumes that previous backups of the database file have been made and are available, for example on tape.

◆ **To recover the database using the -ad option**

1. Make a copy of the database and log file.
2. Restore the database (.db) file, *not* the log file, from tape into a temporary directory.
3. In the temporary directory, start the database, applying the transaction logs using the -ad option.
4. Start the database in your normal way.

Any new activity is appended to the current transaction log.

Example

Suppose you have a consolidated database file named *c:\dbdir\cons.db*. The transaction log file *c:\dbdir\cons.log* is mirrored to *d:\mirdir\cons.mlg*.

Assume that you perform full backups weekly using the following command:

```
dbbackup -c "uid=DBA;pwd=sql" -r e:\backdir
```

Assume that you also perform incremental backups daily using the following command:

```
dbbackup -c "uid=DBA;pwd=sql" -r -t e:\backdir
```

This command backs up the transaction log *cons.log* to the directory *e:\backdir*. The transaction log file is then renamed to *datexx.log*, where *date* is the current date and *xx* is the next set of letters in sequence, and a new transaction log is started. The directory *e:\backdir* is then backed up using a third-party utility.

In this scenario you would be running the Message Agent with the optional directory to point to the renamed transaction log files. The Message Agent command line would be

```
dbremote -c "uid=DBA;pwd=sql" c:\dbdir
```

On the third day following the weekly backup, the database file gets corrupted because of a bad disk block.

◆ **To recover from media failure on the C drive**

1. Replace the *c:* drive.
2. Back up the mirrored transaction log *d:\mirdir\cons.mlg*.
3. Create a temporary directory to perform the recovery in. In this example, it is called *c:\recover*.
4. Restore the most recent backup of the database file, *cons.db* to *c:\recover\cons.db*.
5. Copy the backed up transaction logs to *c:\dbdir*.
6. Apply the renamed transaction logs:

```
dbeng10 c:\recover\cons.db -ad c:\dbdir
```

7. Copy the current transaction log, *c:\dbdir\cons.log* to the recovery directory, giving *c:\recover\cons.log*.
8. Start the database using the following command:

```
dbeng10 c:\recover\cons.db
```

9. Shut down the database server.
10. Back up the recovered database and transaction log from *c:\recover*.
11. Copy the files from *c:\recover* to the appropriate production directories:
 - ◆ Copy *c:\recover\cons.db* to *c:\dbdir\cons.db*.
 - ◆ Copy *c:\recover\cons.log* to *c:\dbdir\cons.log*, and to *d:\mirdir\cons.mlg*.
12. Restart your system as normal.

Backup procedures at remote databases

Backup procedures are not as crucial at remote databases as at the consolidated database. You may choose to rely on replication to the consolidated database as a data backup method. In the event of a media failure, the remote database would have to be re-extracted from the consolidated database, and any operations that have not been replicated would be lost. (You could use the log translation utility to attempt to recover lost operations.)

Even if you do choose to rely on replication to protect remote database data, backups still need to be done periodically at remote databases to prevent the transaction log from growing too large. You should use the same option (rename and restart the log) as at the consolidated database, running the Message Agent so that it has access to the renamed log files. If you set the `delete_old_logs` option on at the remote database, the old log files will be deleted automatically by the Message Agent when they are no longer needed.

Automatic transaction log renaming

You can use the `-x` Message Agent option to eliminate the need to rename the transaction log on the remote computer when the database server is shut down. The `-x` option renames transaction log after it has been scanned for outgoing messages.

Upgrading consolidated databases

This section describes issues in upgrading a consolidated database in a SQL Remote environment. The same considerations apply to SQL Anywhere databases that are primary sites in a Sybase Replication Server installation.

For information about upgrading SQL Remote consolidated databases to version 10, see [“Upgrading SQL Remote” \[SQL Anywhere 10 - Changes and Upgrading\]](#).

Unloading and reloading a database participating in replication

If a database is participating in replication, particular care needs to be taken if you want to unload and reload the databases.

☞ For instructions for unloading and reloading a database in “[Upgrading SQL Remote](#)” [*SQL Anywhere 10 - Changes and Upgrading*]. The instructions in that section apply to databases involved in SQL Remote replication.

Replication is based on the transaction log. When a database is unloaded and reloaded, the old transaction log is no longer available. For this reason, good backup practices are especially important when participating in replication.

☞ For the steps involved in unloading a databases involved in replication, see “[Unloading and reloading a database participating in replication](#)” on page 139.

Using passthrough mode

The publisher of the consolidated database can directly intervene at remote sites using a passthrough mode, which enables standard SQL statements to be passed through to a remote site. By default, passthrough mode statements are executed at the local (consolidated) database as well, but an optional keyword prevents the statements from being executed locally.

Caution

Always test your passthrough operations on a test database with a remote database subscribed. Never run untested passthrough scripts against a production database.

Starting and stopping passthrough

Passthrough mode is started and stopped using the PASSTHROUGH statement. Any statement entered between the starting PASSTHROUGH statement and the PASSTHROUGH STOP statement which terminates passthrough mode is checked for syntax errors, executed at the current database, and also passed to the identified subscriber and executed at the subscriber database. The statements between a starting and stopping passthrough statement are called the **passthrough session**.

The following statement starts a passthrough session which passes the statements to a list of two named subscribers, without being executed at the local database:

```
PASSTHROUGH ONLY
FOR userid_1, userid_2;
```

Directing passthrough statements

The following statement starts a passthrough session that passes the statements to all subscribers to the specified publication:

```
PASSTHROUGH ONLY
FOR SUBSCRIPTION TO [owner].pubname [ ( string ) ] ;
```

Passthrough mode is additive. In the following example, **statement_1** is sent to **user_1**, and **statement_2** is sent to both **user_1** and **user_2**.

```
PASSTHROUGH ONLY FOR user_1 ;
statement_1 ;
PASSTHROUGH ONLY FOR user_2 ;
statement_2 ;
```

The following statement terminates a passthrough session:

```
PASSTHROUGH STOP ;
```

PASSTHROUGH STOP terminates passthrough mode for all remote users.

Order of application of passthrough statements

Passthrough statements are replicated in sequence with normal replication messages, in the order in which the statements are recorded in the log.

Passthrough is commonly used to send data definition language statements. In this case, replicated DML statements use the *before* schema before the passthrough and the *after* schema following the passthrough.

Notes on using passthrough mode

- ◆ You should always test your passthrough operations on a test database with a remote database subscribed. You should never run untested passthrough scripts against a production database.
- ◆ You should always qualify object names with the owner name. PASSTHROUGH statements are not executed at remote databases from the same user ID. Consequently, object names without the owner name qualifier may not be resolved correctly.

Uses and limitations of passthrough mode

Passthrough mode is a powerful tool, and should be used with care. Some statements, especially data definition statements, could cause a running SQL Remote setup to come tumbling down. SQL Remote relies on each database in a setup having the same objects: if a table is altered at some sites but not at others, attempts to replicate data changes will fail.

Also, it is important to remember that in the default setting passthrough mode also executes statements at the local database. To send statements to a remote database without executing them locally you must supply the ONLY keyword. When a passthrough session contains calls to stored procedures, the procedures must exist in the server that is issuing the passthrough commands, even if they are not being executed locally at the server. The following set of statements drops a table not only at a remote database, but also at the consolidated database.

```
-- Drop a table at the remote database
-- and at the local database
PASSTHROUGH TO Joe_Remote ;
DROP TABLE CrucialData ;
PASSTHROUGH STOP ;
```

The syntax to drop a table at the remote database only is as follows:

```
-- Drop a table at the remote database only
PASSTHROUGH ONLY TO Joe_Remote ;
DROP TABLE CrucialData ;
PASSTHROUGH STOP ;
```

The following are tasks that can be carried out on a running SQL Remote setup:

- ◆ Add new users.
- ◆ Resynchronize users.
- ◆ Drop users from the setup.
- ◆ Change the address, message type, or frequency for a remote user.
- ◆ Add a column to a table.

Many other schema changes are likely to cause serious problems if executed on a running SQL Remote setup.

Passthrough works on only one level of a hierarchy

In a multi-tier SQL Remote installation, it becomes important that passthrough statements work on the level of databases immediately beneath the current level. In a multi-tier installation, passthrough statements must be entered at each consolidated database, for the level beneath it.

Operations not replicated in passthrough mode

There are special considerations for some statements in passthrough mode.

Calling procedures

When a stored procedure is called in passthrough mode using a CALL or EXEC statement, the CALL statement itself is replicated and none of the statements inside the procedure are replicated. It is assumed that the procedure on the replicate side has the correct effect.

Control of flow statements and cursor operations

Control-flow statements such as IF and LOOP, as well as any cursor operations, are not replicated in passthrough mode. Any statements within the loop or control structure *are* replicated.

Operations on cursors are not replicated. Inserting rows through a cursor, updating rows in a cursor, or deleting rows through a cursor are not replicated in passthrough mode.

Static embedded SQL SET OPTION statements are not replicated. The following statement is not replicated in passthrough mode:

```
EXEC SQL SET OPTION . . .
```

However, the following dynamic SQL statement is replicated:

```
EXEC SQL EXECUTE IMMEDIATE "SET OPTION . . . "
```

Batches

Batch statements (a group of statements surrounded with a BEGIN and END) are not replicated in passthrough mode. You receive an error message if you try to use batch statements in passthrough mode.

Part IV. Reference

This part presents reference material for SQL Remote.

CHAPTER 8

Utilities and Options Reference

Contents

Message Agent	148
Database Extraction utility	156
SQL Remote options	163
SQL Remote event-hook procedures	167

About this chapter

This chapter provides reference material for the SQL Remote utilities and SQL Remote database options.

It also describes client event-hook stored procedures, which can be used to customize the replication process.

Message Agent

Purpose

To send and apply SQL Remote messages, and to maintain the message tracking system to ensure message delivery.

Syntax

dbremote [*options*] [*directory*]

Options

Option	Description
@ <i>data</i>	Read options from the specified environment variable or configuration file.
-a	Do not apply received transactions.
-b	Run in batch mode.
-c " <i>keyword=value; ...</i> "	Supply database connection parameters.
-dl	Display log messages on screen.
-ek <i>key</i>	Specify encryption key.
-ep	Prompt for encryption key.
-g <i>n</i>	Group transactions consisting of less than <i>n</i> operations.
-k	Close window on completion.
-l <i>length</i>	Maximum message length.
-m <i>size</i>	Maximum amount of memory used for building messages.
-ml <i>directory</i>	Specify the location of offline mirror logs.
-o <i>file</i>	Output messages to file.
-os <i>size</i>	Maximum file size for logging output messages.
-ot <i>file</i>	Truncate file and log output messages.
-p	Do not purge messages.
-q	Run with minimized window.
-r	Receive messages.
-rd <i>minutes</i>	Polling frequency for incoming messages.

Option	Description
-ro <i>filename</i>	Log remote output to file.
-rp <i>number</i>	Number of receive polls before message is assumed lost.
-rt <i>filename</i>	Truncate, and log remote output to file.
-ru <i>time</i>	Waiting period to re-scan log on receipt of a resend.
-s	Send messages.
-sd <i>time</i>	Send polling period.
-t	Replicate all triggers.
-u	Process only backed up transactions.
-ud	On Unix and Linux, run as a daemon.
-ux	On Solaris and Linux, open the console window.
-v	Verbose operation.
-w <i>n</i>	Number of worker threads to apply incoming messages (Not NetWare or Windows CE).
-x [<i>size</i>]	Rename and restart the transaction log.
directory	The directory in which old transaction logs are held.

Description

The Message Agent sends and applies messages for SQL Remote replication, and maintains the message tracking system to ensure message delivery.


The name of the Message Agent executable is `dbremote`.

You can also run the Message Agent from your own application by calling into the DBTools library. For more information, see the file `dbrmt.h` in the `h` subdirectory of your SQL Remote installation directory.

The user ID in the Message Agent command must have either REMOTE DBA or DBA authority.

The optional `directory` parameter specifies a directory in which old transaction logs are held, so that the Message Agent has access to events from before the current log was started.

The Message Agent uses a number of connections to the database. For a listing, see [“Connections used by the Message Agent” on page 109](#).

 For information on REMOTE DBA authority, see [“The Message Agent and replication security” on page 124](#).

Option details

@data

Use this option to read in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used.

The environment variable may contain any set of options. For example, the first of the following pair of statements sets an environment variable holding a set of options for a database server that starts with a cache size of 4 MB, receives messages only, and connects to a database named **field** on a server named **myserver**. The **set** statement should be entered all on one line:

```
set envvar=-m 4096 -r
    -c "eng=myserver;dbn=field;uid=sa;pwd=sysadmin"
dbremote @envvar
```

The configuration file may contain line breaks, and may contain any set of options. For example, the following command file holds a set of options for a Message Agent that starts with a cache size of 4 MB, sends messages only, and connects to a database named **field** on a server named **myserver**:

```
-m 4096
-s
-c "eng=myserver;dbn=field;uid=sa;pwd=sysadmin"
```

If this configuration file is saved as *c:\config.txt*, it can be used in a command as follows:

```
dbremote @c:\config.txt
```

-a

Process the received messages (those in the inbox) without applying them to the database. Used together with **-v** (for verbose output) and **-p** (so the messages are not purged), this option can help detect problems with incoming messages. Used without **-p**, this option purges the inbox without applying the messages, which may be useful if a subscription is being restarted.

-b

Run in batch mode. In this mode, the Message Agent processes incoming messages, scans the transaction log once and processes outgoing messages, and then stops.


-c "parameter=value; ..."

Specify connection parameters. For SQL Anywhere, if this option is not specified, the environment variable **SQLCONNECT** is used.

For example, the following statement runs **dbremote** on a database file named *c:\mydata.db*, connecting with user ID **DBA** and password **sql**:

```
dbremote -c "uid=DBA;pwd=sql;dbf=c:\mydata.db"
```

The Message Agent must be run by a user with **REMOTE DBA** authority or **DBA** authority.

 For information on **REMOTE DBA** authority, see [“The Message Agent and replication security”](#) on page 124.

The Message Agent supports the full range of SQL Anywhere connection parameters.

-dl

Display messages in the Message Agent window or at the command prompt and also in the log file if specified.

Specify encryption key (-ek)

This option allows you to specify the encryption key for strongly encrypted databases directly at the command prompt. If you have a strongly encrypted database, you must provide the encryption key to use the database or transaction log in any way, including offline transaction logs. For strongly encrypted databases, you must specify either `-ek` or `-ep`, but not both. The command will fail if you do not specify a key for a strongly encrypted database.

Prompt for encryption key (-ep)

This option allows you to specify that you want to be prompted for the encryption key. This option causes a dialog box to appear, in which you enter the encryption key. It provides an extra measure of security by never allowing the encryption key to be seen in clear text. For strongly encrypted databases, you must specify either `-ek` or `-ep`, but not both. The command will fail if you do not specify a key for a strongly encrypted database.

```
"language_name,charset_name[,sort_order]"
```

By default, the Message Agent uses the default locale, which is defined in the file `sybase\locales\locales.dat`.

-g

n Instructs the Message Agent to group transactions containing less than *n* operations together with transactions that follow. The default is twenty operations. Increasing the value of *n* can speed up processing of incoming messages, by doing less commits. However, it can also cause deadlock and blocking by increasing the size of transactions.

If neither `-r` nor `-s` is specified, the Message Agent executes all three phases. Otherwise, only the indicated phases are executed.

-k

Close window on completion when used together with the `-o` parameter.

-l length

Specifies the maximum length of each message to be sent, in bytes. Longer transactions are split into more than one message. The default is 50000 bytes and the minimum length is 10000.

Caution

The maximum message length must be the same at all sites in an installation.

For platforms with restricted memory allocation, the value must be less than the maximum memory allocation of the operating system.

-m size

Specifies a maximum amount of memory to be used by the Message Agent for building messages and caching incoming messages. The allowed size can be specified as *n* (in bytes), *nK*, or *nM*. The default is 2048 KB (2 MB).

When all remote databases are receiving unique subsets of the operations being replicated, a separate message for each remote database is built up concurrently. Only one message is built for a group of remote users that are receiving the same operations. When the memory being used exceeds the `-m` value, messages are sent before reaching their maximum size (as specified by the `-l` option).

When messages arrive, they are stored in memory by the Message Agent until they are applied. This caching of messages prevents rereading of messages that are out of order from the message system, which may lower performance on large installations. When the memory usage specified using the `-m` option is exceeded, messages are flushed in a least-recently used fashion.

-ml

This option makes it possible for `dbremote` to delete old mirror log files when either of the following two circumstances occur:

- ◆ the offline mirror log is located in a different directory from the mirror transaction log
- ◆ `dbremote` is run on a different machine from the remote database server

In a typical setup, the active mirror log and renamed mirror transaction logs are located in the same directory, and `dbremote` is run on the same machine as the remote database, so this option is not required and old mirror log files are automatically deleted. Transaction logs in this directory are only affected if the `delete_old_logs` database option is set to `On` or `DELAY`.

-o

Append output to a log file. Default is to send output to the screen.

-os

Specifies the maximum file size for logging output messages. The allowed size can be specified as *n* (bytes), *nK* (KB), or *nM* (MB). By default there is no limit, and the minimum limit is 10000 bytes.

Before SQL Remote logs output messages to a file, it checks the current file size. If the log message will make the file size exceed the specified size, SQL Remote renames the output file to `yymmddxx.dbr`, where *xx* are sequential characters ranging from `AA` to `ZZ`, and `yymmdd` represents the current year, month, and date.

If the Message Agent is running in continuous mode for a long time, this option allows you to manually delete old log files and free up disk space.

-ot

Truncate the log file and then append output messages to it. Default is to send output to the screen.

-p

Process the messages without purging them.

-q

For Windowing operating systems only, starts the Message Agent with a minimized window.

-r

Receive messages. If none of `-r`, `-I`, or `-s` is specified, the Message Agent executes all three phases. Otherwise, only the indicated phases are executed.

The Message Agent runs in continuous mode if called with `-r`. To have the Message Agent shut down after receiving messages, use the `-b` option in addition to `-r`.

-rd time

By default, the Message Agent polls for incoming messages every minute. This option (`rd` stands for **receive delay**) allows the polling frequency to be configured, which is useful when polling is expensive.

You can use a suffix of **s** after the number to indicate seconds, which may be useful if you want frequent polling. For example:

```
dbremote -rd 30s
```

polls every thirty seconds.

☞ For more information on polling, see [“Tuning incoming message polling” on page 113](#).

-ro

This option is for use at consolidated sites. When remote databases are configured to send output log information to the consolidated database, this option writes the information to a file. The option is provided to help administrators troubleshoot errors at remote sites.

☞ For more information, see [“Troubleshooting errors at remote sites” on page 110](#).

-rp

When running in continuous mode, the Message Agent polls at certain intervals for messages. After polling a set number of times (by default, one), if a message is missing, the Message Agent assumes it has got lost and requests that it be resent. On slow message systems, this can result in many unnecessary resend requests. You can set the number of polls before a resend request is issued using this option, to cut down on the number of resend requests.

☞ For more information on configuring this option, see [“Tuning incoming message polling” on page 113](#).

-rt

This option is for use at consolidated sites. It is identical to the -ro option except that the file is truncated on startup.

-ru

Control the **resend urgency**. This is the time between detection of a resend request and when the Message Agent starts fulfilling the request. Use this option to help the Message Agent collect resend requests from multiple users before rescanning the log. The time unit can be any of {s = seconds; m = minutes; h = hours; d = days }

-s

Send messages. If none of -r, -I, or -s is specified, the Message Agent executes all three phases. Otherwise, only the indicated phases are executed.

-sd time

Control the **send delay** which is the time to wait between polls for more transaction log data to send.

-t

All trigger actions are replicated. If you do use this option, you must ensure that the trigger actions are not carried out twice at remote databases, once by the trigger being fired at the remote site, and once by the explicit application of the replicated actions from the consolidated database.

To ensure that trigger actions are not carried out twice, you can wrap an IF CURRENT REMOTE USER IS NULL ... END IF statement around the body of the triggers. This option is available for SQL Anywhere only.

-u

Process only transactions that have been backed up. This option prevents the Message Agent from processing transactions since the latest backup. Using this option, outgoing transactions and confirmation of incoming transactions are not sent until they have been backed up.


This means that only transactions from renamed logs are processed.

-ud

On Unix platforms, you can run the Message Agent as a daemon by supplying the `-ud` option.

If you run the Message Agent as a daemon, you must also supply the `-o` or `-ot` option, to log output information.

If you run the Message Agent as a daemon and are using FTP or SMTP message links, you must store the message link parameters in the database, because the Message Agent does not prompt the user for these options when running as a daemon.

 For information on message link parameters, see [“Setting message type control parameters” on page 99](#).

-ux

When `-ux` is specified, `dbremote` must be able to find a usable display. If it cannot find one, for example because the `DISPLAY` environment variable is not set or because the X Windows Server is not running, `dbremote` fails to start. On Windows, the console opens automatically.

-v

Verbose output. This option displays the SQL statements contained in the messages to the screen and, if the `-o` or `-ot` option is used, to a log file.

-w n

The number of worker threads used to apply incoming messages. The default is zero, which means all messages are applied by the main (and only) thread. A value of 1 (one) would have one thread receiving messages from the message system and one thread applying messages to the database.

The `-w` option makes it possible to increase the throughput of incoming messages with hardware upgrades. Putting the consolidated database on a device that can perform many concurrent operations (a RAID array with a striped logical drive) will improve throughput of incoming messages. Multiple processors in the computer running the Message Agent could also improve throughput of incoming messages.

The `-w` option will not improve performance significantly on hardware that cannot perform many concurrent operations.

Incoming messages from a single remote database will never be applied on multiple threads. Messages from a single remote database are always applied serially in the correct order.

-x

Rename and restart the transaction log after it has been scanned for outgoing messages. In some circumstances, replicating data to a consolidated database can take the place of backing up remote databases, or renaming the transaction log when the database server is shut down. This option is available for SQL Anywhere only.

If the optional *size* qualifier is supplied, the transaction log is renamed only if it is larger than the specified size. The allowed size can be specified as *n* (in bytes), *nK*, or *nM*. The default is 0.

Message system control parameters

SQL Remote uses several registry settings to control aspects of message link behavior.

The message link control parameters are stored in the following places:

- ◆ **Windows** In the registry, at the following location:

```
\\HKEY_CURRENT_USER
  \Software
    \Sybase
      \SQL Remote
```

- ◆ **NetWare** You should create a file named *dbremote.ini* in the *sys:\system* directory to hold the FILE system directory setting.

☞ For a listing of registry settings, see the section for each message system under “[Using message types](#)” on page 96.

Database Extraction utility

You can access the remote database extraction utility in the following ways:

- ◆ From Sybase Central, for interactive use.
- ◆ From the system command prompt, using the dbxtract utility. This is useful for incorporating into batch or command files.

By default, the extraction utility runs at isolation level zero. If you are extracting a database from an active server, you should run it at isolation level 3 (see [“Extraction utility” on page 157](#)) to ensure that data in the extracted database is consistent with data on the server. Running at isolation level 3 may hamper others' turnaround time on the server because of the large number of locks required. It is recommended that you run the extraction utility when the server is not busy, or run it against a copy of the database (see [“Designing an efficient extraction procedure” on page 83](#)).

Objects owned by dbo

The extraction utility does not unload the objects created for the **dbo** user ID during database creation. Changes made to these objects, such as redefining a system procedure, are lost when the data is unloaded. Any objects created by the **dbo** user ID since the initialization of the database are unloaded by the Extraction utility, and so these objects are preserved.

Extracting a remote database in Sybase Central

Running the extraction utility from Sybase Central carries out the following tasks related to creating and synchronizing SQL Remote subscriptions:

- ◆ Creates a command file to build a remote database containing a copy of the data in a specified publication.
- ◆ Creates the necessary SQL Remote objects, such as message types, publisher and remote user IDs, publication and subscription, for the remote database to receive messages from and send messages to the consolidated database.
- ◆ Starts the subscription at both the consolidated and remote databases.

Note

Only tables for users selected in the Filter Objects by Owner dialog appear in the Extract Database wizard. If you want to view tables belonging to a particular database user, right-click the database you are unloading, choose Filter Objects by Owner from the popup menu, and then select the desired user in the resulting dialog.

◆ To extract a remote database from a running database (Sybase Central)

1. Connect to the database.
2. From the Tools menu, choose SQL Anywhere 10 ► Extract Database.

The Extract Database wizard appears.

3. Follow the instructions in the wizard.

Extraction utility

Purpose

To extract a remote SQL Anywhere database from a consolidated SQL Anywhere database.

Syntax

dbxtract [*options*] [*directory*] *subscriber*

Option	Description
@ <i>data</i>	Read in options from a configuration file. See “@ <i>data server option</i> ” [<i>SQL Anywhere Server - Database Administration</i>].
-ac " <i>keyword=value; ...</i> "	Connect to the database specified in the connect string to do the reload.
-al <i>filename</i>	Log file name for this new database.
-an <i>database</i>	Creates a database file with the same settings as the database being unloaded and automatically reloads it.
-b	Do not start subscriptions
-c " <i>keyword=value; ...</i> "	Supply database connection parameters
-d	Unload data only
-ea <i>alg</i>	Specify the encryption algorithm for the new database
-ek <i>key</i>	Specify the encryption key for the new database
-ep	Prompt for the encryption key for the new database
-f	Extract fully qualified publications
-ii	Internal unload, internal reload
-ix	Internal unload, external reload
-l <i>level</i>	Perform all extraction operations at specified isolation level
-n	Extract schema definition only
-o <i>file</i>	Output messages to file
-p <i>character</i>	Escape character
-q	Operate quietly: do not display messages or show windows. When this option is specified, -y must also be specified or the operation will fail.

Option	Description
-r <i>file</i>	Specify name of generated reload Interactive SQL command file (default " <i>reload.sql</i> ")
-u	Unordered data
-v	Verbose messages
-xf	Exclude foreign keys
-xh	Exclude procedure hooks
-xi	External unload, internal reload
-xp	Exclude stored procedures
-xt	Exclude triggers
-xv	Exclude views
-xx	External unload, external load
-y	Overwrite command file without confirmation
<i>directory</i>	The directory to which the files are written. This is not needed if you use -an or -ac
<i>subscriber</i>	The subscriber for whom the database is to be extracted.

Description

The extraction utility creates a command file and a set of associated data files. The command file can be run against a newly-initialized database to create the database objects and load the data for the remote database.


By default, the command file is named *reload.sql*.

If the remote user is a group, then all the user IDs that are members of that group are extracted. This allows multiple users on a remote database with different user IDs, without requiring a custom extraction process.

Extraction utility options

@data

Use this option to read in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used.

 For more information about configuration files, see [“Using configuration files” \[SQL Anywhere Server - Database Administration\]](#).

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

☞ For more information, see “File Hiding utility (dbfhide)” [[SQL Anywhere Server - Database Administration](#)].

Reload the data to an existing database (-ac)

You can combine the operation of unloading a database and reloading the results into an existing database using this option.

For example, the following command (which should be entered all on one line) loads a copy of the data for the field_user subscriber into an existing database file named *c:\newdata.db*:

```
dbxtract -c "uid=DBA;pwd=sql;dbf=c:\olddata.db"
-ac "uid=DBA;pwd=sql;dbf=c:\newdata.db" field_user
```

If you use this option, no copy of the data is created on disk, so you do not specify an unload directory in the command. This provides greater security for your data, but at some cost for performance.

Create a database for reloading (-an)

You can combine the operations of unloading a database, creating a new database, and loading the data using this option.

For example, the following command (which should be entered all on one line) creates a new database file named *c:\mydatacopy.db* and copies the schema and data for the field_user subscriber of *c:\mydata.db* into it:

```
dbxtract -c "uid=DBA;pwd=sql;dbf=c:\mydata.db"
-an c:\mydatacopy.db field_user
```

If you use this option, no copy of the data is created on disk, so you do not specify an unload directory in the command. This provides greater security for your data, but at some cost for performance.

Do not start subscriptions automatically (-b)

If this option is selected, subscriptions at the consolidated database (for the remote database) and at the remote database (for the consolidated database) must be started explicitly using the START SUBSCRIPTION statement for replication to begin.

Connection parameters (-c)

A set of connection parameters, in a string.

- ◆ **connection parameters** The **user ID** should have DBA authority to ensure that the user has permissions on all the tables in the database.

For example, the following statement (which should be typed on one line) extracts a database for remote user ID **joe_remote** from the sample database running on the **sample_server** server, connecting as user ID DBA with password sql. The data is unloaded into the *c:\extract* directory.

```
dbxtract -c "ENG=sample_server;DBN=demo;
UID=DBA;PWD=sql" c:\extract joe_remote
```

If connection parameters are not specified, connection parameters from the SQLCONNECT environment variable are used, if set.

Unload the data only (-d)

If this option is selected, the schema definition is not unloaded, and publications and subscriptions are not created at the remote database. This option is for use when a remote database already exists with the proper schema, and needs only to be filled with data.

Specify encryption algorithm (-ea)

This option allows you to choose a strong encryption algorithm to encrypt your new database. You can choose either AES (the default) or AES_FIPS for the FIPS-approved algorithm. AES_FIPS uses a separate library and is not compatible with AES. Algorithm names are case insensitive. If you specify the -ea option, you must also specify -ep or -ek.

☞ For more information, see “[Strong encryption](#)” [*SQL Anywhere Server - Database Administration*].

Separately licensed component required

ECC encryption and FIPS-approved encryption require a separate license. All strong encryption technologies are subject to export regulations.

See “[Separately licensed components](#)” [*SQL Anywhere 10 - Introduction*].

Specify encryption key (-ek)

This option allows you to create a strongly encrypted database by specifying an encryption key directly in the command. The algorithm used to encrypt the database is AES or AES_FIPS as specified by the -ea option. If you specify the -ek option without specifying -ea, the AES algorithm is used.

Caution

Protect your key! Be sure to store a copy of your key in a safe location. A lost key will result in a completely inaccessible database, from which there is no recovery.

Prompt for encryption key (-ep)

This option allows you to specify that you want to create a strongly encrypted database by inputting the encryption key in a dialog box. This provides an extra measure of security by never allowing the encryption key to be seen in clear text.

You must input the encryption key twice to confirm that it was entered correctly. If the keys don't match, the initialization fails.

☞ For more information, see “[Strong encryption](#)” [*SQL Anywhere Server - Database Administration*].

Extract fully qualified publications (-f)

In most cases, you do not need to extract fully qualified publication definitions for the remote database, since it typically replicates all rows back to the consolidated database anyway.

However, you may want fully qualified publications for multi-tier setups or for setups where the remote database has rows that are not in the consolidated database.

Internal unload, internal load (-ii)

Using this option forces the reload script to use the internal UNLOAD and LOAD TABLE statements rather than the Interactive SQL OUTPUT and INPUT statements to unload and load data, respectively.

This combination of operations is the default behavior.

External operations takes the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the server.

Internal unload, external load (-ix)

Using this option forces the reload script to use the internal UNLOAD statement to unload data, and the Interactive SQL INPUT statement to load the data into the new database.

External operations takes the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the server.

Perform extraction at a specified isolation level (-l)

The default setting is an isolation level of zero. If you are extracting a database from an active server, you should run it at isolation level 3 (see “Extraction utility” on page 157) to ensure that data in the extracted database is consistent with data on the server. Increasing the isolation level may result in large numbers of locks being used by the extraction utility, and may restrict database use by other users.

Unload the schema definition only (-n)

With this definition, none of the data is unloaded. The reload file contains SQL statements to build the database structure only. You can use the SYNCHRONIZE SUBSCRIPTION statement to load the data over the messaging system. Publications, subscriptions, PUBLISH and SUBSCRIBE permissions are part of the schema.

Output messages to file (-o)

Outputs the messages from the extraction process to a file for later review.

Escape character (-p)

The default escape character (\) can be replaced by another character using this option.

Operate quietly (-q)

Display no messages except errors. This option is not available from other environments. This is available only from the command line utility.

Reload filename (-r)

The default name for the reload command file is *reload.sql* in the current directory. You can specify a different file name with this option.

Output the data unordered (-u)

By default the data in each table is ordered by primary key. Unloads are quicker with the -u option, but loading the data into the remote database is slower.

Verbose mode (-v)

The name of the table being unloaded and the number of rows unloaded are displayed. The SELECT statement used is also displayed.

Exclude foreign key definitions (-xf)

You can use this if the remote database contains a subset of the consolidated database schema, and some foreign key references are not present in the remote database.

External unload, internal load (-xi)

The default behavior for unloading the database is to use the UNLOAD statement, which is executed by the database server. If you choose an external unload, dbxtract uses the OUTPUT statement instead. The OUTPUT statement is executed at the client.

External operations takes the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the server.

Exclude stored procedure (-xp)

Do not extract stored procedures from the database.

Exclude triggers (-xt)

Do not extract triggers from the database.

Exclude views (-xv)

Do not extract views from the database.

External unload, external load (-xx)

Use the OUTPUT statement to unload the data, and the INPUT statement to load the data into the new database.

The default unload behavior is to use the UNLOAD statement, and the default loading behavior is to use the LOAD TABLE statement. The internal UNLOAD and LOAD TABLE statements are faster than OUTPUT and INPUT.

External operations takes the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the server.

Operate without confirming actions (-y)

Without this option, you are prompted to confirm the replacement of an existing command file.

SQL Remote options

Function

Replication options are database options included to provide control over replication behavior.

Syntax

```
SET [ TEMPORARY ] OPTION
[ userid. | PUBLIC. ] option-name = [ option-value ]
```

Parameters

Argument	Description
<i>option-name</i>	The name of the option being changed.
<i>option-value</i>	A string containing the setting for the option.

Description

The following options are available.

OPTION	VALUES	DEFAULT
blob_threshold	integer, in KB	256
compression	-1 to 9	6
delete_old_logs	On, Off, <i>n</i> days	Off
external_remote_options	On, Off	Off
qualify_owners	On, Off	On
quote_all_identifiers	On, Off	Off
replication_error	<i>procedure-name</i>	NULL
save_remote_passwords	On, Off	On
sr_date_format	<i>date-string</i>	yyyy/mm/dd
sr_time_format	<i>time-string</i>	hh:nn:ss.Ssssss
sr_timestamp_format	<i>timestamp-string</i>	yyyy/mm/dd hh:nn:ss.Ssssss
subscribe_by_remote	On, Off	On
verify_threshold	<i>integer</i>	256
verify_all_columns	On, Off	Off

These options are used by the Message Agent, and should be set for the user ID specified in the Message Agent command. They can also be set for general public use.

The options are as follows:

blob_threshold option Any value longer than the blob_threshold option is replicated as a BLOB. That is, it is broken into pieces and replicated in chunks, before being reconstituted by using a SQL variable and concatenating the pieces at the recipient site.

compression option Set the level of compression for messages. Values can be from -1 to 9, and have the following meanings:

- ◆ **-1** Send messages in Version 5 format. Message Agents from previous versions of SQL Remote cannot read messages sent in version 6 format. You should ensure that the compression option is set to -1 until all Message Agents in your system are upgraded to version 6.
- ◆ **0** No compression.
- ◆ **1 to 9** Increasing degrees of compression. Creating messages with high compression can take longer than creating messages with low compression.

delete_old_logs option This option is used by SQL Remote and by the SQL Anywhere Replication Agent. The default setting is Off. When set to On, the Message Agent (DBREMOTE) deletes each old transaction log when all the changes it contains have been sent and confirmed as received. When set to *n* days, logs are deleted that were created more than the specified number of days ago.

external_remote_options This option is used by SQL Remote to indicate whether the message link parameters should be stored in the database (Off) or externally (On). By default, the setting is Off.

qualify_owners option Controls whether SQL statements being replicated by SQL Remote should use qualified object names. The default is On.

quote_all_identifiers option Controls whether SQL statements being replicated by SQL Remote should use quoted identifiers. The default is Off.

When this option is off, the *dbremote* quotes identifiers that require quotes by SQL Anywhere (as it has always done). When the option is on, all identifiers are quoted.

replication_error option Specifies a stored procedure called by the Message Agent when a SQL error occurs. By default no procedure is called.

The replication error procedure must have a single argument of type CHAR, VARCHAR, or LONG VARCHAR. The procedure may be called once with the SQL error message and once with the SQL statement that causes the error.

While the option allows you to track and monitor SQL errors in replication, you must still design them out of your setup: this option is not intended to resolve such errors.

You can use a table with DEFAULT CURRENT REMOTE USER to record the remote site that caused the error.

save_remote_passwords option When a password is entered into the message link dialog box on first connection, the parameter values are saved. By default, `save_remote_passwords` is on and the password is saved. If you are storing the message link parameters externally, rather than in the database, you may not want to save the passwords. You can prevent the passwords from being saved by setting this option to No.

sr_date_format option The Message Agent uses this option when replicating columns that store a date. The option is a string build from the following symbols:

Symbol	Description
yy	Two digit year
yyyy	Four-digit year
mm	Two-digit month
mmm	Character format for month
dd	Two-digit day

Each symbol is substituted with the date being replicated.

If you set the **mm** format symbol in upper case, the corresponding characters are also upper case.

For the digit formats, the case of the option setting controls padding. If the symbols are the same case (such as DD), the number is padded with zeroes. If the symbols are mixed case (such as Mm), the number is not zero padded.

sr_time_format option The Message Agent uses this option when replicating columns that store a time. The option is a string build from the following symbols:

Symbol	Description
hh	Two digit hours (24-hour clock)
nn	Two-digit minutes
mm	Two-digit minutes if following a colon (as in hh:mm)
ss[s...]	Two-digit seconds plus optional fractions of a second.

Using mixed case in the formatting string suppresses leading zeroes.

sr_timestamp_format The Message Agent replicates datetime information using this option. This is the timestamp, datetime, and smalldatetime data types.

The format strings are taken from the `sr_date_format` and `sr_time_format` settings.

The default setting is the `sr_date_format` setting, followed by the `sr_time_format` setting.

subscribe_by_remote option When set to On, operations from remote databases on rows with a subscribe by value that is NULL or an empty string assume the remote user is subscribed to the row. When set to Off, the remote user is assumed not to be subscribed to the row.

The only limitation of this option is that it will lead to errors if a remote user really does want to INSERT (or UPDATE) a row with a NULL or empty subscription expression (for information held only at the consolidated database). This is reasonably obscure and can be worked around by assigning a subscription value in your installation that belongs to no remote user.

☞ For more information about this option, see [“Using the subscribe_by_remote option with many-to-many relationships”](#) on page 53.

verify_threshold option If the data type of a column is longer than the threshold, old values for the column are not verified when an UPDATE is replicated. The default setting is 1000.

This option keeps the size of SQL Remote messages down, but has the disadvantage that conflicting updates of long values are not detected.

verify_all_columns option The default setting is Off. When set to On, messages containing updates published by the local database are sent with all column values included, and a conflict in any column triggers a RESOLVE UPDATE trigger at the subscriber database.

Example

The following statement sets the verify_all_columns option to Off in SQL Anywhere, for all users:

```
SET OPTION PUBLIC.verify_all_columns = 'Off'
```

SQL Remote event-hook procedures

The following stored procedure names and arguments provide the interface for customizing synchronization at SQL Remote databases.

Notes

Unless otherwise stated, the following apply to event-hook procedures:

- ◆ The stored procedures must have DBA authority.
- ◆ The procedure must not commit or rollback operations, or perform any action that performs an implicit commit. The actions of the procedure are automatically committed by the calling application.
- ◆ You can troubleshoot the hooks by turning on the Message Agent verbose mode.

The #hook_dict table

The #hook_dict table is created immediately before a hook is called using the following CREATE statement:

```
CREATE table #hook_dict(
  name VARCHAR(128) NOT NULL UNIQUE,
  value VARCHAR(255) NOT NULL )
```

The Message Agent uses the #hook_dict table to pass values to hook functions; hook functions use the #hook_dict table to pass values back to the Message Agent.

sp_hook_dbremote_begin and sp_hook_ssrm Begin

Function

Use this stored procedure to add custom actions at the beginning of the replication process.

Rows in #hook_dict table

Name	Values	Description
send	true or false	Indicates if the process is performing the send phase of replication.
receive	true or false	Indicates if the process is performing the receive phase of replication.

Description

If a procedure of this name exists, it is called when the Message Agent starts.

sp_hook_dbremote_end and sp_hook_ssrm_end

Function

Use this stored procedure to add custom actions just before the Message Agent exits.

Rows in #hook_dict table

Name	Values	Description
send	true or false	Indicates if the process is performing the send phase of replication.
receive	true or false	Indicates if the process is performing the receive phase of replication
exit code	integer	A non-zero exit code indicates an error.

Description

If a procedure of this name exists, it is called as the last event before the Message Agent shuts down.

sp_hook_dbremote_shutdown and sp_hook_ssrm_shutdown

Function

Use this stored procedure to initiate a Message Agent shutdown.

Rows in #hook_dict table

Name	Values	Description
send	true or false	Indicates if the process is performing the send phase of replication.
receive	true or false	Indicates if the process is performing the receive phase of replication
shutdown	true or false	This row is false when the procedure is called. If the procedure updates the row to true the Message Agent is shut down.

Description

If a procedure of this name exists, it is called when the Message Agent is neither sending nor receiving messages, and permits a hook-initiated shutdown of the Message Agent.

sp_hook_dbremote_receive_begin and sp_hook_ssrm_receive_begin

Function

Use this stored procedure to perform actions before the start of the receive phase of replication.

Rows in #hook_dict

None

[sp_hook_dbremote_receive_end](#) and [sp_hook_ssrm_receive_end](#)**Function**

Use this stored procedure to perform actions after the end of the receive phase of replication.

Rows in #hook_dict

None

[sp_hook_dbremote_send_begin](#) and [sp_hook_ssrm_send_begin](#)**Function**

Use this stored procedure to perform actions before the start of the send phase of replication.

Rows in #hook_dict

None

[sp_hook_dbremote_send_end](#) and [sp_hook_ssrm_send_end](#)**Function**

Use this stored procedure to perform actions after the end of the send phase of replication.

Rows in #hook_dict

None

[sp_hook_dbremote_message_sent](#) and [sp_hook_ssrm_message_sent](#)**Function**

Use this stored procedure to perform actions after any message is sent.

Rows in #hook_dict

Name	Values
remote user	The message destination

sp_hook_dbremote_message_missing and sp_hook_ssrmt_message_missing

Function

Use this stored procedure to perform actions when the Message Agent has determined that one or more messages is missing from a remote user.

Rows in #hook_dict

Name	Values
remote user	The name of the remote user who will have to resend messages.

sp_hook_dbremote_message_apply_begin and sp_hook_ssrmt_message_apply_begin

Function

Use this stored procedure to perform actions just before the Message Agent applies a set of messages from a user.

Rows in #hook_dict

Name	Values
remote user	The name of the remote user who sent the messages about to be applied.

sp_hook_dbremote_message_apply_end and sp_hook_ssrmt_message_apply_end

Function

Use this stored procedure to perform actions just after the Message Agent has applied a set of messages from a user.

Rows in #hook_dict

Name	Values
remote user	The name of the remote user who sent the messages that have been applied.

CHAPTER 9

System Objects for SQL Remote

Contents

SQL Remote system tables	172
--------------------------------	-----

About this chapter

This chapter provides links to the SQL Anywhere system tables that are used by SQL Remote.

SQL Remote system tables

SQL Remote system information is held in the SQL Anywhere catalog. A more comprehensible version of this information is held in a set of system views. Following are the views you can use to access SQL Remote data:

- ◆ “SYSARTICLE system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSARTICLECOL system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSPUBLICATION system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSREMOTEOPTION system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSREMOTEOPTIONTYPE system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSREMOTETYPE system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSREMOTEEUSER system view” [[SQL Anywhere Server - SQL Reference](#)]
- ◆ “SYSSUBSCRIPTION system view” [[SQL Anywhere Server - SQL Reference](#)]

CHAPTER 10

SQL Remote SQL Statements

Contents

SQL Remote statements	174
-----------------------------	-----

About this chapter

This chapter provides links to the SQL Anywhere SQL statements that are used by SQL Remote.

SQL Remote statements

Following are the SQL statements used for executing SQL Remote commands:

- ◆ “ALTER REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “CREATE PUBLICATION statement [MobiLink] [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “CREATE REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “CREATE SUBSCRIPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “CREATE TRIGGER statement” [*SQL Anywhere Server - SQL Reference*]
- ◆ “DROP PUBLICATION statement [MobiLink] [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “DROP REMOTE MESSAGE TYPE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “DROP SUBSCRIPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “GRANT CONSOLIDATE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “GRANT PUBLISH statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “GRANT REMOTE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “GRANT REMOTE DBA statement [MobiLink] [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “PASSTHROUGH statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “REMOTE RESET statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “REVOKE CONSOLIDATE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “REVOKE PUBLISH statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “REVOKE REMOTE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “REVOKE REMOTE DBA statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “SET REMOTE OPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “START SUBSCRIPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “STOP SUBSCRIPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]
- ◆ “UPDATE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]

Part V. Appendices

The appendices provide additional information that is not necessarily required for everyday use of the application.

Appendix A

Supported Platforms and Message Links

About this appendix

This appendix summarizes the platforms and message links that SQL Remote supports.

Supported message systems

SQL Remote exchanges data among databases using an underlying message system. SQL Remote supports the following message systems:

- ◆ **File sharing** A simple system requiring no extra software.
- ◆ **FTP** Internet file transfer protocol.
- ◆ **SMTP/POP** Internet email protocol.
- ◆ **MAPI** Microsoft Messaging Application Programming Interface, used in Microsoft products and in cc:Mail release 8 and later.
- ◆ **VIM** Vendor Independent Messaging, used in Lotus Notes and in some versions of Lotus cc:Mail.

Not all systems are supported on all operating systems. For all systems other than the file sharing system, you must have purchased and installed the appropriate message system software for SQL Remote to function over this system. SQL Remote does not include the underlying message system software.

Supported operating systems

SQL Remote for SQL Anywhere is available for the following operating systems:

- ◆ **Windows 2000/XP** All message links.
- ◆ **Windows CE** FILE, FTP, and SMTP/POP links. For the file link, *dbremote* looks in *\My Documents \Synchronized Files*. On the desktop machine, the SQLREMOTE environment variable or directory message link parameter for the FILE link should be set to the following:

```
%SystemRoot%\Profiles\userid\Personal\ce-machine-name\ Synchronized Files
```

where *userid* and *ce-machine-name* are set to the appropriate values. With this setup, ActiveSync automatically synchronizes the message files between the desktop and CE system.

Check Mobile Devices ► Tools ► ActiveSync Options to ensure that file synchronization is activated.

☞ For information on setting message link parameters, see [“The file message system” on page 100](#).

- ◆ **Sun Microsystems Solaris/Sparc** File sharing, FTP, and SMTP/POP only.
- ◆ **Novell NetWare** File sharing, FTP, and SMTP/POP only.
- ◆ **Linux** File sharing, FTP, and SMTP/POP only.

☞ For information about Unix support for SQL Remote, see [SQL Remote for SQL Anywhere](#).

Index

Symbols

#hook_dict table

SQL Remote dbremote, 167

SQL Remote unique primary keys, 64

-a option

SQL Remote [dbremote], 150

-ac option

extraction [dbxtract] utility, 159

-al option

extraction [dbxtract] utility, 157

-an option

extraction [dbxtract] utility, 159

-b option

extraction [dbxtract] utility, 159

SQL Remote [dbremote], 150

-c option

extraction [dbxtract] utility, 159

SQL Remote [dbremote], 150

-d option

extraction [dbxtract] utility, 160

-dl option

SQL Remote [dbremote], 150

-ea option

extraction [dbxtract] utility, 160

-ek option

extraction [dbxtract] utility, 160

SQL Remote [dbremote], 151

-ep option

extraction [dbxtract] utility, 160

SQL Remote [dbremote], 151

-f option

extraction [dbxtract] utility, 160

-g option

SQL Remote [dbremote], 151

-ii option

extraction [dbxtract] utility, 160

-ix option

extraction [dbxtract] utility, 161

-k option

SQL Remote [dbremote], 151

-l option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 151

SQL Remote Message Agent, 110

-m option

SQL Remote [dbremote], 151

-ml option

SQL Remote [dbremote], 152

-n option

extraction [dbxtract] utility, 161

-o option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 152

-os option

SQL Remote [dbremote], 152

-ot option

SQL Remote [dbremote], 152

-p option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 152

-q option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 152

-r option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 152

-rd option

SQL Remote [dbremote], 152

-ro option

SQL Remote [dbremote], 153

-rp option

SQL Remote [dbremote], 153

-rt option

SQL Remote [dbremote], 153

-ru option

SQL Remote [dbremote], 153

-s option

SQL Remote [dbremote], 153

-sd option

SQL Remote [dbremote], 153

-t option

SQL Remote [dbremote], 153

-u option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 154

-ud option

SQL Remote [dbremote], 154

-ux option

SQL Remote [dbremote], 154

-v option

extraction [dbxtract] utility, 161

SQL Remote [dbremote], 154

- w option
 - SQL Remote [dbremote], 154
- x option
 - SQL Remote [dbremote], 154
- xf option
 - extraction [dbxtract] utility, 161
- xh option
 - extraction [dbxtract] utility, 157
- xi option
 - extraction [dbxtract] utility, 161
- xp option
 - extraction [dbxtract] utility, 162
- xt option
 - extraction [dbxtract] utility, 162
- xv option
 - extraction [dbxtract] utility, 162
- xx option
 - extraction [dbxtract] utility, 162
- y option
 - extraction [dbxtract] utility, 162
- @data option
 - extraction [dbxtract] utility, 158
 - SQL Remote [dbremote], 150

A

- about SQL Remote, 4
- addresses
 - SQL Remote file sharing, 100
 - SQL Remote FTP, 101
- administering
 - SQL Remote, 123
- administering SQL Remote
 - about, 123
- articles
 - SQL Remote creation, 31

B

- backups
 - SQL Remote recovery procedures, 109
 - SQL Remote remote databases, 139
 - SQL Remote transaction log management, 130
- batch mode
 - SQL Remote Message Agent, 108
- blob_threshold option
 - SQL Remote option, 164

C

- cache
 - SQL Remote, 113
- ccMail
 - SQL Remote, 96
- compression option
 - SQL Remote option, 164
- conflict detection
 - SQL Remote, 20
- conflict resolution
 - SQL Remote approaches, 55
 - SQL Remote triggers, 56
- conflicts
 - SQL Remote locking, 39
 - SQL Remote management, 55
 - SQL Remote not errors in, 126
 - SQL Remote reporting , 61
 - SQL Remote, about, 26
- connections
 - SQL Remote Message Agent, 109
- CONSOLIDATE permissions
 - SQL Remote, 89
 - SQL Remote granting, 93
- consolidated databases
 - SQL Remote, 10
- Contacts SQL Remote example
 - about, 42
- continuous mode
 - SQL Remote Message Agent, 108
- conventions
 - documentation, x
 - file names in documentation, xii
- create article wizard
 - SQL Remote adding articles in , 36
- Create Publication wizard
 - SQL Remote, 31
- create SQL Remote message type wizard
 - adding message types in Sybase Central, 97
- CREATE SUBSCRIPTION statement
 - SQL Remote, 71

D

- daemon
 - SQL Remote dbremote, 154
 - SQL Remote Message Agent, 154
- data movement technologies
 - SQL Remote replication, 3

- data recovery
 - SQL Remote, 109
- database extraction utility
 - SQL Remote, 157
 - syntax, 157
- dbo user
 - SQL Remote system objects, 156
- dbremote
 - #hook_dict table, 167
 - about SQL Remote, 108
 - introduction, 9
 - options, 148
 - SQL Remote security, 124
 - syntax, 148
- dbunload utility
 - SQL Remote, 139
- dbxtract utility
 - about, 79
 - options, 158
 - sp_hook_dbxtract_begin procedure, 64
 - SQL Remote, 81, 157
 - syntax, 157
- debug control parameter
 - SQL Remote FILE message type, 100
 - SQL Remote FTP message type, 101
 - SQL Remote MAPI message type, 105
 - SQL Remote SMTP message type, 104
 - SQL Remote VIM message type, 106
- delete_old_logs option
 - SQL Remote option, 164
 - SQL Remote transaction log management, 134
- Deleting Corrupt Message error
 - SQL Remote, 118
- deploying
 - SQL Remote databases, 75
- design
 - SQL Remote, 29
 - SQL Remote many-to-many relationships, 48
 - SQL Remote principles, 17
- design overview
 - SQL Remote, 18
- directory control parameter
 - SQL Remote FILE message type, 100
- directory option
 - SQL Remote [dbremote], 148
 - SQL Remote [dbxtract], 157
- documentation
 - conventions, x

- SQL Anywhere, viii
- dropping
 - SQL Remote message types, 98
 - SQL Remote publications, 37

E

- encode_dll control parameter
 - SQL Remote FILE message type, 100
 - SQL Remote FTP message type, 101
- encoding
 - SQL Remote custom, 119
- encoding and compressing messages
 - SQL Remote, 118
- encoding scheme
 - SQL Remote, 118
- encryption
 - SQL Remote, 110
- environment variables
 - SQLREMOTE, 99
- errors
 - SQL Remote default handling, 126
 - SQL Remote reporting by Message Agent, 126
- event hooks
 - sp_hook_dbremote_begin stored procedure, 167
 - sp_hook_dbremote_end SQL Remote stored procedure, 168
 - sp_hook_dbremote_message_apply_begin stored procedure, 170
 - sp_hook_dbremote_message_apply_end stored procedure, 170
 - sp_hook_dbremote_message_missing stored procedure, 170
 - sp_hook_dbremote_message_sent stored procedure, 169
 - sp_hook_dbremote_receive_begin stored procedure, 168
 - sp_hook_dbremote_receive_end stored procedure, 169
 - sp_hook_dbremote_send_begin stored procedure, 169
 - sp_hook_dbremote_send_end stored procedure, 169
 - sp_hook_dbremote_shutdown stored procedure, 168
 - sp_hook_ssrm Begin stored procedure, 167
 - sp_hook_ssrm_end stored SQL Remote procedure, 168

- sp_hook_ssrm_message_apply_begin stored procedure, 170
- sp_hook_ssrm_message_apply_end stored procedure, 170
- sp_hook_ssrm_message_missing stored procedure, 170
- sp_hook_ssrm_message_sent stored procedure, 169
- sp_hook_ssrm_receive_begin stored procedure, 168
- sp_hook_ssrm_receive_end stored procedure, 169
- sp_hook_ssrm_send_begin stored procedure, 169
- sp_hook_ssrm_send_end stored procedure, 169
- sp_hook_ssrm_shutdown stored procedure, 168

- external_remote_options option
 - SQL Remote option, 164

- extract database wizard
 - extracting remote databases in Sybase Central, 156

- extracting
 - SQL Remote and mixed operating systems, 79
 - SQL Remote databases, 75, 79
 - SQL Remote reload files, 81

- extraction utility
 - options, 158
 - SQL Remote, 79, 81, 157
 - syntax, 157

F

- feedback
 - documentation, xv
 - providing, xv
- FILE message type
 - SQL Remote, 96, 100
 - SQL Remote control parameters, 100
- Force_Download control parameter
 - SQL Remote MAPI message type, 105
- frequency
 - SQL Remote, 92
- FTP message type
 - SQL Remote, 96, 101
 - SQL Remote control parameters, 101
 - SQL Remote troubleshooting, 102

G

- global autoincrement
 - SQL Remote, 63

- global_database_id option
 - SQL Remote, 64
- GRANT PUBLISH statement
 - SQL Remote, 89

H

- handling lost or corrupt messages
 - SQL Remote, 121
- hooks
 - SQL Remote, 167
- host control parameter
 - SQL Remote FTP message type, 101
- how statements are replicated
 - SQL Remote, 19

I

- icons
 - used in manuals, xii
- install-dir
 - documentation usage, xii
- invalid_extensions parameter
 - SQL Remote FILE message type, 100
 - SQL Remote FTP message type, 101
- IPM_Receive control parameter
 - SQL Remote MAPI message type, 105
- IPM_Send control parameter
 - SQL Remote MAPI message type, 105

L

- locking
 - SQL Remote, 39
- log management
 - SQL Remote, 109
- Lotus Notes
 - SQL Remote supported message types, 96
 - SQL Remote VIM message system, 106

M

- many-to-many relationships
 - SQL Remote publication design, 48
- MAPI message type
 - SQL Remote, 96, 105
 - SQL Remote control parameters, 105
- media failures
 - SQL Remote, 109
- Message Agent
 - about SQL Remote, 108

- batch mode, 108
- connections, 109
- continuous mode, 108
- daemon, 154
- introduction, 9
- message tracking, 120
- options, 148
- output in SQL Remote , 126
- performance, 112
- reporting errors, 126
- running, 124
- running as a service, 124
- security, 110
- security in SQL Remote, 124
- settings, 110
- SQL Remote administration, 88
- subscription processing, 25
- syntax, 148
- transaction log management, 130, 139
- trigger replication, 21
- tuning throughput, 112
- message link parameters
 - SQL Remote external_remote_options, 164
- message tracking
 - SQL Remote administration, 88
- message type control parameters
 - SQL Remote, 99
- message types
 - SMTP in SQL Remote, 103
 - SQL Remote, 96, 97
 - SQL Remote and VIM, 106
 - SQL Remote dropping, 98
 - SQL Remote file sharing, 100
 - SQL Remote FTP, 101
 - SQL Remote MAPI, 105
 - SQL Remote VIM, 106
- messages
 - SQL Remote caching, 113
 - SQL Remote synchronizing databases, 86
- Microsoft Exchange
 - SQL Remote profile, 105
- multi-tier installations
 - SQL Remote permissions, 94

N

- NetWare
 - SQL Remote, 101

- SQL Remote supported message types, 96
- newsgroups
 - technical support, xv
- Notes, vii
 - (see also Lotus Notes)
 - SQL Remote, 96
 - SQL Remote VIM message system, 106

O

- options
 - SQL Remote, 163
- output_log_send_limit remote option
 - SQL Remote troubleshooting, 110
- output_log_send_now remote option
 - SQL Remote troubleshooting, 110
- output_log_send_on_error remote option
 - SQL Remote troubleshooting, 110

P

- partitioning
 - SQL Remote, 31
- passthrough mode
 - SQL Remote, 141
- PASSTHROUGH statement
 - SQL Remote, 141
- password control parameter
 - SQL Remote FTP message type, 101
 - SQL Remote VIM message type, 106
- passwords
 - SQL Remote saving of, 165
- Path control parameter
 - SQL Remote VIM message type, 106
- performance
 - SQL Remote Message Agent, 112
 - SQL Remote publications, 40
- permissions
 - SQL Remote granting CONSOLIDATE, 93
 - SQL Remote management, 89
 - SQL Remote multi-tier installations, 94
 - SQL Remote revoking CONSOLIDATE, 94
 - SQL Remote revoking REMOTE, 94
- platforms
 - SQL Remote supported operating systems, 177
- policy example
 - SQL Remote publications, 48
- pop3_host control parameter
 - SQL Remote SMTP message type, 104

- pop3_password control parameter
 - SQL Remote SMTP message type, 104
- pop3_userid control parameter
 - SQL Remote SMTP message type, 104
- port control parameter
 - SQL Remote FTP message type, 101
- primary key pools
 - SQL Remote, 66
- primary keys
 - SQL Remote, 61
 - SQL Remote primary key pools, 66
 - SQL Remote unique values, 63
- principles of SQL Remote design
 - about, 17
- publication design
 - SQL Remote, 29
- publications
 - SQL Remote alteration, 36
 - SQL Remote creation, 31
 - SQL Remote design, 39
 - SQL Remote dropping, 37
 - SQL Remote locking, 39
 - SQL Remote many-to-many relationships, 48
 - SQL Remote replication, 10
 - SQL Remote transactions, 39
- PUBLISH permissions
 - SQL Remote, 89
- publishing
 - SQL Remote, 31
- Q**
- qualify_owners option
 - SQL Remote option, 164
- quote_all_identifiers option
 - SQL Remote option, 164
- R**
- receive_all control parameter
 - SQL Remote VIM message type, 106
- reconnect_pause parameter
 - SQL Remote FTP message type, 101
- reconnect_retries parameter
 - SQL Remote FTP message type, 101
- recovery
 - SQL Remote, 109
- referential integrity
 - SQL Remote, 61
- reload files
 - SQL Remote database extraction, 81
- REMOTE permissions
 - SQL Remote, 89
- remote permissions
 - SQL Remote management, 89
- remoteuser SQL Remote table
 - message tracking in SQL Remote, 120
 - using in SQL Remote, 120
- replication, vii
 - (see also SQL Remote)
 - backup procedures in SQL Remote, 130
 - backups, 139
 - BLOBs, 23
 - conflicts, 26
 - data definition statements, 22
 - data recovery, 109
 - data types, 23
 - Message Agent, 148
 - mixed operating systems, 79
 - passthrough mode, 141
 - primary key errors in SQL Remote, 61
 - primary keys in SQL Remote, 63
 - procedures, 21
 - publication design, 39
 - publications, 10
 - referential integrity errors in SQL Remote, 61
 - SQL Remote dbremote, 148
 - SQL statements, 141
 - subscriptions, 10
 - transaction log management, 139
 - transaction log management in SQL Remote, 130
 - triggers, 21
 - triggers in SQL Remote, 61
 - upgrading databases, 139
- replication conflicts
 - SQL Remote, 26
 - SQL Remote management, 55
- replication errors
 - SQL Remote, 26
- replication errors and conflicts
 - SQL Remote, 26
- replication options
 - blob_threshold, 164
 - compression, 164
 - delete_old_logs, 164
 - external_remote_options, 164
 - qualify_owners, 164

- quote_all_identifiers, 164
- replication_error, 126, 164
- save_remote_passwords, 165
- SQL Remote replication_error, 27
- sr_date_format, 165
- sr_time_format, 165
- sr_timestamp_format, 165
- subscribe_by_remote, 166
- verify_all_columns, 166
- verify_threshold, 166
- replication_error option
 - SQL Remote error handling procedures, 126
 - SQL Remote option, 164
- reporting
 - SQL Remote conflicts , 61
- reporting errors
 - SQL Remote Message Agent, 126
- resend requests
 - SQL Remote, 114
- REVOKE PUBLISH statement
 - SQL Remote, 89
- REVOKE statement
 - SQL Remote, 94
- revoking consolidate permissions
 - SQL Remote, 94
- revoking remote permissions
 - SQL Remote, 94
- root control parameter
 - SQL Remote FTP message type, 101
- running
 - SQL Remote Message Agent, 124

S

- samples
 - SQL Remote policy example, 48
- samples-dir
 - documentation usage, xii
- save_remote_passwords option
 - SQL Remote option, 165
- selecting a send frequency
 - SQL Remote, 92
- SEND AT
 - SQL Remote frequency setting, 92
- SEND EVERY
 - SQL Remote frequency setting, 92
- send frequency
 - SQL Remote Message Agent, 108
 - SQL Remote selection, 92
- send_vim_mail control parameter
 - SQL Remote VIM message type, 106
- services
 - SQL Remote Message Agent, 124
- SMTP message type
 - control parameter in SQL Remote, 103
 - SQL Remote, 96, 103
- SMTP/POP
 - SQL Remote addresses, 104
- smtp_authenticate control parameter
 - SQL Remote SMTP message type, 104
- smtp_host control parameter
 - SQL Remote SMTP message type, 104
- smtp_password control parameter
 - SQL Remote SMTP message type, 104
- smtp_userid control parameter
 - SQL Remote SMTP message type, 104
- sp_hook_dbremote_begin stored procedure
 - SQL Remote syntax, 167
- sp_hook_dbremote_end stored procedure
 - SQL Remote syntax, 168
- sp_hook_dbremote_message_apply_begin stored procedure
 - SQL Remote syntax, 170
- sp_hook_dbremote_message_apply_end stored procedure
 - SQL Remote syntax, 170
- sp_hook_dbremote_message_missing stored procedure
 - SQL syntax, 170
- sp_hook_dbremote_message_sent stored procedure
 - SQL Remote syntax, 169
- sp_hook_dbremote_receive_begin stored procedure
 - SQL Remote syntax, 168
- sp_hook_dbremote_receive_end stored procedure
 - SQL Remote syntax, 169
- sp_hook_dbremote_send_begin stored procedure
 - SQL Remote syntax, 169
- sp_hook_dbremote_send_end stored procedure
 - SQL Remote syntax, 169
- sp_hook_dbremote_shutdown stored procedure
 - SQL Remote syntax, 168
- sp_hook_dbextract_begin procedure
 - SQL Remote, 64
- sp_hook_ssrm Begin stored procedure
 - SQL Remote syntax, 167
- sp_hook_ssrm_end stored procedure

- SQL Remote syntax, 168
- sp_hook_ssrm_message_apply_begin stored procedure
 - SQL Remote syntax, 170
- sp_hook_ssrm_message_apply_end stored procedure
 - SQL Remote syntax, 170
- sp_hook_ssrm_message_missing stored procedure
 - SQL Remote syntax, 170
- sp_hook_ssrm_message_sent stored procedure
 - SQL Remote syntax, 169
- sp_hook_ssrm_receive_begin stored procedure
 - SQL Remote syntax, 168
- sp_hook_ssrm_receive_end stored procedure
 - SQL Remote syntax, 169
- sp_hook_ssrm_send_begin stored procedure
 - SQL Remote syntax, 169
- sp_hook_ssrm_send_end stored procedure
 - SQL Remote syntax, 169
- sp_hook_ssrm_shutdown stored procedure
 - SQL Remote syntax, 168
- SQL Anywhere
 - documentation, viii
- SQL Remote, vii
 - (see also replication)
 - about, 4
 - administering, 88, 123
 - backup procedures, 130
 - backups, 139
 - components, 8
 - concepts, 7
 - conflict detection, 20
 - dbxtract utility, 157
 - deployment overview, 76
 - design overview, 29
 - design principles, 17
 - event hooks, 167
 - Message Agent introduction, 9
 - Message Agent performance, 112
 - message delivery, 120
 - message tracking and delivery, 120
 - mobile workforces, 12
 - publications, 10
 - replicating data types, 23
 - replicating dates, 24
 - replicating DDL statements, 22
 - replicating deletes, 19
 - replicating inserts, 19
 - replicating procedures, 21
 - replicating times, 24
 - replicating triggers, 21
 - replicating updates, 19
 - replication system recovery procedures, 109
 - resolving date conflicts, 58
 - SQL Anywhere system tables, 172
 - SQL statements, 174
 - subscribers, 12
 - subscriptions, 10
 - supported message systems, 177
 - supported platforms, 177
 - system objects, 172
 - transaction log management, 130
 - unloading databases, 139
 - upgrading consolidated databases, 139
 - utilities and options reference, 147
- SQL Remote administration
 - about, 87
- SQL Remote components
 - about, 8
- SQL Remote concepts
 - about, 7
- SQL Remote options
 - about, 163
- SQL statements
 - SQL Remote list, 174
- SQLANY.INI
 - SQL Remote, 99
- SQLREMOTE environment variable
 - alternative to, 100
 - setting message control parameters, 99
- sr_date_format option
 - SQL Remote option, 165
- sr_time_format option
 - SQL Remote options, 165
- sr_timestamp_format option
 - SQL Remote option, 165
- stable queue
 - SQL Remote cleaning, 151
- subscribe_by_remote option
 - SQL Remote option, 166
- subscriber option
 - SQL Remote [dbxtract], 157
- subscription expressions
 - SQL Remote cost of evaluating, 25
 - SQL Remote using, 34
- subscriptions
 - SQL Remote creation, 71

- SQL Remote replication, 10
- support
 - newsgroups, xv
- supported platforms
 - SQL Remote, 177
- suppress_dialogs control parameter
 - SQL Remote MAPI message type, 105
 - SQL Remote SMTP message type, 104
 - SQL Remote VIM message type, 106
- suppress_dialogs parameter
 - SQL Remote FTP message type, 101
- synchronizing data over a message system
 - SQL Remote, 86
- system objects
 - SQL Remote, 172
 - SQL Remote dbo user, 156
- system tables
 - SQL Remote, 172

T

- technical support
 - newsgroups, xv
- territory realignment
 - SQL Remote foreign keys, 44
 - SQL Remote many-to-many relationships, 51
 - SQL Remote UPDATES, 19
- testing
 - SQL Remote deployments, 77
- tracking SQL errors
 - SQL Remote, 27
- transaction log
 - Message Agent, 148
 - SQL Remote Message Agent, 148
 - SQL Remote offsets, 120
 - SQL Remote publications, 25
- transaction log mirror
 - SQL Remote, 130
- triggers
 - SQL Remote, 45, 61
- troubleshooting
 - SQL Remote errors, 110
- typical SQL Remote setups
 - about, 13

U

- unique column values
 - SQL Remote, 63

- Unix
 - SQL Remote supported message types, 96
- unlink_delay control parameter
 - SQL Remote FILE message type, 100
- unloading
 - SQL Remote consolidated databases, 139
- UPDATE conflicts
 - SQL Remote, 55
- UPDATE statement
 - SQL Remote territory realignment, 19
- upgrading
 - SQL Remote consolidated databases, 139
- user control parameter
 - SQL Remote FTP message type, 101
- Userid control parameter
 - SQL Remote VIM message type, 106

V

- verify_all_columns option
 - SQL Remote option, 166
- verify_threshold option
 - SQL Remote option, 166
- VIM message type
 - SQL Remote, 96, 106
 - SQL Remote control parameters, 106

W

- welcome to SQL Remote
 - , 3
- Windows
 - SQL Remote supported message types, 96
- wizards
 - extract database in SQL Remote, 156
