SQL Remote™

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Copyright © 2010 iAnywhere Solutions, Inc. Portions copyright © 2010 Sybase, Inc. All rights reserved.

This documentation is provided AS IS, without warranty or liability of any kind (unless provided by a separate written agreement between you and iAnywhere).

You may use, print, reproduce, and distribute this documentation (in whole or in part) subject to the following conditions: 1) you must retain this and all other proprietary notices, on all copies of the documentation or portions thereof, 2) you may not modify the documentation, 3) you may not do anything to indicate that you or anyone other than iAnywhere is the author or source of the documentation.

iAnywhere®, Sybase®, and the marks listed at http://www.sybase.com/detail?id=1011207 are trademarks of Sybase, Inc. or its subsidiaries. ® indicates registration in the United States of America.

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

Contents

About this book v

About the SQL Anywhere documentation v

Introducing SQL Remote 1

Typical SQL Remote setups 1
SQL Remote components 4
Understanding the SQL Remote replication process 6

Creating SQL Remote systems 9

Publications and articles 10
User permissions 18
Subscriptions 30
Understanding transaction log-based replication 31
Replication conflicts and errors 39
Update conflicts 40
Row not found errors 47
Referential integrity errors 47
Duplicate primary key errors 50
Partitioning rows among remote databases 56
Using disjoint data partitions 56
Using overlap partitions 61
Assigning unique identification numbers to each database 68

Managing SQL Remote systems 71

Extracting remote databases 72
Extracting remote databases to a reload file 73
Understanding the SQL Remote Message Agent (dbremote) 78
Improving SQL Remote performance 84
Understanding the Guaranteed Message Delivery System 93
Controlling message size 98

SQL Remote message systems 99
Backing up SQL Remote systems 109
Recover consolidated databases manually 114
Recover consolidated databases automatically 115
Reporting and handling replication errors 117
Security 122
Upgrading and resynchronization 122
SQL Remote passthrough mode 124
Resynchronizing subscriptions 126

SQL Remote reference 131

SQL Remote utilities and options reference 131
SQL Remote system objects 155
SQL Remote SQL statements 156

Index 159

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

About this book

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.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in four formats:

- **DocCommentXchange** DocCommentXchange is a community for accessing and discussing SQL Anywhere documentation on the web.

  To access the documentation, go to http://dcx.sybase.com.

- **HTML Help** On Windows platforms, the HTML Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools.

  To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **HTML Help (English)**.

- **Eclipse** On Unix platforms, the complete Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere installation.

- **PDF** The complete set of SQL Anywhere books is provided as a set of Portable Document Format (PDF) files. You must have a PDF reader to view information.

  To access the PDF documentation on Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **PDF (English)**.

  To access the PDF documentation on Unix operating systems, use a web browser to open */documentation/en/pdf/index.html* under the SQL Anywhere installation directory.

Documentation conventions

This section lists the conventions used in this documentation.

Operating systems

SQL Anywhere runs on a variety of platforms. Typically, the behavior of the software is the same on all platforms, but there are variations or limitations. These are commonly based on the underlying operating system (Windows, Unix), and seldom on the particular variant (IBM AIX, Windows Mobile) or version.

To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family includes platforms that are used primarily on server, desktop, and laptop computers, as well as platforms used on mobile devices. Unless otherwise specified, when the documentation refers to Windows, it refers to all supported Windows-based platforms, including Windows Mobile.

  Windows Mobile is based on the Windows CE operating system, which is also used to build a variety of platforms other than Windows Mobile. Unless otherwise specified, when the documentation refers to Windows Mobile, it refers to all supported platforms built using Windows CE.

- **Unix** Unless otherwise specified, when the documentation refers to Unix, it refers to all supported Unix-based platforms, including Linux and Mac OS X.

For the complete list of platforms supported by SQL Anywhere, see "Supported platforms" [*SQL Anywhere 12 - Introduction*].

Directory and file names

Usually references to directory and file names are similar on all supported platforms, with simple transformations between the various forms. In these cases, Windows conventions are used. Where the details are more complex, the documentation shows all relevant forms.

These are the conventions used to simplify the documentation of directory and file names:

- **Uppercase and lowercase directory names** On Windows and Unix, directory and file names may contain uppercase and lowercase letters. When directories and files are created, the file system preserves letter case.

  On Windows, references to directories and files are *not* case sensitive. Mixed case directory and file names are common, but it is common to refer to them using all lowercase letters. The SQL Anywhere installation contains directories such as *Bin32* and *Documentation*.

  On Unix, references to directories and files *are* case sensitive. Mixed case directory and file names are not common. Most use all lowercase letters. The SQL Anywhere installation contains directories such as *bin32* and *documentation*.

  The documentation uses the Windows forms of directory names. You can usually convert a mixed case directory name to lowercase for the equivalent directory name on Unix.

- **Slashes separating directory and file names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in *install-dir\Documentation\en\PDF* (Windows form).

  On Unix, replace the backslash with the forward slash. The PDF documentation is found in *install-dir/documentation/en/pdf*.

- **Executable files** The documentation shows executable file names using Windows conventions, with a suffix such as *.exe* or *.bat*. On Unix, executable file names have no suffix.

  For example, on Windows, the network database server is *dbsrv12.exe*. On Unix, it is *dbsrv12*.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- ***install-dir*** During the installation process, you choose where to install SQL Anywhere. The environment variable SQLANY12 is created and refers to this location. The documentation refers to this location as *install-dir*.

  For example, the documentation may refer to the file *install-dir/readme.txt*. On Windows, this is equivalent to *%SQLANY12%\readme.txt*. On Unix, this is equivalent to *$SQLANY12/readme.txt* or *${SQLANY12}/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY12 environment variable" [*SQL Anywhere Server - Database Administration*].

- ***samples-dir*** During the installation process, you choose where to install the samples included with SQL Anywhere. The environment variable SQLANYSAMP12 is created and refers to this location. The documentation refers to this location as *samples-dir*.

  To open a Windows Explorer window in *samples-dir*, choose **Start** » **Programs** » **SQL Anywhere 12** » **Sample Applications And Projects**.

  For more information about the default location of *samples-dir*, see "SQLANYSAMP12 environment variable" [*SQL Anywhere Server - Database Administration*].

Command prompts and command shell syntax

Most operating systems provide one or more methods of entering commands and parameters using a command shell or command prompt. Windows command prompts include Command Prompt (DOS prompt) and 4NT. Unix command shells include Korn shell and bash. Each shell has features that extend its capabilities beyond simple commands. These features are driven by special characters. The special characters and features vary from one shell to another. Incorrect use of these special characters often results in syntax errors or unexpected behavior.

The documentation provides command line examples in a generic form. If these examples contain characters that the shell considers special, the command may require modification for the specific shell. The modifications are beyond the scope of this documentation, but generally, use quotes around the parameters containing those characters or use an escape character before the special characters.

These are some examples of command line syntax that may vary between platforms:

- **Parentheses and curly braces** Some command line options require a parameter that accepts detailed value specifications in a list. The list is usually enclosed with parentheses or curly braces. The documentation uses parentheses. For example:

  ```
  -x tcpip(host=127.0.0.1)
  ```

  Where parentheses cause syntax problems, substitute curly braces:

  ```
  -x tcpip{host=127.0.0.1}
  ```

  If both forms result in syntax problems, the entire parameter should be enclosed in quotes as required by the shell:

  ```
  -x "tcpip(host=127.0.0.1)"
  ```

- **Semicolons** On Unix, semicolons should be enclosed in quotes.
- **Quotes** If you must specify quotes in a parameter value, the quotes may conflict with the traditional use of quotes to enclose the parameter. For example, to specify an encryption key whose value contains double-quotes, you might have to enclose the key in quotes and then escape the embedded quote:

  ```
  -ek "my \"secret\" key"
  ```

  In many shells, the value of the key would be `my "secret" key`.
- **Environment variables** The documentation refers to setting environment variables. In Windows shells, environment variables are specified using the syntax *%ENVVAR%*. In Unix shells, environment variables are specified using the syntax *$ENVVAR* or *${ENVVAR}*.

Contacting the documentation team

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

You can leave comments directly on help topics using DocCommentXchange. DocCommentXchange (DCX) is a community for accessing and discussing SQL Anywhere documentation. Use DocCommentXchange to:

- View documentation
- Check for clarifications users have made to sections of documentation
- Provide suggestions and corrections to improve documentation for all users in future releases

Go to http://dcx.sybase.com.

Finding out more and requesting technical support

Newsgroups

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

When you write to one of these newsgroups, always provide details about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng12 -v**.

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

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

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

For web development issues, see http://groups.google.com/group/sql-anywhere-web-development.

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

Developer Centers

The **SQL Anywhere Tech Corner** gives developers easy access to product technical documentation. You can browse technical white papers, FAQs, tech notes, downloads, techcasts and more to find answers to your questions as well as solutions to many common issues. See http://www.sybase.com/developer/library/sql-anywhere-techcorner.

The following table contains a list of the developer centers available for use on the SQL Anywhere Tech Corner:

Name	URL	Description
SQL Anywhere .NET Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/microsoft-net	Get started and get answers to specific questions regarding SQL Anywhere and .NET development.
PHP Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/php	An introduction to using the PHP (PHP Hypertext Preprocessor) scripting language to query your SQL Anywhere database.

Name	URL	Description
SQL Anywhere Windows Mobile Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/windows-mobile	Get started and get answers to specific questions regarding SQL Anywhere and Windows Mobile development.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Introducing SQL Remote

SQL Remote is a message-based technology designed for the two-way replication of database transactions between a consolidated database and large numbers of remote databases. Administration and resource requirements at the remote sites are minimal, making SQL Remote well suited to mobile devices.

SQL Remote provides the following functionality:

- **Multiple subscriber support** SQL Remote allows occasionally connected users to replicate data between a SQL Anywhere consolidated database and a large number of remote SQL Anywhere databases, typically including many mobile databases.
- **Transaction log-based replication** SQL Remote uses the transaction log for replication. As a result, only changed data is replicated during an update. It ensures proper transaction atomicity throughout the replication system and maintains consistency among the databases involved in the replication.
- **Central administration** SQL Remote is centrally administered at the consolidated database. A company can have a large mobile workforce with many unique databases without maintaining each remote database individually. In addition, SQL Remote operation is invisible to the end user.
- **Economical memory use** To run efficiently, SQL Remote uses memory economically. This allows you to use SQL Remote on existing remote computers and devices without having to invest in new hardware. Replication is possible to and from remote computers and devices with limited space; only relevant data is replicated from the consolidated database to the remote databases.
- **Multi-platform support** SQL Remote is supported on several operating systems and message links. SQL Anywhere databases can be copied from one file or operating system to another. See http://www.sybase.com/detail?id=1061806.

Typical SQL Remote setups

SQL Remote is designed for replication systems with the following requirements:

- **Large numbers of remote databases** SQL Remote can support thousands of remote databases in a single installation because the messages for many remote databases can be prepared simultaneously.
- **Occasionally connected** SQL Remote supports databases that are occasionally connected or indirectly connected to the network. SQL Remote is not designed for instantaneous data availability at each site. For example, it may use an SMTP email system to carry the replication.
- **Low to high latency** High latency means a long lag time between data being entered at one database and being replicated to each database in the system. With SQL Remote, replication messages can be sent at intervals of seconds, minutes, hours, or days.
- **Low to moderate volume** As replication messages are delivered occasionally, a high transaction volume at each remote database can lead to a large volume of messages. SQL Remote is best suited to

systems with a relatively low volume of replicated data per remote database. At the consolidated database, SQL Remote can prepare messages for multiple databases simultaneously.

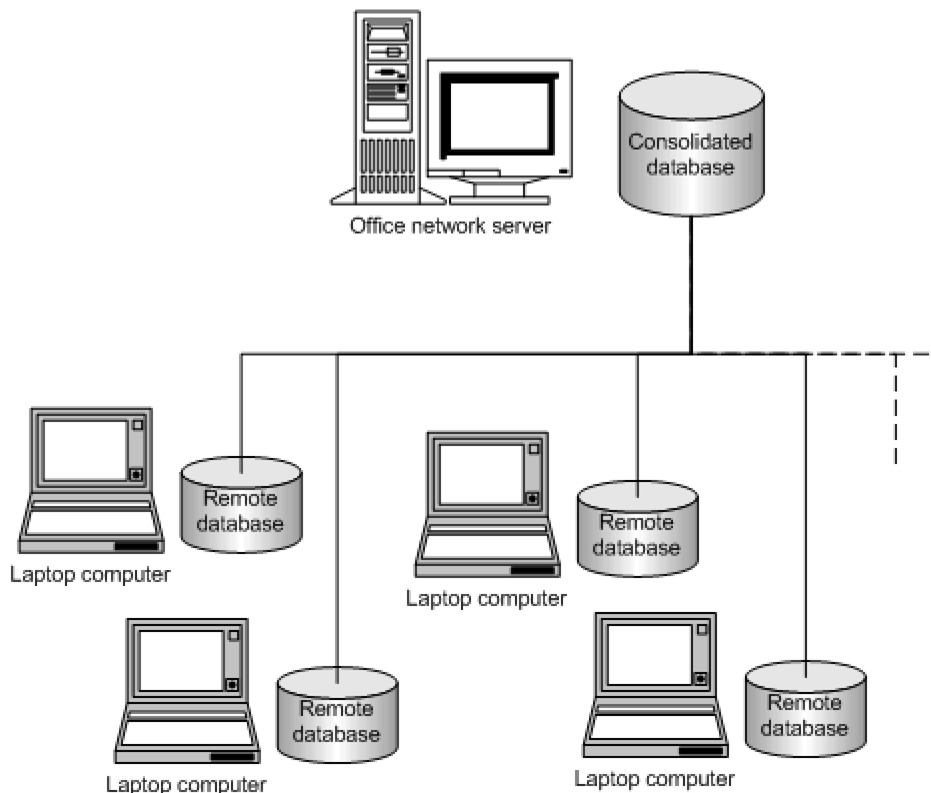
- **Homogeneous databases** Each SQL Anywhere database in the system must have a similar schema.

See also

- "Comparing synchronization technologies" [*SQL Anywhere 12 - Introduction*]
- "Choosing a synchronization technology" [*SQL Anywhere 12 - Introduction*]

Server-to-remote database replication for mobile workforces

In the following example, SQL Remote provides two-way replication between a consolidated database on an office network, and personal databases on the laptop computers of sales representatives. An SMTP email system is used as a message transport.



To manage the consolidated database, the office network server runs a SQL Anywhere database server. SQL Remote connects to the consolidated database in the same way as any other client application.

Each sales representative's laptop computer includes a SQL Anywhere personal server, a SQL Anywhere remote database, and SQL Remote.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

While away from the office, a sales representative can connect to the internet to run SQL Remote, which carries out the following functions:

- Collects publication updates from the consolidated database on the office network server.
- Submits any local updates, such as new orders, to the consolidated database on the office network server.

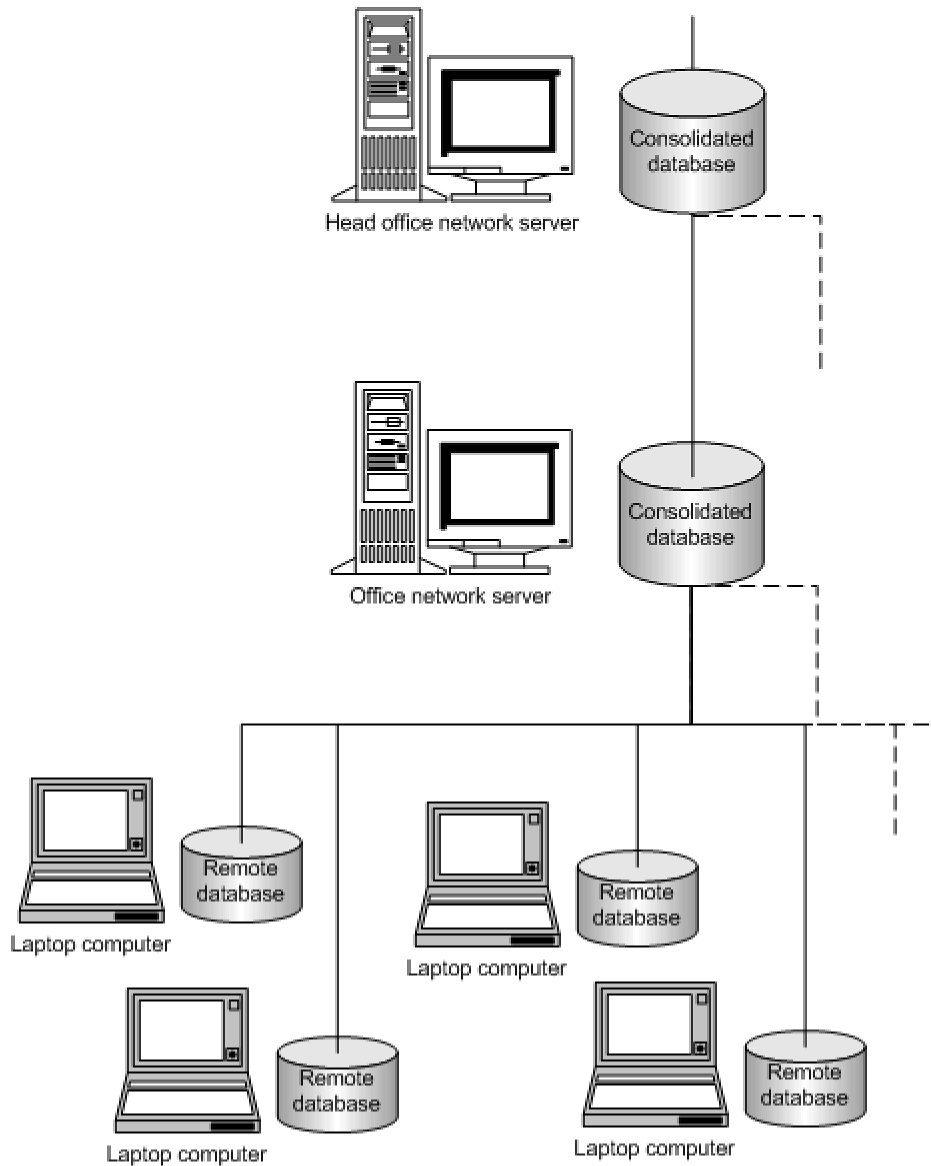
The publication updates from the office network database may include new specials on the products the sales representative handles, or new pricing and inventory information. These updates are read by SQL Remote on the laptop and applied to the sales representative's remote database automatically, without requiring any additional action from the sales representative.

The new orders recorded by the sales representative are also automatically submitted and applied to the office network database without any extra action from the sales representative.

Server-to-server database replication among offices

In this example, SQL Remote provides two-way replication between the database servers at the sales offices or outlets, and the central company office. The only work required at the sales offices is the initial setup and ongoing maintenance of the server.

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.

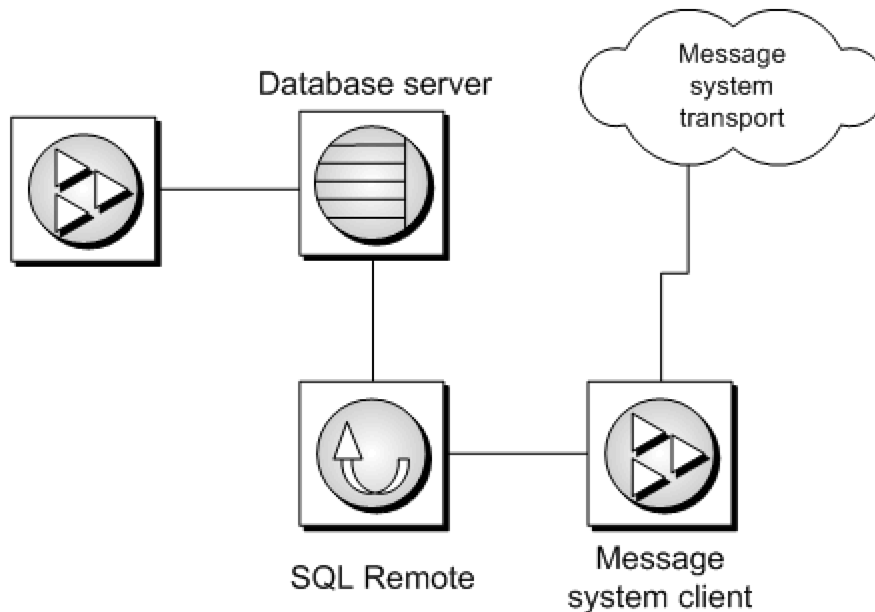


SQL Remote can be configured to allow each office to receive its own set of data. Tables such as staff records can be kept private in the same database as the replicated data.

SQL Remote components

The following components are required for SQL Remote:

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0



- **Database server** A SQL Anywhere database is required at the consolidated and each remote site.
- **SQL Remote** To send and receive replication messages from database to database, SQL Remote must be installed at the consolidated site and the remote sites.

  The SQL Remote Message Agent connects to the database server via a client/server connection. The SQL Remote Message Agent may run on the same computer as the database server or on a different computer.
- **Message system client software** SQL Remote uses existing message systems to transport replication messages.

  If you are using a shared file or FTP message system, the message system is included with your operating system.

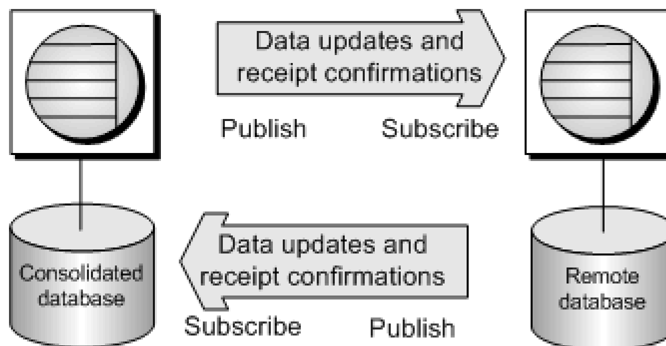
  If you are using an SMTP email system, you must have an email client installed at the consolidated and each remote site.
- **Client applications** The client application can use ODBC, embedded SQL, or several other programming interfaces. Client applications do not have to know if they are using a consolidated or

remote database. From the client application perspective, there is no difference. For specific details about the SQL Anywhere programming interfaces, see the list below:

- "SQL Anywhere .NET support" [*SQL Anywhere Server - Programming*]
- "ODBC support" [*SQL Anywhere Server - Programming*]
- "OLE DB and ADO development" [*SQL Anywhere Server - Programming*]
- "Embedded SQL" [*SQL Anywhere Server - Programming*]
- "JDBC support" [*SQL Anywhere Server - Programming*]
- "Sybase Open Client support" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere C API support" [*SQL Anywhere Server - Programming*]
- "Perl DBI support" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere PHP extension" [*SQL Anywhere Server - Programming*]
- "Python support" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere Ruby API support" [*SQL Anywhere Server - Programming*]

Understanding the SQL Remote replication process

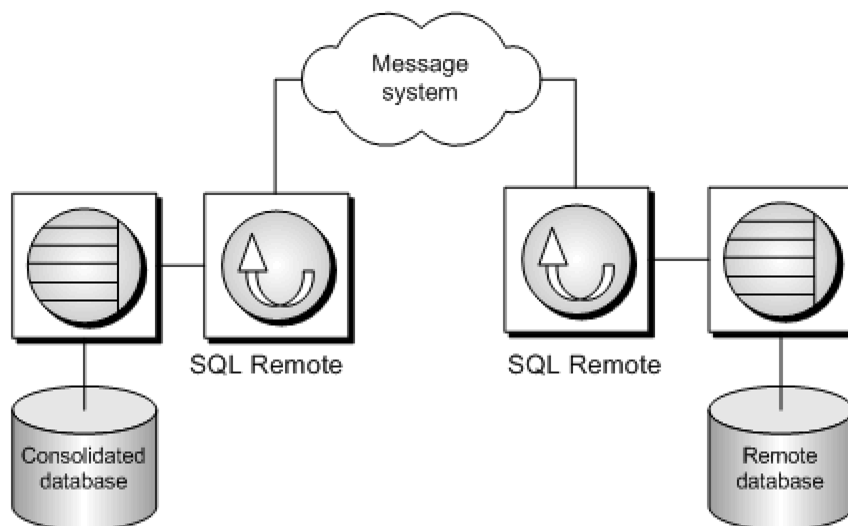
With SQL Remote, messages are always sent two ways. The consolidated database sends messages containing publication updates to remote databases, and remote databases send updated data and receipt confirmation messages to the consolidated database.



When remote database users modify data, their changes are replicated to the consolidated database. When these changes are applied at the consolidated database, they become part of the consolidated database's publication, and are included with the updates sent to all remote databases (except the one the update came from). In this way, replication from remote database to remote database takes place via the consolidated database.

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

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0



Steps involved in the SQL Remote replication process

1. At each consolidated and remote database participating in replication, there is a message agent and a transaction log that manages replication. All committed changes are recorded and stored in the transaction log.

2. Periodically, the SQL Remote Message Agent on the consolidated database scans the transaction log and packages all the committed transactions made to each publication (section of data) into messages. The consolidated database's SQL Remote Message Agent then sends the relevant changes to remote users who subscribe to those publications. The SQL Remote Message Agent sends the changes using a messaging system. SQL Remote supports SMTP email systems, FTP, and FILE.

3. The SQL Remote Message Agent at the remote database accepts the messages sent from the consolidated database and sends a confirmation to the consolidated database that the messages have been received. Then, the SQL Remote Message Agent applies the transactions to the remote database.

4. At any time, a remote user can run the SQL Remote Message Agent to package the transactions made at the remote database into messages and send them back to the consolidated database.

5. The SQL Remote Message Agent at the consolidated site processes the messages from the remote database and applies the transactions to the consolidated database.

See also

- “Creating SQL Remote systems” on page 9
- “Managing SQL Remote systems” on page 71

Creating SQL Remote systems

Use the consolidated database to complete all SQL Remote administrative tasks.

To create a SQL Remote system

1. Choose your SQL Anywhere consolidated database or create a new SQL Anywhere database. The remote databases, which are also SQL Anywhere databases, are created from the consolidated database.

   When creating a new SQL Anywhere database, consider how SQL Remote uses primary keys. For example, a good practice is to choose BIGINT with global autoincrement for the primary key column data type. See "Duplicate primary key errors" on page 50.

2. Determine what data to replicate.

   When creating an efficient replication system, you decide on the tables that you want to use, the columns from those tables, and finally the subset of rows to replicate. Only include the information that is needed.

3. Create publications on the consolidated database.

   SQL Remote uses a publish and subscribe model to ensure that the correct information reaches its intended user. Arrange the data that you want to replicate into publications on the consolidated database. See "Publications and articles" on page 10.

4. Create a publisher user on the consolidated database.

   A publisher is a user with PUBLISH authority and the publisher is used to uniquely identify the consolidated database. See "PUBLISH permission" on page 20.

5. Create the remote user.

   A remote user is used to uniquely identify a remote database. See "REMOTE permission" on page 23.

   When you create a remote user, you define the message type to use when transporting the data and, optionally, you define how frequently to send the data.

6. Subscribe the remote users to publications by creating subscriptions. See "Subscriptions" on page 30.

7. Determine how the remote users can use the data.

   Remote users can always read their data. You can also allow them to update, delete, and insert data. See "Understanding transaction log-based replication" on page 31.

8. Choose a method for resolving conflicts.

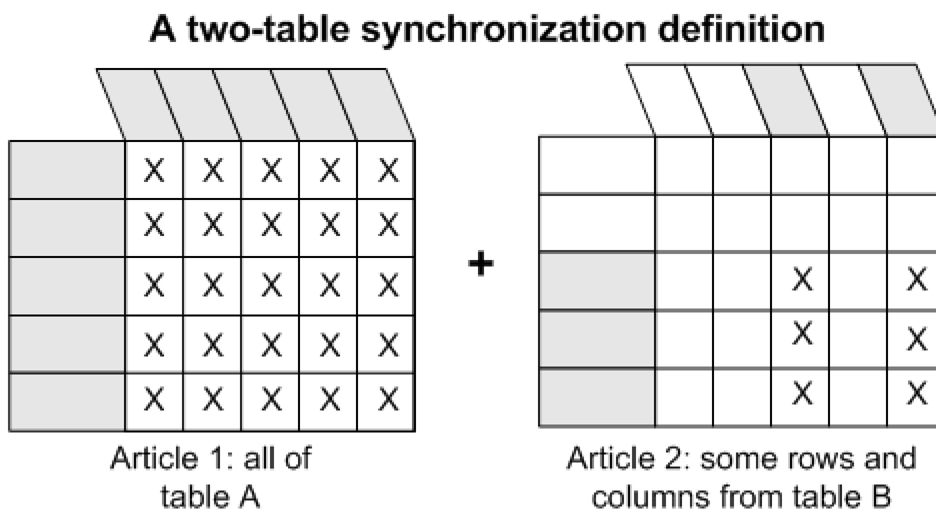
   Conflicts can occur during replication when your remote users update, delete, or insert data. You must implement methods for resolving conflicts. See "Default resolution for update conflicts" on page 41.

9. Deploy the SQL Remote system.

   Create the remote databases and install the appropriate software. See "Managing SQL Remote systems" on page 71.

Publications and articles

A **publication** defines the set of data to be replicated. A publication can include data from several database tables. An **article** refers to a table in a publication. Each article in a publication can consist of the entire table or a subset of the rows and columns in the table.



Limitations

A publication cannot include views or stored procedures. For information about how SQL Remote replicates procedures and triggers, see "Replicating procedures" on page 35 and "Replicating triggers" on page 35.

Viewing publications and articles (Sybase Central)

In Sybase Central, publications appear in the **Publications** folder in the left pane. Any articles you create for a publication appear on the **Articles** tab in the right pane when you select a publication.

Create publications

You create publications based on existing tables in the consolidated database.

Use the following procedures to create publications that consist of all the columns and rows in a table.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

To publish tables (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.
2. In the left pane, select the **Publications** folder.
3. Choose **File** » **New** » **Publication**.
4. In the **What Do You Want To Name The New Publication** field, type a name for the publication. Click **Next**.
5. Click **Next**.
6. In the **Available Tables** list, select a table. Click **Add**.
7. Click **Finish**.

To publish tables (SQL)

1. Connect to the consolidated 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. See "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example, the following statement creates a publication that publishes the entire Customers table:

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

   The following statement creates a publication that publishes the entire SalesOrders, SalesOrderItems, and Products tables:

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

Publish only some columns in a table

Use the following procedures to create a publication that contains all the rows, but only some of the columns, of a table.

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

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.
2. In the left pane, expand the **Publications** folder.
3. Choose **File** » **New** » **Publication**.

4. In the **What Do You Want To Name The New Publication** field, type a name for the publication. Click **Next**.
5. Click **Next**.
6. On the **Available Tables** list, select a table. Click **Add**. Click **Next**.
7. On the **Available 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**. Click **Next**.
8. Click **Finish**.

To publish only some columns in a table (SQL)

1. Connect to the consolidated 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 parentheses following the table name.

   For example, the following statement creates a publication that publishes all the rows of the ID, CompanyName, and City columns of the Customers table. This publication does *not* publish the Surname, GivenName, Street, State, Country, PostalCode, and Phone columns of the Customers table.

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

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Referential integrity errors" on page 47

Publish only some rows in a table

To create a publication that contains only some of the rows in a table, you must write a search condition that matches only the rows you want to publish. Use of one of the following clauses in your search condition:

- **SUBSCRIBE BY clause** Use the SUBSCRIBE BY clause when multiple subscribers to a publication receive different rows from a table.

  The SUBSCRIBE BY clause is recommended when your SQL Remote system requires a large number of subscriptions. The SUBSCRIBE BY clause allows many subscriptions to be associated with a single publication, whereas the WHERE clause does not. Subscribers receive rows depending on the value of a supplied expression.

  Publications using a SUBSCRIBE BY clause are more compact, easier to understand, and provide better performance than maintaining several WHERE clause publications.

See "Publish only some rows using the SUBSCRIBE BY clause" on page 14.

- **WHERE clause** 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.

  All unpublished rows must have a default value. Otherwise, when the remote database tries to insert new rows from the consolidated database, an error occurs.

  You can combine a WHERE clause in an article.

  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 WHERE clause does not allow many subscriptions to be associated with a single publication; however the SUBSCRIBE BY clause does.

  See "Publish only some rows using a WHERE clause" on page 15.

Example

You need a publication that enables each sales representative to:

- Subscribe to their sales orders.
- Update their sales orders locally.
- Replicate their sales to the consolidated database.

If you use the WHERE clause, you would need to create separate publications for each sales representative. The following publication is for a sales representative named Sam Singer; each of the other sales representatives would need a similar publication.

```
CREATE PUBLICATION PubOrdersSamSinger (
   TABLE SalesOrders
      WHERE Active = 1
);
```

The following statement subscribes Sam Singer to the PubsOrdersSamSinger publication.

```
CREATE SUBSCRIPTION
TO PubOrdersSamSinger
FOR Sam_Singer;
```

If you use the SUBSCRIBE BY clause, you need only one publication. All of the sales representatives can use the following publication:

```
CREATE PUBLICATION PubOrders (
   TABLE SalesOrders
   SUBSCRIBE BY SalesRepresentativeID
);
```

The following statement subscribes Sam Singer to the PubsOrders publication by his ID, 8887.

```
CREATE SUBSCRIPTION
TO PubOrders ('8887')
FOR Sam_Singer;
```

Publish only some rows using the SUBSCRIBE BY clause

Use the following procedure to create a publication using the SUBSCRIBE BY clause. For information about using the SUBSCRIBE BY clause and its alternative the WHERE clause, see "Publish only some rows in a table" on page 12.

To create a publication using the SUBSCRIBE BY clause (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.
2. In the left pane, select the **Publications** folder.
3. Choose **File** » **New** » **Publication**.
4. In the **What Do You Want To Name The New Publication** field, type a name for the publication. Click **Next**.
5. Click **Next**.
6. In the **Available Tables** list, select a table. Click **Add**. Click **Next**.
7. On the **Available 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**. Click **Next**.
8. Click **Next**.
9. On the **Specify SUBSCRIBE BY Restrictions** page:
   a. Click a table in the **Articles** list.
   b. Click **Column** and choose a column from the dropdown list.
10. Click **Finish**.

To create a publication using the SUBSCRIBE BY clause (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE PUBLICATION statement that includes a SUBSCRIBE BY clause.

Examples

The following statement creates a publication that publishes the ID, CompanyName, City, State, and Country columns of the Customers table and to match the rows with subscribers uses the value of the State column:

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

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

```
    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

- "Publish only some rows using a WHERE clause" on page 15
- "Using disjoint data partitions" on page 56
- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Subscribe to a publication" on page 30

Publish only some rows using a WHERE clause

Use the following procedures to create a publication that uses a WHERE clause to include all the columns, but only some of the rows of a table. For information about using the WHERE clause and its alternative the SUBSCRIBE BY clause, see "Publish only some rows in a table" on page 12.

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

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.

2. In the left pane, select the **Publications** folder.

3. Choose **File** » **New** » **Publication**.

4. In the **What Do You Want To Name The New Publication** field, type a name for the publication. Click **Next**.

5. Click **Next**.

6. In the **Available Tables** list, select a table. Click **Add**. Click **Next**.

7. On the **Available 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**. Click **Next**.

8. On the **Specify WHERE Clauses** page:

   a. Click a table in the **Articles** list.

b. Type a WHERE clause into the **The Selected Article Has The Following WHERE Clause** field.

9. 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 uses a WHERE clause to include the rows you want to include in the publication.

   For example, the following statement creates a publication that publishes the ID, CompanyName, City, State, and Country columns of the Customers table, for customers marked as active in the Status column. The Status column is not published.

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

   The following statements subscribe two employees to the same publication. Both Ann Taylor and Sam Singer receive the same data.

```
CREATE SUBSCRIPTION
TO PubCustomers
FOR Ann_Taylor;

CREATE SUBSCRIPTION
TO PubCustomers)
FOR Sam_Singer;
```

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

See “CREATE PUBLICATION statement [MobiLink] [SQL Remote]” [*SQL Anywhere Server - SQL Reference*].

See also

- “WHERE clause and primary keys” on page 47

Alter publications

You alter a publication by adding, modifying, deleting articles, or by renaming the publication.

> **Caution**
> Altering publications in a running SQL Remote system can cause replication errors and a loss of data in the replication system. See “Upgrading and resynchronization” on page 122.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

To alter a publication (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user who owns the publication or as a user with DBA authority.
2. In the left pane, select the **Publications** folder.
3. Right-click the article you want to alter and choose **Properties** to edit the publication.

To alter a 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.

   For example, the following statement adds the Customers table to the PubContacts publication.

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

   For example, the following statement redefines the Customers article in the PubContacts publication.

   ```
   ALTER PUBLICATION PubContacts (
      ALTER ARTICLE Customers ( name, surname )
   );
   ```

See also

- "ALTER PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Drop a publication

When you drop a publication, all subscriptions to that publication are automatically deleted.

> **Caution**
> Dropping publications in a running SQL Remote system can cause replication errors and a loss of data in the replication system. See "Upgrading and resynchronization" on page 122.

To delete a publication (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, expand the **Publications** folder.
3. Right-click the desired publication and choose **Delete**.

To delete a publication (SQL)

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

   For example, the following statement drops the publication named PubOrders.

   ```
   DROP PUBLICATION PubOrders;
   ```

   See "DROP PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

User permissions

SQL Remote uses a consistent system to manage the users who have permissions on remote and consolidated databases.

Users of databases involved in SQL Remote replication are identified by one or more of the following permissions:

- **PUBLISH** Every database in a SQL Remote system publishes information. Therefore, every database must have a publisher. To create a publisher, grant one user PUBLISH permission. The publisher user must be unique throughout the SQL Remote system. When sending data, the publisher represents the database. For example, when a database sends a message, its publisher user name is included with the message. When a database receives a message, it can identify the database that sent the message by the publisher name in the message.
- **REMOTE** A database, such as a consolidated database, that sends messages to other databases must specify which remote databases it sends messages to. To specify these remote databases on the consolidated database, grant REMOTE permission to the publishers of the remote databases. REMOTE permission identifies databases that receive messages from the current database.
- **CONSOLIDATE** Each remote database must specify the consolidated database that it receives messages from. To specify a consolidated database on the remote database, grant CONSOLIDATE permission to the publisher of the consolidated database. A remote database can only receive messages from one consolidated database. CONSOLIDATE permission identifies the database that sends messages to the current database.

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

Extraction utility (dbxtract) sets permissions automatically

By default, the Extraction utility (dbxtract) and the **Extract Database Wizard** grant the appropriate PUBLISH and CONSOLIDATE permissions to users in the remote databases.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Single-tiered hierarchy

In a single-tiered hierarchy, there is one consolidated database with one or more remote databases underneath. In such a hierarchy, the consolidated database grants REMOTE permission to the publishers of the remote databases. Each remote database grants CONSOLIDATE permission to the consolidated database.

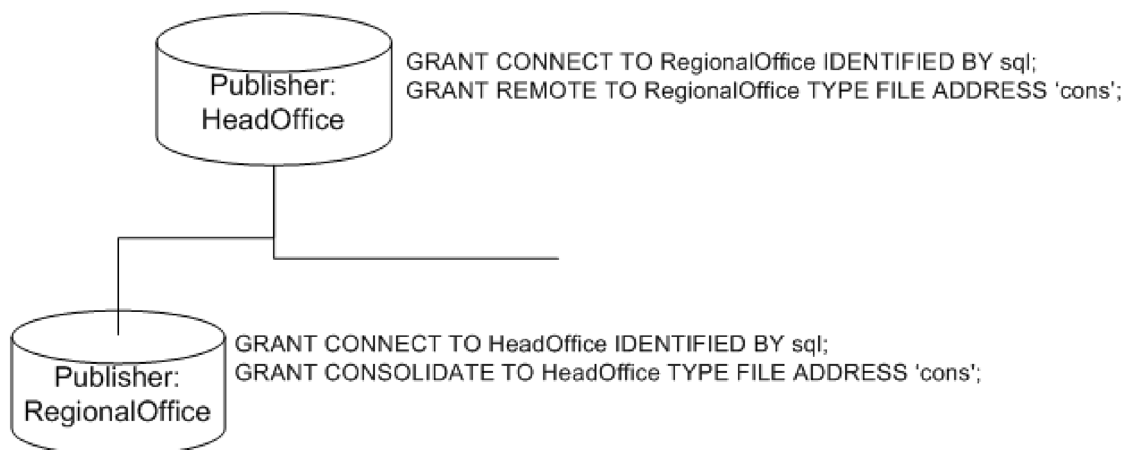
For example, there is a consolidated database identified by its publisher, HeadOffice, and a remote database identified by its publisher, RegionalOffice.

On the consolidated database, HeadOffice, you:

- Create a user with the same name as the publisher of the remote database: RegionalOffice.
- Grant REMOTE permission to RegionalOffice. This identifies RegionalOffice as a remote database of the HeadOffice.

On the remote database, RegionalOffice, you:

- Create a user with the same name as the publisher of the consolidated database: HeadOffice.
- Grant CONSOLIDATE permission to RegionalOffice. This identifies RegionalOffice as a remote database of the HeadOffice.



Dbxtract sets permissions automatically

By default, the Extraction utility (dbxtract) and the **Extract Database Wizard** grant the appropriate PUBLISH and CONSOLIDATE permissions to users in the remote databases.

Multi-tiered hierarchy

In a multi-tier hierarchy, all remote databases immediately below the current database are granted REMOTE permission. The database immediately above the current database in the hierarchy is granted CONSOLIDATE permission.

For example, there is a consolidated database identified by its publisher, HeadOffice, which has a remote database, RegionalOffice. However, the RegionalOffice database also has a remote database: Office.

On the consolidated database, HeadOffice, you:

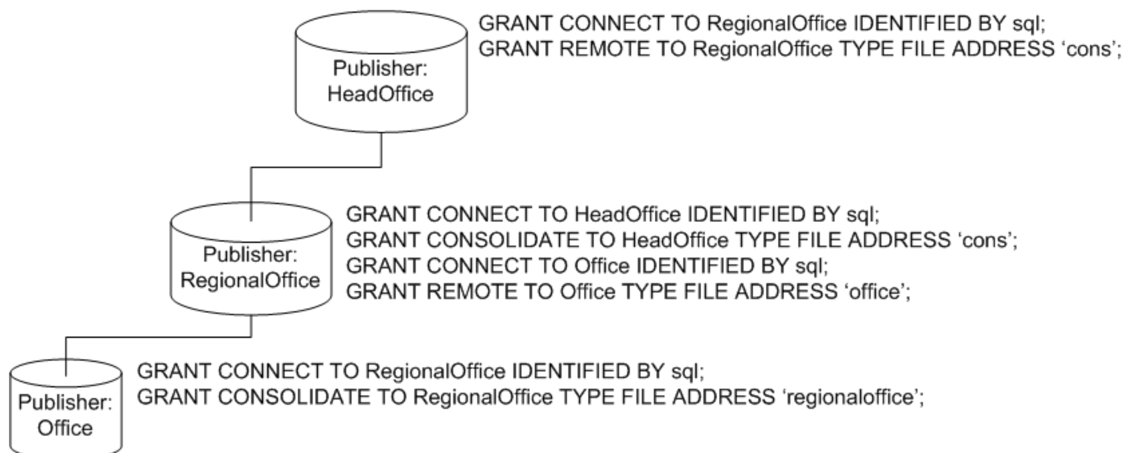
- Create a user with the same name as the publisher of the remote database RegionalOffice.
- Grant REMOTE permission to the user RegionalOffice. This identifies RegionalOffice as a database that receives messages from HeadOffice.

On the RegionalOffice database, you:

- Create a user with the same name as the publisher of the consolidated database HeadOffice.
- Grant CONSOLIDATE permission to HeadOffice. This identifies HeadOffice as the consolidated database for RegionalOffice; that is, HeadOffice is the database that sends messages to RegionalOffice.
- Create a user with the same name as the database immediately below RegionalOffice: Office.
- Grant REMOTE permission to Office. This identifies Office as a database that receives messages from RegionalOffice.

On the Office database, you:

- Create a user with the same name as the publisher of the consolidated database: RegionalOffice.
- Grant Consolidate permission to the RegionalOffice user. This identifies RegionalOffice as the consolidated database for Office; that is RegionalOffice sends messages to Office.



PUBLISH permission

Every database in a SQL Remote system requires a publisher, which is a unique user with PUBLISH permission. All outgoing SQL Remote messages, including publication updates and receipt confirmations, are identified by their publisher. Every database in a SQL Remote system sends receipt confirmations.

PUBLISH permission has no authority except to identify the publisher in outgoing messages.

When PUBLISH permission is granted to a user name with GROUP permission, it is not inherited by members of the group.

You create a publisher by granting a user PUBLISH permission.

Create a publisher

Use the following procedures to create users and grant them PUBLISH permission.

To a publisher (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. Create a user to serve as the publisher, if one doesn't already exist.
   a. In the left pane, select the **Users & Groups** folder.
   b. Choose **File** » **New** » **User**.
   c. Follow the instructions in the **Create User Wizard**. Assign the user a password.
3. In the **Users & Groups** folder, right-click the user you created, and choose **Change To Publisher**.

To create a publisher (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE USER statement to create a user to serve as the publisher. Assign the user a password.

   For example the following creates a user named cons with the password sql:

   ```
   CREATE USER cons IDENTIFIED BY SQL;
   ```

3. Execute a GRANT CONNECT statement to grant CONNECT permission to the user. See "GRANT statement" [*SQL Anywhere Server - SQL Reference*].

   For example, the following statement grants CONNECT permission to the user cons:

   ```
   GRANT CONNECT TO cons IDENTIFIED BY SQL;
   ```

4. Execute a GRANT PUBLISH statement to grant PUBLISH permission to the user. See "GRANT PUBLISH statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example, the following statement grants PUBLISH permission to the user cons:

   ```
   GRANT PUBLISH TO cons;
   ```

Extraction utility (dbxtract)

When a remote database is extracted by the Extraction utility (dbxtract) or the **Extract Database Wizard**, the remote user becomes the publisher of the remote database, and is granted PUBLISH permission.

See also

- "Revoke PUBLISH permission" on page 22
- "View the publisher" on page 22

Revoke PUBLISH permission

Use the following procedure to revoke the PUBLISH permission from a user.

> **Caution**
> Changing the publisher at a remote database or at the consolidated database can cause serious problems for any subscriptions that the database is involved in, including the loss of information. See "Changes to avoid on a running system" on page 123.

To revoke PUBLISH permission (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select the **Users & Groups** folder.
3. Right-click the user who has PUBLISH permission and choose **Revoke Publisher**.

To revoke PUBLISH permission (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a REVOKE PUBLISH statement to revoke the PUBLISH permission from the current publisher. See "REVOKE PUBLISH statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example:

   ```
   REVOKE PUBLISH FROM cons;
   ```

See also

- "PUBLISH permission" on page 20
- "View the publisher" on page 22

View the publisher

To identify the publisher (Sybase Central)

- In the left pane, select the **Users & Groups** folder.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

In the right pane, the publisher is the user whose **Type** is **Publisher**.

To identify the publisher (SQL)

- Use the CURRENT PUBLISHER constant. The following statement retrieves the publisher user name:

```
SELECT CURRENT PUBLISHER;
```

See also

- "PUBLISH permission" on page 20
- "Revoke PUBLISH permission" on page 22

REMOTE permission

Granting REMOTE permission is also referred to as adding a remote user to the database. Publishers of databases directly below the current database in a SQL Remote hierarchy are granted REMOTE permission by the current database.

When granting REMOTE permission to a user, you must configure the following settings:

- **Message system** You cannot create a new remote user until at least one message system is defined in the database. See "SQL Remote message systems" on page 99.

- **Send frequency** When you use SQL statements to grant REMOTE permission, setting the send frequency is optional. See "Setting the send frequency" on page 81.

Granting REMOTE permission to a user:

- Identifies the user as a remote user.

- Specifies a message type to use for exchanging messages with this remote user.

- Provides an address to send messages to.

- Indicates how often messages should be sent to the remote user.

The publisher for a database cannot have REMOTE and CONSOLIDATE permission on the same database. This would identify the publisher as both the sender and recipient of outgoing messages.

Granting REMOTE permission to groups

Although, you can grant REMOTE permission to a group, the REMOTE permission does *not* automatically apply to all users in the group. You must explicitly grant REMOTE permission to each user in the group.

Grant REMOTE permission

Use the following procedures to add a remote user.

To create a remote user (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select the **SQL Remote Users** folder.
3. Choose **File** » **New** » **SQL Remote User**.
4. Follow the instructions in the **Create Remote User Wizard**.

To make an existing user a remote user (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select the **SQL Remote Users** folder.
3. Right-click the user and choose **Change To Remote User**.
4. In the window, select the message type, enter an address, choose the send frequency, and click **OK**.

To create a remote user (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE USER statement to create a user.

   For example:

   ```
   CREATE USER remote1;
   ```

3. Execute a GRANT CONNECT statement to grant the user CONNECT permission. See "GRANT statement" [*SQL Anywhere Server - SQL Reference*].

   For example:

   ```
   GRANT CONNECT TO remote1;
   ```

4. Execute a GRANT REMOTE statement to grant the user REMOTE permission. See "GRANT REMOTE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example:

   ```
   GRANT REMOTE TO userid;
   ```

   For example, the following statement grants REMOTE permission to user S_Beaulieu, and specifies that the remote database:

   - Uses an SMTP email as its message system.
   - Sends messages using the email address s_beaulieu@acme.com:
   - Sends message daily at 10 P.M.

   ```
   GRANT REMOTE TO S_Beaulieu
   TYPE smtp
   ```


Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

```
ADDRESS 's_beaulieu@acme.com'
SEND AT '22:00';
```

The following statement grants REMOTE permission to user rem1, and specifies that rem1's remote database uses a FILE message system with the address rem1.

```
GRANT CONNECT TO rem1 IDENTIFIED BY SQL
GRANT REMOTE TO rem1 TYPE FILE ADDRESS 'rem1';
```

See also

- "Revoke REMOTE permission" on page 25

Revoke REMOTE permission

Revoking a user's REMOTE permission:

- Removes a user from the SQL Remote system.
- Reverts that user or group to a normal user/group.
- Unsubscribes the user or group from all publications.

To revoke REMOTE permission (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.
2. In the left pane, expand either the **Users & Groups** folder or the **SQL Remote Users** folder.
3. Right-click the remote user or group and choose **Revoke Remote**.

To revoke REMOTE permission (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a REVOKE REMOTE statement to revoke the REMOTE permission from the current user or group. See "REVOKE REMOTE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example, the following statement revokes REMOTE permission from user S_Beaulieu.

   ```
   REVOKE REMOTE FROM S_Beaulieu;
   ```

See also

- "Grant REMOTE permission" on page 23

CONSOLIDATE permission

Databases directly above the current database in a SQL Remote hierarchy are granted CONSOLIDATE permission by the current database. At each remote database, the consolidated database must be granted CONSOLIDATE permission.

CONSOLIDATE permission must be granted even from read-only remote databases to the consolidated database, because receipt confirmations are sent from the remote databases to the consolidated database.

When granting CONSOLIDATE permission to a user, you must configure the following settings:

- **Message system** You cannot create a new consolidated user until at least one message system is defined in the database. See "SQL Remote message systems" on page 99.
- **Send frequency** When you use SQL statements to grant CONSOLIDATE permission, setting the send frequency is optional. See "Setting the send frequency" on page 81.

Granting CONSOLIDATE permission:

- Identifies a user as a consolidated user.
- Specifies a message type to use for exchanging messages with this consolidated user.
- Provides an address to send messages to.
- Indicates how often messages should be sent to the consolidated user.

The publisher for a database cannot have REMOTE and CONSOLIDATE permissions on the same database. This would identify the publisher as both the sender and recipient of outgoing messages.

Extraction utility (dbxtract)

When a remote database is extracted by the Extraction utility (dbxtract) or the **Extract Database Wizard**, the GRANT CONSOLIDATE statement is executed automatically on the remote database.

Grant CONSOLIDATE permission

Use the following procedures to grant CONSOLIDATE permission to a user.

It is recommended that you grant CONSOLIDATE permission to the publisher of the consolidated database.

To specify a consolidated database (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, click the database, and then choose **File** » **Properties**.
3. Click the **SQL Remote** tab.
4. Select **This Remote Database Has A Corresponding Consolidated Database**.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

5. Configure the **Message Type**, **Address**, and **Send Frequency** settings.

6. Click **OK** to close the **Consolidated Database Properties** window.

To specify a consolidated database (SQL)

1. Execute a GRANT CONNECT statement to grant CONNECT permission to the publisher of the consolidated database. See “GRANT statement” [*SQL Anywhere Server - SQL Reference*].

   For example:

   ```
   GRANT CONNECT TO cons;
   ```

2. Execute a GRANT CONSOLIDATE statement to grant CONSOLIDATE permission to the consolidated user. See “GRANT CONSOLIDATE statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*].

   For example, the following statement grants CONSOLIDATE permission to the hq_user user and specifies that the consolidated database uses the SMTP email system:

   ```
   GRANT CONSOLIDATE TO hq_user
   TYPE SMTP
   ADDRESS 'hq_address';
   ```

   There is no SEND clause in this GRANT CONSOLIDATE statement, so SQL Remote sends messages to the consolidated database and then stops.

   For example, the following statement grants CONSOLIDATE permission to the cons user and specifies that the consolidated database uses the FILE system:

   ```
   GRANT CONNECT TO "cons" IDENTIFIED BY SQL;
   GRANT CONSOLIDATE TO "cons" TYPE "FILE" ADDRESS 'cons';
   GRANT CONNECT TO "rem1" IDENTIFIED BY SQL;
   GRANT PUBLISH TO "rem1";
   CREATE REMOTE MESSAGE TYPE FILE ADDRESS 'rem1';
   ```

See also

- “Revoke CONSOLIDATE permission” on page 27

Revoke CONSOLIDATE permission

When you revoke CONSOLIDATE permission from a user, SQL Anywhere:

- Removes the user from the SQL Remote system.
- Reverts that user or group to a normal user/group.
- Unsubscribes the user or group from all publications.

To revoke CONSOLIDATE permission (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. Expand either the **Users & Groups** folder or the **SQL Remote Users** folder.
3. Right-click the consolidated user or group and choose **Revoke Consolidated**.
4. Click **Yes**.

To revoke CONSOLIDATE permission (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a REVOKE CONSOLIDATE statement to revoke CONSOLIDATE permission from the current user or group. See "REVOKE CONSOLIDATE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   For example:

   ```
   REVOKE CONSOLIDATE FROM Cons_User;
   ```

See also

- "Grant CONSOLIDATE permission" on page 26

Granting REMOTE DBA authority

A user name with REMOTE DBA authority has full DBA authority on the database only when connecting from SQL Remote. With REMOTE DBA authority, SQL Remote has full access to the database and can make changes specified in the messages. Only users with REMOTE DBA or DBA authority can run SQL Remote.

As a special authority, REMOTE DBA, has the following properties:

- **No distinct permissions when not connected from SQL Remote** A user that is granted REMOTE DBA authority has no extra privileges on any connection apart from SQL Remote. Therefore, even if the user name and password for a REMOTE DBA user are widely distributed, there are no security issues. As long as the user name has no permissions beyond CONNECT granted on the database, no one can use this user name to access data in the database.
- **Full DBA permission from SQL Remote** When connecting from SQL Remote, a user with REMOTE DBA authority has full DBA permission on the database.

When to grant REMOTE DBA authority

It is recommended that when you are creating users on the consolidated database that you grant REMOTE DBA authority to the publisher of the consolidated database and to each remote user. When a remote database is extracted by the Extraction utility (dbxtract) or the **Extract Database Wizard**, the remote user

becomes the publisher of the remote database, and is granted PUBLISH permission and REMOTE DBA authority.

This recommendation simplifies the administration of users. Each remote user only needs one user name to connect to the database, whether from the SQL Remote (which provides the user with full DBA authority) or from any other client application (in which case the REMOTE DBA authority grants the user no extra permissions).

See also

- "REMOTE DBA authority" [*SQL Anywhere Server - Database Administration*]
- "GRANT REMOTE DBA statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Grant REMOTE DBA authority

To grant REMOTE DBA authority (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select either the **Users & Groups** folder or the **SQL Remote Users** folder.
3. Right-click the user and choose **Properties**.
4. Click the **Authorities** tab and select the **Remote DBA** option.
5. Click **Apply** and then click **OK**.

To grant REMOTE DBA authority (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a GRANT REMOTE DBA statement to grant REMOTE DBA authority to a user.

   For example:

   ```
   GRANT REMOTE DBA TO dbremote
   IDENTIFIED BY dbremote;
   ```

See also

- "REMOTE DBA authority" [*SQL Anywhere Server - Database Administration*]
- "GRANT REMOTE DBA statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Revoke REMOTE DBA authority

Revoking REMOTE DBA authority removes the user's ability to have full DBA authority on the database when connected from SQL Remote. See "Granting REMOTE DBA authority" on page 28.

Use the following procedure to revoke REMOTE DBA authority from a user.

To revoke REMOTE DBA authority (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select either the **Users & Groups** folder or the **SQL Remote Users** folder.
3. Right-click the user and choose **Properties**.
4. Click the **Authorities** tab and clear the **Remote DBA** option.
5. Click **OK**.

To revoke REMOTE DBA authority (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a REVOKE REMOTE DBA statement to revoke REMOTE DBA authority from a user.

   For example:

   ```
   REVOKE REMOTE DBA FROM dbremote;
   ```

See also

- "REVOKE REMOTE DBA statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Grant REMOTE permission" on page 23

Subscriptions

You subscribe a user to a publication by creating a **subscription**. Each database that shares information in a publication must have a subscription to the publication.

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

Subscribe to a publication

To subscribe a user to a publication, you need the following information:

- **User name** The user who is being subscribed to the publication. This user must have been granted REMOTE permission.
- **Publication name** The name of the publication to which the user is being subscribed.
- **Subscription value** The subscription value only applies if your publication includes a SUBSCRIBE BY clause. The subscription value is the value that is tested against the SUBSCRIBE BY clause of the publication. For example, if a publication has the name of a column containing an employee ID as a SUBSCRIBE BY clause, the value of the employee ID of the subscribing user must be provided when the subscription is created. The subscription value is always a string.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

This value is only needed when the publication has a SUBSCRIBE BY clause. See "Publish only some rows using the SUBSCRIBE BY clause" on page 14.

To subscribe a user to the publication (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select the **Publications** folder.
3. Select a publication.
4. In the right pane, click the **SQL Remote Subscriptions** tab.
5. Choose **File** » **New** » **SQL Remote Subscription**.
6. Follow the instructions in the **Create SQL Remote Subscription Wizard**.

   The details of the subscription are different depending on whether the publication uses a subscription expression.

To subscribe a remote user to a publication (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE SUBSCRIPTION statement to subscribe a user to a publication.

   For example, the following statement creates a subscription for user name SamS to the CustomerPub publication, which was created using a WHERE clause:

   ```
   CREATE SUBSCRIPTION
     TO CustomerPub
     FOR SamS;
   ```

   For example, the following statement creates a subscription for user name SamS to the PubOrders publication, defined with a subscription expression SalesRepresentative, requesting the rows for Sam Singer's own sales:

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

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Publish only some rows using the SUBSCRIBE BY clause" on page 14

Understanding transaction log-based replication

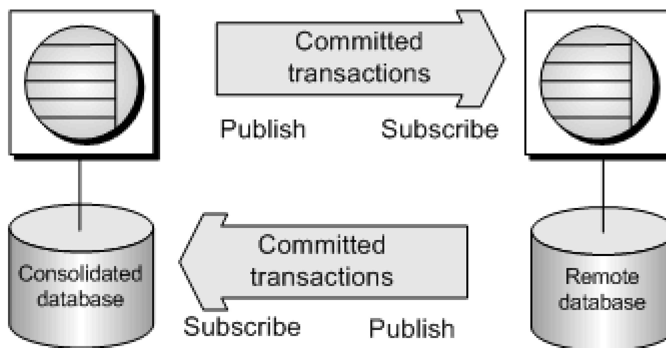
SQL Remote replicates:

- **Committed changes** Changes that have been made to databases as recorded in their transaction log.

- **Changes that modify data that belong to publications** SQL Remote scans the transaction log for committed changes to rows that belong to publications, packages the SQL statements into messages, and sends them to the subscribed databases.

On the consolidated database, all committed transactions in the transaction log that belong to publications are sent periodically to the remote databases.

On the remote databases, all committed transactions in the transaction log that belong to publications are sent periodically to the consolidated database.



The database server handles publications

The SQL Anywhere database server is the component that evaluates the publications and writes the information to the transaction log. The more publications you have, the more work the database server must do.

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 transaction 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 additional information in the transaction 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 several publications, many expressions must be evaluated. In contrast, the number of *subscriptions* is irrelevant to the database server.

- **Many-valued expressions** Some expressions are many-valued, which can lead to additional information in the transaction log. This can affect performance.

Subscriptions are handled by SQL Remote

SQL Remote is the component that carries out the replication of statements.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

During the send phase, the SQL Remote Message Agent maps the current subscriptions to the publication information in the transaction log and generates the appropriate messages for each remote user. See "The SQL Remote Message Agent (dbremote)" on page 84.

See also

- "The transaction log" [*SQL Anywhere Server - Database Administration*]

Replicating INSERT and DELETE statements

For SQL Remote, the INSERT and DELETE statements are the simplest statements to replicate:

- When an INSERT statement is executed on one database, it is sent as an INSERT statement to the subscribed databases in the SQL Remote system.
- When a DELETE statement is executed on one database, it is sent as a DELETE statement to the subscribed databases in the SQL Remote system.

Consolidated database

SQL Remote copies each INSERT or DELETE statement from the consolidated database transaction log, and sends them to each remote database that subscribes to the row being inserted or deleted. When only a subset of the columns in the table is subscribed to, the INSERT statements sent to the remote databases contain only those columns.

Remote databases

SQL Remote copies each INSERT or DELETE statement from a remote database transaction log and sends it to the consolidated database that subscribes to the row being inserted or deleted. The consolidated database then applies the statement, which results in writing to its transaction log. When the consolidated database transaction log is processed by SQL Remote, the changes are eventually propagated to the other remote sites. SQL Remote ensures that statements are not sent to the remote user that initially executed them.

See also

- "Duplicate primary key errors" on page 50
- "Row not found errors" on page 47

Replicating UPDATE statements

UPDATE statements might not be replicated exactly as they are entered in the database. The following scenarios describe how an UPDATE statement is replicated:

- When an UPDATE statement has the effect of updating a row in a given remote user's subscription, it is sent to that user as an UPDATE statement.
- When 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.

- When 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.

To demonstrate how an UPDATE statement can be replicated, the following example uses a consolidated database and three remote databases for the users: Ann, Marc, and ManagerSteve.

Consolidated

ID	Rep	Dept
1	Ann	101
2	Marc	101
3	Marc	101
4	Rob	102

>

Ann

ID	Rep
1	Ann

Marc

ID	Rep
2	Marc
3	Marc

ManagerSteve

ID	Rep
1	Ann
2	Marc
3	Marc

On the consolidated database, there is a publication, named cons, that is created with the following statement:

```
CREATE PUBLICATION "cons"."p1" (
 TABLE "DBA"."customers" ( "ID", "Rep") SUBSCRIBE BY repid
);
```

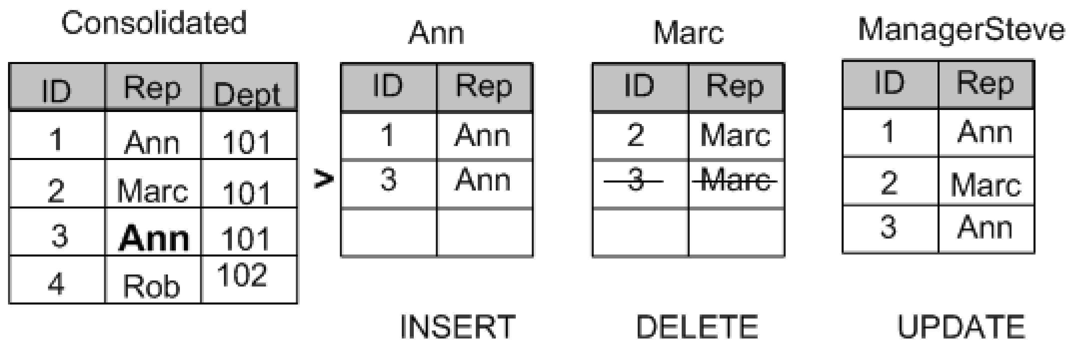
Ann and Marc subscribe to the cons publication by their respective Rep column values. ManagerSteve subscribes to the cons publication with both Ann and Marc's Rep column values. The following statements subscribe the three users to the publication cons:

```
CREATE SUBSCRIPTION
  TO "cons"."p1"( 'Ann' )
  FOR "Ann";
CREATE SUBSCRIPTION
  TO "cons"."p1"( 'Marc' )
  FOR "Marc";
CREATE SUBSCRIPTION
  TO "cons"."p1"( 'Ann' )
  FOR "ManagerSteve";
CREATE SUBSCRIPTION
  TO "cons"."p1"( 'Marc' )
  FOR "ManagerSteve";
```

On the consolidated database, an UPDATE statement that changes the Rep value of a row from Marc to Ann is replicated to:

- Marc as a DELETE statement.
- Ann as an INSERT statement.
- ManagerSteve as an UPDATE statement.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0



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.

See also

- "Update conflicts" on page 40

Replicating procedures

SQL Remote replicates procedures by replicating the actions of a procedure. The procedure call is not replicated. Instead, the individual actions (INSERT, UPDATE, and DELETE statements) of the procedure are replicated.

Replicating triggers

Typically, remote databases have the same triggers defined as the consolidated database does.

By default, SQL Remote does not replicate the actions performed by the triggers. Instead, when an action that fires a trigger on the consolidated database is replicated on the remote database, the duplicate trigger is automatically fired on the remote database. This avoids permissions issues and the possibility of each action occurring twice. There are some exceptions to this rule:

- **Replication of RESOLVE UPDATE triggers** The actions carried out by conflict resolution, or RESOLVE UPDATE triggers *are* replicated from the consolidated database to all remote databases, including the remote database that sent the message that created the conflict. See "Default resolution for update conflicts" on page 41.

- **Replication of BEFORE triggers** The actions of a BEFORE trigger that modifies a row being updated *are* replicated before the UPDATE statement actions.

  For example, a BEFORE UPDATE trigger that increases 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 fires on the remote database when the UPDATE statement is replicated.

A BEFORE UPDATE trigger that sets a column to the time of the last update also receives the time the UPDATE statement is replicated.

To prevent this problem, you must ensure that, at the subscriber database, the BEFORE UPDATE trigger is not present or does not perform the replicated action.

An option to replicate trigger actions

To replicate all trigger actions when sending messages, use the -t option for the SQL Remote Message Agent (dbremote). See "SQL Remote Message Agent utility (dbremote)" on page 131.

When you use the -t option, ensure that the trigger actions are not carried out twice at remote databases, once when the replicated trigger actions are applied, and once when the trigger is fired on the remote database.

To ensure that trigger actions are not carried out twice, use one of the following options:

- Wrap an IF CURRENT REMOTE USER IS NULL ... END IF statement around the body of the trigger.
- Set the SQL Anywhere fire_triggers option to Off for the SQL Remote user name. See "fire_triggers option" [*SQL Anywhere Server - Database Administration*].

Avoiding trigger errors

If a publication includes only a subset of a database, a trigger at the consolidated database can refer to tables or rows that are present at the consolidated database, but are not present on the remote databases. When such a trigger is fired on the remote database, errors occur. To avoid these errors, use an IF statement to make the trigger actions conditional, and:

- Have the actions of the trigger be conditional on the value of CURRENT PUBLISHER. See "CURRENT PUBLISHER special value" [*SQL Anywhere Server - SQL Reference*].
- Have the 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. See "System functions" [*SQL Anywhere Server - SQL Reference*].
- Have actions of the trigger be conditional on a SELECT statement that determines if the rows exist.

Extraction utility (dbxtract)

By default, the database Extraction utility (dbxtract) and the **Extract Database Wizard** extract the trigger definitions.

See also

- "Using the CURRENT REMOTE USER special value" on page 44

Data definition statements

Data definition statements (CREATE, ALTER, and DROP) are not replicated by SQL Remote unless they are executed in passthrough mode.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

See "SQL Remote passthrough mode" on page 124.

Data types

SQL Remote does not perform any character set conversions.

Use compatible sort orders and character sets

The character set and collation used by the SQL Anywhere consolidated database must be the same as the remote database's. For information about supported character sets, see "International languages and character sets" [*SQL Anywhere Server - Database Administration*].

BLOBs

BLOBs include the LONG VARCHAR, LONG BINARY, TEXT, and IMAGE data types.

When SQL Remote replicates an INSERT or UPDATE statement, it uses a variable in place of the BLOB value. That is, the BLOB is broken into pieces and replicated in chunks. At the recipient database, the pieces are reconstituted by using a SQL variable and concatenated. The value of the variable is built up by a sequence of statements of the form:

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

The variable makes the size of the SQL statements involving long values smaller, so 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.

Controlling replication of BLOBs

The SQL Anywhere blob_threshold option allows further control over the replication of long values. Any value that is longer than the blob_threshold option is replicated as though it is a BLOB value. See "blob_threshold option [SQL Remote]" [*SQL Anywhere Server - Database Administration*].

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 statement). The default value for the option is 1000. When the data type of a column is longer than the threshold, old values for the column are not verified when an UPDATE statement is replicated. This reduces the size of SQL Remote messages, but has the disadvantage that conflicting updates of long values are not detected.

Use the following technique to detect conflicts when the verify_threshold is being used to reduce the size of messages.

To detect conflicts with BLOB values when the verify_threshold is set

1. Configure your databases so that whenever a BLOB is updated, a last_modified column in the same table is also updated.
2. Configure your publications so that the last_modified column is replicated with the BLOB column.

   When the BLOB column and the last_modified column are replicated, the values in the last_modified column can be verified. If there is a conflict with the last_modified column, then there is a conflict with the BLOB column as well.

See "verify_threshold option [SQL Remote]" [*SQL Anywhere Server - Database Administration*].

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 SQL Remote is run once at the end of the day, all 20 updates are replicated. If the document is 200 KB in length, then 4 MB of messages are sent.

It is recommended that you use 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. This results in a single update (200 KB of messages).

See also

- "SET statement" [*SQL Anywhere Server - SQL Reference*]

Dates and times

When date or time columns are replicated, SQL Remote 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 SQL Remote to send a date of May 2, 1998 as 1998-05-02.

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

When replicating dates and times:

- The time, date, and timestamp formats must be consistent throughout the SQL Remote system.
- The order of the year, month, and day used for the date and timestamp formats must match the setting of the date_order database option.
- The date_order option for the duration of each connection can be changed.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

See also

- "sr_date_format option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]
- "sr_time_format option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]
- "sr_timestamp_format [SQL Remote]" [*SQL Anywhere Server - Database Administration*]
- "date_order option" [*SQL Anywhere Server - Database Administration*]

Replication conflicts and errors

SQL Remote allows databases to be updated at multiple databases. Careful design is required to avoid replication errors, especially when the database has a complicated structure.

Replication conflicts

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

Conflicts occur in many systems. SQL Remote allows appropriate resolution of conflicts as part of the regular operation of a SQL Remote system, using triggers and procedures. See "Default resolution for update conflicts" on page 41.

Replication errors

Replication errors fall into the following categories:

- **Row not found errors** A user deletes a 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. See "Row not found errors" on page 47.

- **Referential integrity errors** When a column containing a foreign key is included in a publication, but the associated primary key is not included, INSERT statements that reference the foreign key fail.

  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.

  See "Referential integrity errors" on page 47.

- **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. See "Duplicate primary key errors" on page 50.

Delivery errors

For information about delivery errors and how they are handled, see "Understanding the Guaranteed Message Delivery System" on page 93.

See also

- "Reporting and handling replication errors" on page 117

Update conflicts

Update conflicts cannot happen when data is shared for reading, or when each row (as identified by its primary key) is updated at only one database. Update conflicts only occur when data is updated at more than one database.

To replicate UPDATE statements, SQL Remote issues a separate UPDATE statement for each row. These single-row statements can fail for one of the following reasons:

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

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

  On the consolidated database, SQL Remote allows **conflict resolution** operations to take place. For example, when a conflict is detected, the consolidated database can:

  - Use the default conflict resolution. See "Default resolution for update conflicts" on page 41.

  - Use a customized conflict resolution that uses the VERIFY clause. See "Custom conflict resolution using a VERIFY clause" on page 41.

  - Use a customized conflict resolution that uses triggers. See "Custom conflict resolution using triggers" on page 43.

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

- **The row to be updated does not exist** Each row is identified by its primary key values. If the row has been deleted or if a primary key has been altered by another user, the row to be updated cannot be found.

  On remote databases, the update does not occur.

  On the consolidated database, the update does not occur.

- **A table without a primary key or unique constraint refers to all columns in the WHERE clause of replicated updates** When two remote databases make separate updates to the same row and replicate the changes to the consolidated database, the first changes to arrive on the consolidated database are applied; changes from the second database are not applied.

  As a result, databases become inconsistent. All replicated tables should have a primary key or a unique constraint and the columns in the constraint should never be updated.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Default resolution for update conflicts

Update conflicts only occur when data is being updated at more than one site. For example:

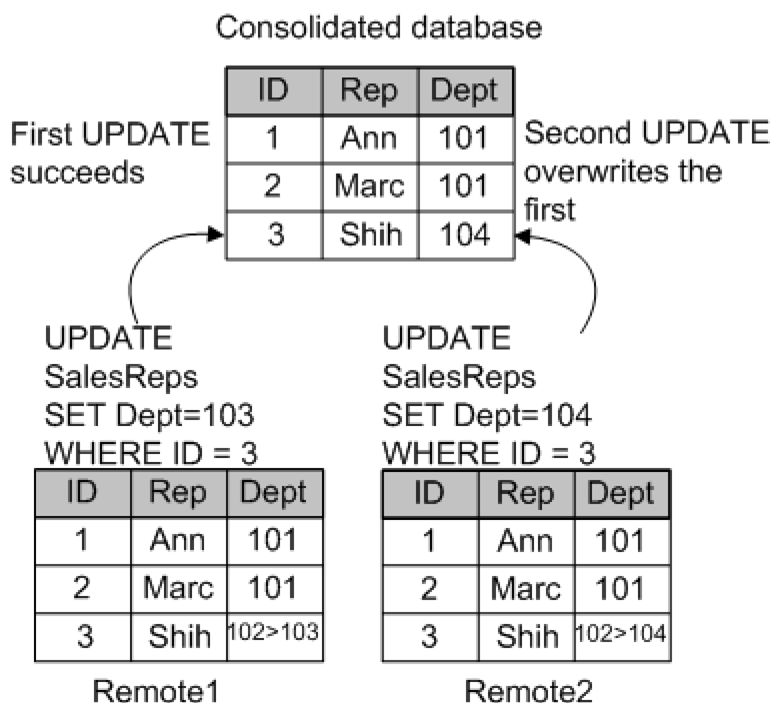
1. User 1 updates a row at remote site 1.
2. User 2 updates the same row at remote site 2.
3. The update from User 1 is sent and applied to the consolidated database.
4. The update from User 2 is sent to the consolidated database.

   The **default** method for resolving this type of update conflict is the following:

   a. The more recent operation (in this example that from User 2) succeeds. It becomes the value in the consolidated database and it is the value that is replicated to all other databases subscribed to that row.

   b. All other updates (in this example that from User 1) are lost.

   c. No report is made of the conflict.



Custom conflict resolution using a VERIFY clause

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

SQL Remote generates UPDATE statements in the messages that use the VERIFY clause. An UPDATE statement changes the value of one or more rows from the existing value to a new value. UPDATE statements that include a VERIFY clause also contain the existing value of the row.

When applying an UPDATE statement, the consolidated database compares its existing value of the row with what the remote database expects the existing value of the row to be. An update conflict is detected by the database server when the VERIFY clause values don't match the rows in the database.

For example, 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 sent and applied to the consolidated database.
4. The update from User 2 is sent to the consolidated database.

   Because the UPDATE statement contains a VERIFY clause, the consolidated database can detect conflicts. On the consolidated database, SQL remote compares the value in its row with the old row value that User 2 sent. As these values are not the same, there is an update conflict.

   When an update conflict is detected, the consolidated database:

   a. Fires any conflict resolution triggers defined for the operation.

      You 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. See "Custom conflict resolution using triggers" on page 43.

   b. Executes the UPDATE statements.

   c. Sends any actions of the conflict resolution trigger and the UPDATE statement to all remote databases, including the sender of the message that triggered the conflict.

      Typically, SQL Remote does not replicate the actions of triggers; the trigger is assumed present on the remote database. Conflict resolution triggers are fired only on consolidated databases, and so their actions are replicated to remote databases.

5. The remote databases receives the UPDATE statements from the consolidated database.

   On remote databases, RESOLVE UPDATE triggers are not fired when a message from a consolidated database contains an update conflict.

6. On the remote database, the UPDATE statements are processed.

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

UPDATE statements with a VERIFY clause

An UPDATE statement with a VERIFY clause takes the following form:

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

UPDATE *table-list*
SET *column-name* = *expression*, ...
[**VERIFY** (*column-name*, ...)
  VALUES (*expression*, ...)]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], ...]

The VERIFY clause can be used only when the *table-list* parameter 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. 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, a multi-row UPDATE statement executed on a database is replicated as a set of single-row UPDATE statements by the SQL Remote, so this imposes no constraints on client applications.

See also

- "UPDATE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Using the verify_all_columns option

By default, the database option verify_all_columns is Off. When it is set to Off, only those columns that are updated are checked. When the verify_all_columns option is set to On, then:

- All columns are verified on replicated UPDATE statements.
- A RESOLVE UPDATE trigger is fired whenever any column is different.
- The size of messages is bigger, because more information is sent for each UPDATE statement.

You can set the verify_all_columns option either for the PUBLIC group or just for the user contained in the SQL Remote connection string. See "verify_all_columns option [SQL Remote]" [*SQL Anywhere Server - Database Administration*].

The Extraction utility (dbxtract)

When the verify_all_columns option is set on the consolidated database before the remote databases are extracted, then the verify_all_columns option on the remote databases is set by Extraction utility (dbxtract) and the **Extract Database Wizard**.

Custom conflict resolution using triggers

Custom conflict resolution can take several forms. For example, in some applications, resolution can:

- Compare the dates of the original transactions. See "Resolving date conflicts" on page 44.
- Perform calculations on the results of two or more updates. See "Resolving inventory conflicts" on page 45.

- Report the conflict into a table.

Custom conflict resolution requires you to write RESOLVE UPDATE triggers.

Using the RESOLVE UPDATE conflict resolution trigger

RESOLVE UPDATE triggers fire before each row is updated. 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
```

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 return a `column not found` error.

Using the CURRENT REMOTE USER special value

The CURRENT REMOTE USER special value is set by the receive phase of SQL Remote when it is applying messages to the database. The CURRENT REMOTE USER special value is most useful in triggers to determine whether the operations being applied are being applied by the receive phase of SQL Remote, and if they are, which remote user generated the operations being applied.

See also

- "CREATE TRIGGER statement" [*SQL Anywhere Server - SQL Reference*]
- "UPDATE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Replicating triggers" on page 35
- "-t option, SQL Remote Message Agent utility (dbremote)" on page 138

Resolving date conflicts

To illustrate how you can resolve date conflicts, suppose a table in a contact management system has a column holding the more 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 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.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

By default, the Monday update would proceed, leaving the column with the incorrect information that the more recent contact occurred on Friday. However, update conflicts on this column should be resolved by inserting the more recent date in the row.

Implementing the solution

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

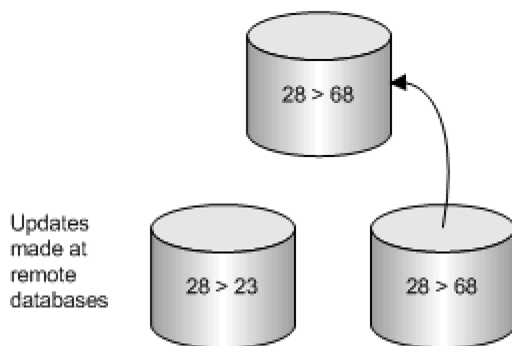
```
CREATE TRIGGER contact_date RESOLVE UPDATE
 ON Contacts
 REFERENCING OLD AS old_name
  NEW AS new_name
 FOR EACH ROW
 BEGIN
   IF new_name.contact_date <
      old_name.contact_date THEN
          SET new_name.contact_date
           = old_name.contact_date
   END IF
 END;
```

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

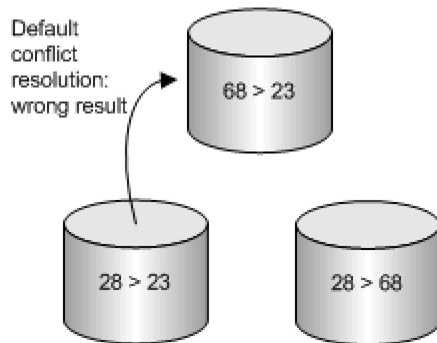
Consider a warehouse system for a manufacturer of sporting goods. There is a table of product information, with a Quantity column holding the number of each product left in stock. An update to this column typically depletes the quantity in stock or, if a new shipment is brought in, adds to it.

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 her database. Meanwhile, before this update is replicated to the consolidated database, another sales representative receives 40 returned tee shirts. This sales representative enters the returns into his remote database and replicates the changes to the consolidated database at the warehouse, adding 40 to the Quantity column to make it 68.

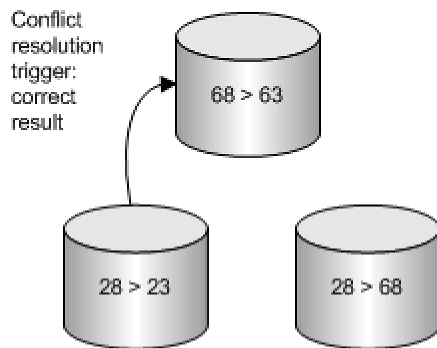


The warehouse entry is added to the database: the Quantity column now shows there are 68 small tank-top tee shirts in stock. When the update from the first sales representative arrives, it causes a conflict—SQL Anywhere detects that the update is from 28 to 23, but that the current value of the column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

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



In this example, the conflict should be resolved by summing the changes to the inventory column to produce the 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 statement was executed (28) to the new value being sent, before the UPDATE statement is implemented. So, new_name.Quantity becomes 63 (= 23 + 68 - 28), and this value is entered into the Quantity column.

Consistency is maintained on the remote database as follows:

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

1. The original remote UPDATE statement 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.

Row not found errors

A user deletes a 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.

To replicate UPDATE and DELETE statements correctly, you must include all of the primary key columns in the article.

When an UPDATE or a DELETE statement 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 unique constraint. A unique index is not enough.

WHERE clause and primary keys

The primary key columns are used in the WHERE clause of replicated UPDATE and DELETE statements. When a table has no primary key, the WHERE clause refers to all columns in the table. See "Replicating INSERT and DELETE statements" on page 33.

See also

- "Reporting and handling replication errors" on page 117

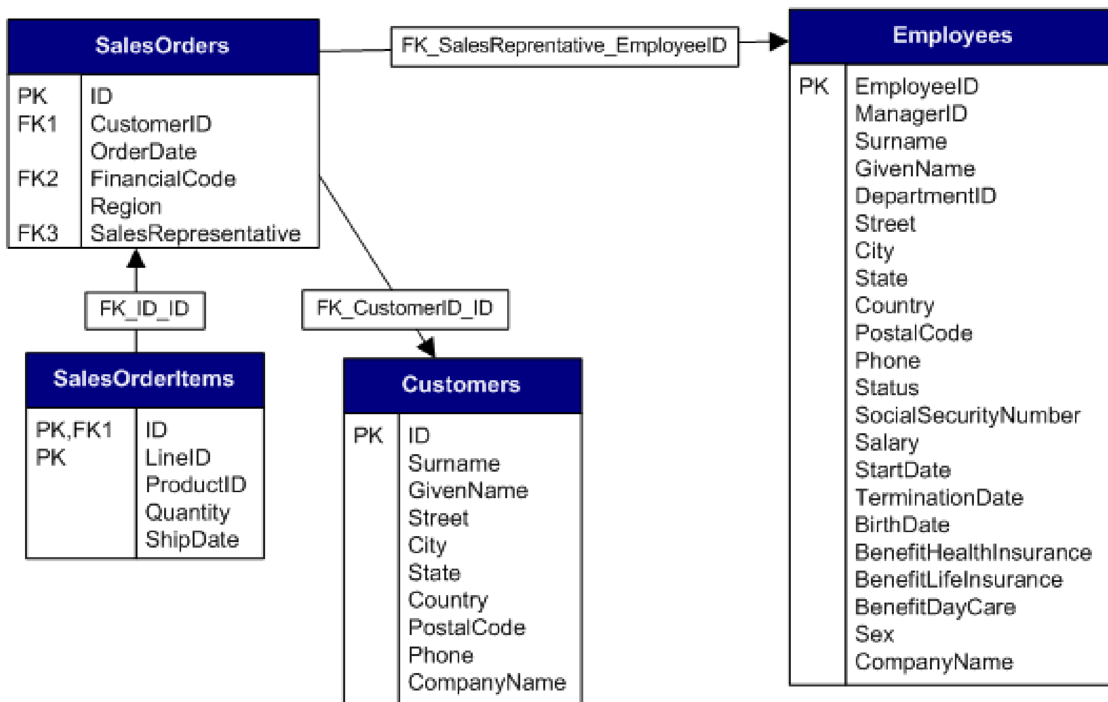
Referential integrity errors

The tables in a relational database are often related through foreign key references. As a result, referential integrity constraints ensure that the database remains consistent. See "Enforcing entity and referential integrity" [*SQL Anywhere Server - SQL Usage*].

When you replicate only a part of a database, you must ensure that the replicated database still has referential integrity.

You want to avoid unreplicated referenced table errors. Your remote databases should not contain foreign keys that point to unreplicated tables.

For example, in a consolidated database the SalesOrders table has a foreign key to the Employees table. SalesOrders.SalesRepresentative is the foreign key that references the primary key, Employees.EmployeeID.



A publication, PubSales, is created that excludes the Employees table, but includes the entire SalesOrder table.

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

A remote user, Rep1, subscribes to the PubSales publication. Then, you extract Rep1 from the consolidated database and try to create a database for Rep1. However, the database creation fails because Rep1 is missing the Employees table. To avoid this problem, you can:

- **Remove the foreign key reference** To exclude foreign key references, specify the -xf option when using the Extraction utility (dbxtract).

  However, if you remove the foreign key reference from the remote database, then there is no constraint in the remote database to prevent an invalid value from being inserted into the SalesRepresentative column of the SalesOrders table.

  If an invalid value is inserted in the SalesRepresentative column at the remote database, the replicated INSERT statement fails on the consolidated database.

- **Include the missing table in the publication** Include the Employees table (or at least its primary key) in the publication. For example:

  ```
  CREATE PUBLICATION PubSales (
     TABLE Customers,
  ```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

```
        TABLE SalesOrders,
        TABLE SalesOrderItems,
        TABLE Products,
        TABLE Employees
    );
```

See also

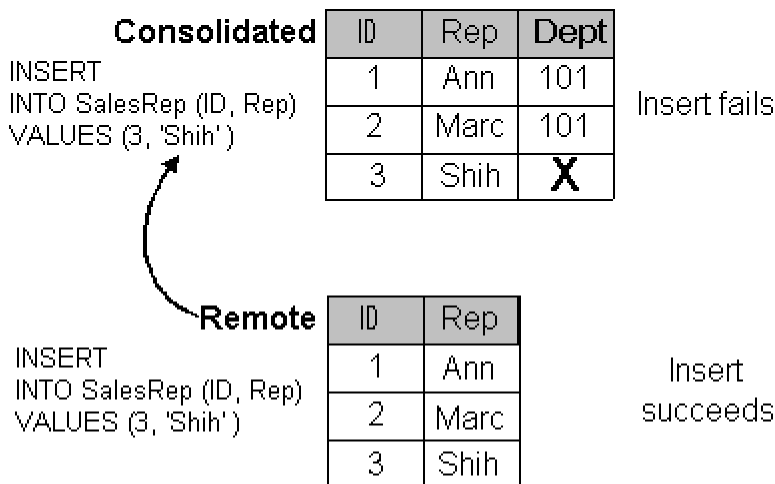
- "Reporting and handling replication errors" on page 117

Insert errors

When replicating INSERT statements from a remote database to a consolidated database, you can only exclude the following columns from the publication:

- 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 fail when replicated on the consolidated database.



> **Using BEFORE triggers as an alternative**
> An exception to this example is when a BEFORE trigger is used to maintain the columns that are not included in the INSERT statement.

See also

- "Replication conflicts and errors" on page 39

Duplicate primary key errors

When all users are connected to the same database, there is no problem ensuring that each INSERT statement uses a unique primary key. If a user tries to re-use a primary key, 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 remote databases, insert a row using the same primary key value. Each of their statements succeeds because the primary key value is unique on each remote database.

However, when these two users replicate their databases with the same consolidated database, a problem arises. The first database to replicate with the consolidated database succeeds. However, the second insert to reach a given database in the replication system fails.

Primary key values must be unique

To avoid primary key errors, you must ensure that when a database inserts a row, its primary key is guaranteed to be unique across all databases in the system. There are several techniques for achieving this goal, including:

1. Using the default global autoincrement feature of SQL Anywhere databases. See “Global autoincrement columns” on page 50.

2. Using a primary key pool to maintain a list of unused, unique primary key values at each site. See “Using primary key pools” on page 51.

These techniques can be used either separately or together to avoid duplicate primary keys.

See also

- “Reporting and handling replication errors” on page 117

Global autoincrement columns

Use the GLOBAL AUTOINCREMENT default to assign each remote database a unique global database identification number.

When you specify the GLOBAL AUTOINCREMENT default for a column, 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.

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

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

See also

- “The GLOBAL AUTOINCREMENT default” [*SQL Anywhere Server - SQL Usage*]

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 clause in a CREATE TABLE or ALTER TABLE statement.

Partition size

Optionally, the partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size may be any non-negative integer, although the partition size is generally chosen so that the supply of numbers within any one partition is rarely 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 your column is not of type INT or BIGINT, it is recommended that you specify the partition size explicitly.

Example

The following statement creates a table with two columns: an integer that holds a customer identification number and a character string that holds the customer's name. The identification number column, ID, uses the GLOBAL AUTOINCREMENT default and has a partition size of 5000.

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

See also

- “CREATE TABLE statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*]

Using primary key pools

A **primary key pool** is a table that contains a set of primary key values for each database in a SQL Remote system. A master primary key pool table is created and stored on the consolidated database. Remote users subscribe to the consolidated database primary key pool table to receive 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 their pool. The pool is maintained by periodically running a procedure on the consolidated database that replenishes the supply.

The primary key pool technique requires the following components:

- **Primary key pool table** On the consolidated database, you need a table to hold valid primary key values for each database in the system. See "Create a primary key pool table" on page 52.

- **Replenishment procedure** On the consolidated database, you need a stored procedure to keep the key pool table filled. See "Fill and replenish the key pool" on page 53.

- **Sharing of key pools** Each remote database in the system must subscribe to its own set of valid values from the consolidated database key pool table. See "Replicate the primary key pool" on page 53.

- **Data entry procedures** On the remote databases, new rows are entered using a stored procedure that picks the next valid primary key value from the pool and then deletes that value from the key pool. See "Use the primary keys from the key pool" on page 54.

Create a primary key pool table

To create a primary key pool table (SQL)

1. On the consolidated database, execute the following statement to create a primary key pool table:

```
CREATE TABLE KeyPool (
   table_name VARCHAR(128) NOT NULL,
   value INTEGER NOT NULL,
   location CHAR(12) NOT NULL,
   PRIMARY KEY (table_name, value),
);
```

Column	Description
table_name	Holds the names of tables for which primary key pools must be maintained. For example, if new sales representatives are added only on the consolidated database, only the Customers table needs a primary key pool and this column is redundant.
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 systems, this can be the same as the rep_key value of the SalesReps table. In other systems, there are users other than sales representatives; in such systems, the two identifiers should be distinct.

2. To increase performance, execute the following statement to create an index on the primary key table:

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

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Replicate the primary key pool

You can either incorporate the primary key pool into an existing publication or share it as a separate publication. Use the following procedure to create a separate publication for the primary key pool and subscribe users to it.

To replicate the primary key pool (SQL)

1. On the consolidated database, 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.

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "CREATE SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Fill and replenish the key pool

Every time a remote user adds a new customer, the remote user's pool of available primary keys is depleted by one. Periodically, you need to replenish the contents of the primary key pool table on the consolidated database and then replicate the new primary keys to the remote databases.

To initially fill the primary key pool (SQL)

1. On the consolidated database, create a procedure to fill the primary key pool.

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

   For example:

```
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;
```

2. Insert an initial primary key value in the primary key pool for each user.

   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();
```

   The ReplenishPool 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.

3. Periodically run ReplenishPool.

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

Use the primary keys from the key pool

When a sales representative adds a new customer to the Customers table, the primary key value to be inserted is obtained using a stored procedure. This example uses a stored procedure to supply the primary key value, and a stored procedure to do the insert.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

To use the primary keys (SQL)

1. Create a procedure to run on the remote databases to obtain a primary key from the primary key pool table.

   For example, 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;
   ```

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

2. Create a procedure that runs on the remote databases to insert a new row in a subscribed table.

   For example, 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 can enhance this procedure by testing the new_cust_key value obtained from NewKey to check that it is not NULL, and prevent the insert if it is NULL.

Partitioning rows among remote databases

Each remote database can contain a different subset of the data stored in the consolidated database. You can create your publications and subscriptions so that data is partitioned among remote databases.

The partitioning can be disjoint, or it can contain overlaps. For example, if each employee has their own set of customers, with no shared customers, the partitioning is **disjoint**. If there are shared customers who appear in more than one remote database, the partitioning contains **overlaps**.

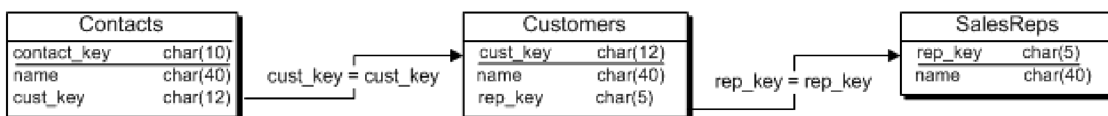
Sometimes, the rows of a table need to be partitioned even when the subscription expression does not exist in the table. See "Using disjoint data partitions" on page 56.

Sometimes, when there is a many-to-many relationship in the database, the tables need to be partitioned. See "Using overlap partitions" on page 61.

Using disjoint data partitions

Data partitioning is disjoint when the remote databases do not share data. For example, each sales representative has their own set of customers and they do not share customers with other sales representatives.

In the following example, three tables store information about the interactions between sales representatives and customers: Customers, Contacts, and SalesReps. Each sales representative sells to several customers. For some customers, there is a single contact, and for other customers there are multiple contacts.



Description of Contacts, Customers, and SalesReps tables

The following table describes the Customers, Contacts, and SalesReps database tables as described in "Using disjoint data partitions" on page 56.

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Table	Description	Table definition
Contacts	All individual contacts that do business with the company. Each contact belongs to a single customer. The Contacts table includes the following columns: ● **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.	CREATE TABLE Contacts (contact_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, cust_key CHAR(12) NOT NULL, FOREIGN KEY REFERENCES Customers, PRIMARY KEY (contact_key));
Customers	All customers that do business with the company. The Customers table includes the following columns: ● **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.	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));
SalesReps	All sales representatives that work for the company. The SalesReps table includes the following columns: ● **rep_key** An identifier for each sales representative. This is the primary key. ● **name** The name of each sales representative.	CREATE TABLE SalesReps (rep_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, PRIMARY KEY (rep_key));

A sales representative must subscribe to a publication that provides the following information:

- **A list of the all the sales representatives working for the company** The following statement creates a publication that publishes the entire SalesRep table:

  ```
  CREATE PUBLICATION SalesRepData (
     Table SalesReps ...)
  );
  ```

- **A list of customers assigned to them** This information is available in the Customers table. The following statement creates a publication that publishes the Customers table, which contains the rows that match the value of the rep_key column in the Customers table:

  ```
  CREATE PUBLICATION SalesRepData (
     TABLE Customers SUBSCRIBE BY rep_key ...
  );
  ```

- **A list of the contact information for their assigned customers** This information is available in the Contacts table. The Contacts table must be partitioned among the sales representatives, but there is no reference to the rep_key value in the SalesRep table. To solve this problem, you can use a subquery in the Contacts article that references the rep_key column of the Customers table.

  The following statement creates a publication that publishes the Contacts table, which contains the rows that reference the rep_key column of the Customers table.

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

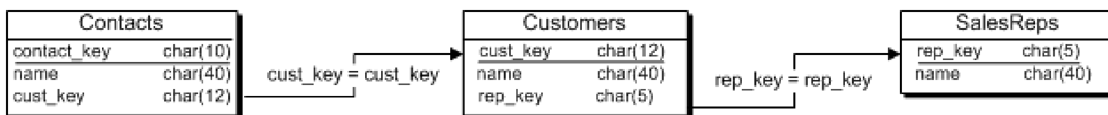
  One row in the Customers table has the cust_key value in the current row of the Contacts table; the WHERE clause in the SUBSCRIBE BY statement ensures that the subquery only returns a single value.

The following statement creates the complete publication:

```
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 )
);
```

Using BEFORE UPDATE triggers

In the following example, three tables store information about the interactions between sales representatives and customers: Customers, Contacts, and SalesReps. Each sales representative sells to several customers. For some customers, there is a single contact, and for other customers there are multiple contacts.



For detailed descriptions of the tables, see "Description of Contacts, Customers, and SalesReps tables" on page 56.

A sales representative subscribes to a publication that provides a copy of the SalesRep table, a copy of the Customers table with the details of the customers assigned to them, and a copy of the Contacts table with the details of the contacts that correspond to their customers. For example, each sales representative subscribes to the following publication:

```
CREATE PUBLICATION SalesRepData (
   TABLE SalesReps
```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

```
    TABLE Customers
       SUBSCRIBE BY rep_key
    TABLE Contacts
       SUBSCRIBE BY (SELECT rep_key
          FROM Customers
          WHERE Contacts.cust_key = Customers.cust_key )
);
```

For a detailed description of this publication, see "Using disjoint data partitions" on page 56.

Maintaining referential integrity

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.

On the consolidated database, when a customer is reassigned to a new sales representative, the rep_key value in the Customers table is updated.

The following statement reassigns a customer, cust1, to another sales representative, rep2.

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

This update is replicated:

- As a DELETE statement to the Customers table on the old sales representative's remote database.
- As an INSERT statement to the Customers table on the new sales representative's remote database.

The Contacts table is **not** changed. There are no entries in the consolidated database transaction log about the Contacts table. As a result, SQL Remote on the remote databases cannot reassign the cust_key rows of the Contacts table. This inability causes the following referential integrity problem: the Contacts table on the remote database of the old sales representative contains a cust_key value for which there is no longer a customer.

A solution is to use a BEFORE UPDATE trigger. A BEFORE UPDATE trigger does not make any change to the database tables, but does create an entry in the consolidated database transaction log.

This BEFORE UPDATE trigger must be fired:

- Before the UPDATE statement is run, so that the BEFORE value of the row is evaluated and added to the transaction log.
- FOR EACH ROW rather than for each statement. The information provided by the trigger must be the new subscription expression.

For example, the following statement creates a BEFORE UPDATE trigger.

```
CREATE TRIGGER "UpdateCustomer" BEFORE UPDATE OF "rep_key"
// only fire the trigger when rep_key is modified, not any other column
ORDER 1 ON "Cons"."Customers"
/* REFERENCING OLD AS old_name NEW AS new_name */
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
;
```

SQL Remote uses the information placed in the transaction log to determine which subscribers receive which rows.

The consolidated database transaction log contains two entries after this statement is executed:

- INSERT and DELETE statements for the Contacts table generated by the BEFORE UPDATE trigger.

```
--BEGIN TRIGGER-1029-0000461705
--BEGIN TRANSACTION-1029-0000461708
BEGIN TRANSACTION
go
--UPDATE PUBLICATION-1029-0000461711 Cons.Contacts
--PUBLICATION-1029-0000461711-0002-NEW_SUBSCRIBE_BY-rep2
--PUBLICATION-1029-0000461711-0002-OLD_SUBSCRIBE_BY-rep1
--NEW-1029-0000461711
--INSERT INTO Cons.Contacts(contact_key,name,cust_key)
--VALUES ('5','Joe','cust1')
go
--OLD-1029-0000461711
--DELETE FROM Cons.Contacts
-- WHERE contact_key='5'
go
--END TRIGGER-1029-0000461743
```

- The original UPDATE statement that was executed, as well as INSERT and DELETE statements for those users that gained or lost the row respectively.

```
--PUBLICATION-1029-0000461746-0002-NEW_SUBSCRIBE_BY-rep2
--PUBLICATION-1029-0000461746-0002-OLD_SUBSCRIBE_BY-rep1
--NEW-1029-0000461746
--INSERT INTO Cons.Customers(cust_key,name,rep_key)
--VALUES ('cust1','company1','rep2')
go
--OLD-1029-0000461746
--DELETE FROM Cons.Customers
-- WHERE cust_key='cust1'
go
--UPDATE-1029-0000461746
UPDATE Cons.Customers
   SET rep_key='rep2'
VERIFY (rep_key)
VALUES ('1')
 WHERE cust_key='cust1'
go
--COMMIT-1029-0000461785
COMMIT WORK
```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

SQL Remote scans the transaction log for the BEFORE and AFTER tags. Based on this information, it can determine which remote users get an INSERT, UPDATE, or DELETE statement.

- When a user is in the BEFORE list and not in the AFTER list, then a DELETE statement is sent on the Contacts table.
- When a user is in the AFTER list and not the BEFORE list, then an INSERT statement is sent on the Contacts table.
- When a user is in both the BEFORE and AFTER lists, nothing is done to the Contacts table but the UPDATE statement on the Customers table is sent.

When the BEFORE and AFTER lists are identical, the remote user already has the row and an UPDATE statement is sent.

Notes on the trigger

In the following example, you must use a BEFORE UPDATE trigger. In other contexts, BEFORE DELETE and BEFORE INSERT are necessary.

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

In this example, you use a BEFORE trigger.

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

The UPDATE statement lists the affected publication and table. The WHERE clause in the statement describes the affected rows. This UPDATE statement does not change the data in the table; it makes entries in the transaction log.

In this example, the subscription expression returns a single value. However, subqueries returning multiple values can also be used. The value of the subscription expression must be the value after the update.

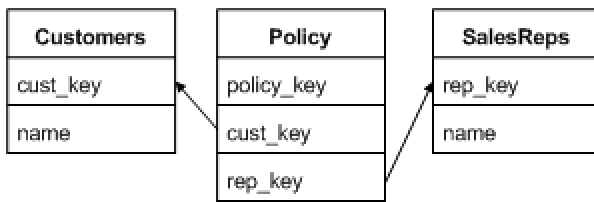
In this example, the only subscriber to the row is the new sales representative. For an example of a row that has existing and new subscribers, see “Using overlap partitions” on page 61.

Using overlap partitions

Data partitioning overlaps when the remote databases share data. For example, sales representatives share customers amongst themselves.

Suppose three tables store information about the interactions between sales representatives and customers: Customers, Policy, and SalesReps. Each sales representative sells to several customers, and some

customers deal with more than one sales representative. The Policy table has foreign keys to both the Customers and SalesReps tables. There is a many-to-many relationship between Customers and SalesReps.



Description of Customers, Policy, and SalesReps tables

The following table describes Customers, Policy, and SalesReps database tables as discussed in "Using overlap partitions" on page 61.

<table>
<tr><th>Table</th><th>Description</th></tr>
<tr><td>Customers</td><td>All customers that do business with the company. The Customers table has the following columns:

● cust_key A primary key column containing an identifier for each customer.

● name A column containing the name of each customer.

The following statements create this table:

<pre>CREATE TABLE Customers (
   cust_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   PRIMARY KEY (cust_key)
);</pre></td></tr>
</table>

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

<table>
<tr><th>Table</th><th>Description</th></tr>
<tr><td>Policy</td><td>
A three-column table that maintains the many-to-many relationship between customers and sales representatives. The Policy table has the following columns:

policy_key A primary key column containing an identifier for the sales relationship.
cust_key A column containing a foreign key for the customer representative in a sales relationship.
rep_key A column containing a foreign key for the sales representative in a sales relationship.

The following statements create this table:
<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>
</td></tr>
<tr><td>SalesReps</td><td>
All sales representatives that work for the company. The SalesReps table has the following columns:

rep_key An identifier for each sales representative. This is the primary key.
name The name of each sales representative.

The following statements create this table:
<pre>
CREATE TABLE SalesReps (
   rep_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   PRIMARY KEY (rep_key)
);
</pre>
</td></tr>
</table>

Partitioning data

The many-to-many relationship between customers and sales representatives introduces new challenges for sharing information properly.

Sales representatives must subscribe to a publication that provides the following information:

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

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

- **Those rows from the Policy table that include sales relationships involving the sales representative subscribed to the data** This article uses a SUBSCRIBE BY subscription expression to specify a column used to partition the data among the sales representatives:

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

  The subscription expression ensures that each sales representative receives only those rows in 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.

- **Those rows from the Customers table listing customers that deal with the sales representative subscribed to the data** The Customers table has no reference to the sales representative value that is used in the subscriptions to partition the data. This problem can be 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. That is, there are overlapping subscriptions.

  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.

The following statement creates the complete publication:

```
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
   )
);
```

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 the sales representatives that deal with the particular customer. When a subscription expression has multiple values, the row is replicated to all subscribers whose subscription

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

matches any of the values. This ability to have multiple-valued subscription expressions allows overlapping partitioning of a table.

Maintaining referential integrity when reassigning rows among subscribers

To cancel a sales relationship between a customer and a sales representative, a row in the Policy table is deleted. In this example, the Policy table change is properly replicated to the old sales representative. However, no change has been made to the Customers table, and so no changes to the Customers table are replicated to the old sales representative.

In the absence of triggers, this can leave a 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 BEFORE triggers that fire when changes are made to the Policy table. These special triggers makes no changes to the database tables, but they do make an entry in the transaction log that SQL Remote uses to maintain data in subscriber databases.

A BEFORE INSERT trigger

For example, the following statements create a BEFORE INSERT 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

The following statements create a BEFORE DELETE 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;
```

The SUBSCRIBE BY clause of the UPDATE PUBLICATION statement contains a subquery, and this subquery can be multiple-valued.

Multiple-valued subqueries

The subquery in the SUBSCRIBE clause of the UPDATE PUBLICATION is a UNION expression, and can be multiple-valued:

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

- The first part of the UNION is the set of existing sales representatives dealing with the customer, taken from the Policy table.

  The result set of the subscription query must be all those sales representatives receiving the row, not just the new sales representatives.

- 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 subquery in the BEFORE DELETE trigger is multiple-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 representatives 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**).

  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 at more than one remote site between replication messages, which can 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.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- Updates on the Policy table have not been described here. Either they should be prevented, or a BEFORE UPDATE trigger must be created that combines features of the BEFORE INSERT and BEFORE DELETE triggers shown in the example.

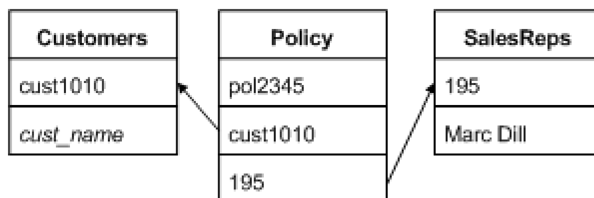
Using the subscribe_by_remote option with many-to-many relationships

When the subscribe_by_remote option is set to On, operations from remote databases on rows with a SUBSCRIBE BY value of NULL or an empty string assume the remote user is subscribed to the row. By default, the subscribe_by_remote option is set to On.

The subscribe_by_remote option solves a problem that otherwise would arise with some publications. The following publication uses a subquery for the Customers table subscription expression because customers can belong to several sales representatives:

```
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
   ),
);
```

For example, Marc Dill is a Sales representative who has just arranged a policy with a new customer. He inserts a new row in the Customers table and inserts a row in the Policy table to assign the new customer to himself.



On the consolidated database, SQL Remote carries out the insert of the Customers row and SQL Anywhere records the subscription value in the transaction log, at the time of the insert.

Later, when the SQL Remote scans the transaction 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 Customer is sent back to Marc Dill as a DELETE statement.

As long as subscribe_by_remote is set to On, the SQL Remote assumes the row belongs to the Sales representative who inserted it, the INSERT statement 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.

See also

- "subscribe_by_remote option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]

Assigning unique identification numbers to each database

You must assign a different identification number to each remote database. You can create and distribute 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.

Using the global_database_id option

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.

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

- When the column contains no values in the current partition, the first default value is $pn + 1$.
- When the column contains values in the current partition, but all are less than $p(n + 1)$, the next default value is one greater than the previous maximum value in this range.
- Default column values are not affected by values in the column outside 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, when 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 example, 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 when 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. See "Declaring DEFAULT GLOBAL AUTOINCREMENT" on page 51.

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. See "Using primary key pools" on page 51.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

You can set an event handler to automatically notify the database administrator (or do some other action) when the partition is nearly exhausted. See "Defining trigger conditions for events" [*SQL Anywhere Server - Database Administration*].

See also

- "global_database_id option" [*SQL Anywhere Server - Database Administration*]

Setting the global_database_id value

To set the global database identification number (SQL)

- Set the value of the global_database_id option. 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.

See also

- "global_database_id option" [*SQL Anywhere Server - Database Administration*]

Setting unique database identification numbers when extracting databases

If you use the Extraction utility (dbxtract) or the **Extract Database Wizard** to create your remote databases, you can write a stored procedure to automate the task of setting unique database identification numbers.

To automate setting unique database identification numbers

1. Create a stored procedure named **sp_hook_dbxtract_begin**.

   For example, to extract a database for remote user user2 with a user_id of 1001, execute the following statements:

   ```
   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';
```

Each extracted or re-extracted database gets a different global_database_id. The first starts at 1001, the next at 2001, and so on.

2. Run the Extraction utility (dbxtract) with the -v option or the **Extract Database Wizard** to extract your remote databases. The Extraction utility does the following tasks:

   a. Creates a temporary table name #hook_dict, with the following contents:

name	value
extracted_db_global_id	user ID being extracted

   When you write a 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 DEFAULT GLOBAL AUTOINCREMENT values.

   - When you do not define an sp_hook_dbxtract_begin procedure, the extracted database has a global_database_id set to 101.
   - When you define a sp_hook_dbxtract_begin procedure that does not modify any rows in the #hook_dict, then the global_database_id is still set to 101.

   b. Calls the **sp_hook_dbxtract_begin**.

   c. Outputs the following information to assist in debugging procedure hooks:

   - 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.

See also

- "The #hook_dict table" on page 152
- "SQL Remote system procedures" on page 151
- "global_database_id option" [*SQL Anywhere Server - Database Administration*]

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Managing SQL Remote systems

You deploy and administer a SQL Remote system from the consolidated database.

To deploy and administer a SQL Remote system

1. Set up the consolidated database.

   See “Creating SQL Remote systems” on page 9.

2. Review and test your SQL Remote system.

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

3. Create remote databases and deploying the design.

   As the DBA of the consolidated database, you deploy SQL Remote by:

   a. Creating a SQL Anywhere database for each remote user, with their own initial copy of the data, and starting their subscriptions. See “Extracting remote databases” on page 72.

   b. Installing on each remote user's computer the SQL Anywhere database server, the remote database, SQL Remote, and the client application. See “Deploying embedded database applications” [*SQL Anywhere Server - Programming*] and “Deploying SQL Remote” [*SQL Anywhere Server - Programming*].

4. Run the SQL Remote Message Agent (dbremote) to exchange messages.

   To exchange messages, you need to:

   a. Decide whether to run the SQL Remote Message Agent (dbremote) in continuous mode or batch mode on the consolidated and remote databases. “Choosing the SQL Remote Message Agent (dbremote) mode” on page 79.

   b. Ensure that the system is properly configured with correct user names, SQL Remote Message Agent (dbremote) connection strings, permissions, and so on. See “Understanding the SQL Remote Message Agent (dbremote)” on page 78.

5. Manage messages.

   Use the Guaranteed Message Delivery System to manage the messages being sent back and forth among many databases. See “Understanding the Guaranteed Message Delivery System” on page 93.

6. Improve performance.

   See “Improving SQL Remote performance” on page 84.

7. Implement a backup and recovery strategy.

You must create and implement a backup and recovery strategy for the consolidated database. See “Backing up SQL Remote systems” on page 109.

8. Handle errors.

   See “Reporting and handling replication errors” on page 117.

9. Upgrade the software and database schemas as required.

   See “Upgrading and resynchronization” on page 122.

Extracting remote databases

To create a database for a remote user, you **extract** the remote database from the consolidated database.

You can use either the **Extract Database Wizard** or the Extraction utility (dbxtract) to extract a remote database from a consolidated database for a specified remote user. Either method allows you to do one or more of the following tasks:

- **Automatically extract and reload the schema and data directly into a new or existing database** This is an appropriate method to use when learning about SQL Remote. If you use this method, no interim copy of the data is created on disk. This method provides greater security for your data. However, it is more time consuming to implement. See “Automatically extract remote databases” on page 72.

- **Extract the schema and data to files, and then load them into a new or existing database** When deploying SQL Remote, this method is preferred. You can edit the schema file to customize the extraction and creation of your remote databases. See “Extracting remote databases to a reload file” on page 73.

  To increase efficiency when creating more than one remote database, see “Creating multiple remote databases” on page 77.

Automatically extract remote databases

For information extracting remote databases to a reload file, see “Extracting remote databases” on page 72.

Use the following procedure to extract a consolidated database and reload the schema and data into a new database. No interim copy of the data is created on disk.

To automatically extract a remote database (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.

2. From the **Tools** menu, choose **SQL Anywhere 12** » **Extract Database**.

3. When prompted, choose **Extract And Reload Into A New Database**.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

When prompted, choose **Extract Structure And Data**.

4. Follow the instructions in the wizard and accept the defaults.

   The new remote database is created with the appropriate schema, remote users, publications, subscriptions, and triggers. By default, the data from the consolidated database is extracted to the remote database and the subscriptions are started. However, the wizard does not start the SQL Remote Message Agent, so no messages are exchanged. See "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

To automatically extract a remote database (SQL)

1. Connect to the consolidated database as a user with DBA authority.

2. Run the Extraction utility (dbxtract) and specify the -ac option to extract to an existing database or the -an option to extract to a new database. See "Extraction utility (dbxtract)" on page 140.

   If you specify the -an option, you must create an empty database before running the Extraction utility (dbxtract). For example, the following command creates an empty database named *mydata.db*:

   ```
   dbinit c:\remote\mydata.db
   ```

   Run the following command to extract a new remote database from a consolidated database located at *c:\consolidateddata.db*. The new database is for the remote user named field_user and the new database is created at *c:\remote\mydata.db*:

   ```
   dbxtract -c "UID=DBA;PWD=sql;DBF=c:\consolidateddata.db"
   -an c:\remote\mydata.db field_user
   ```

   The new remote database, *mydata.db*, is created with the appropriate schema, remote users, publications, subscriptions, and triggers. By default, the data from the consolidated database is extracted into the remote databases and the subscriptions are started. However, the Extraction utility (dbxtract) does not start the SQL Remote Message Agent, so no messages are exchanged. See "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

See also

● "Extracting remote databases to a reload file" on page 73

Extracting remote databases to a reload file

For information about automatically extracting remote databases, see "Extracting remote databases" on page 72.

In most deployment scenarios, you need to customize the extraction and creation of remote databases. You can create a custom extraction by choosing to extract the database into a command file and a series of text files. Then, you can edit these files as required.

When you extract the database into files, you decide whether to create:

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- **A SQL command file named *reload.sql* that contains the statements necessary to build the remote database schema** See -n option "Extraction utility (dbxtract)" on page 140.

  For example, run the following command:

  ```
  dbxtract -c "UID=DBA;PWD=sql;DBF=c:\cons\cons.db" -n "c:\remote
  \reload.sql" UserName
  ```

- **A series of data files, each of which contains the contents of a database table** A series of data files, each of which contains the contents of a database table. A new directory is created, named *extract*, to contain the data files. You can use these files to load data into an existing remote database. See -d option "Extraction utility (dbxtract)" on page 140.

  For example, run the following command:

  ```
  dbxtract -c "UID=DBA;PWD=sql;DBF=c:\cons\cons.db" -d "c:\remote1\" UserName
  ```

- **Both the *reload.sql* file and the data files** A new directory is created, named *extract*, to contain the data files. The *reload.sql* file contains instructions to load the data files. For example, run the following command:

  ```
  dbxtract -c "UID=DBA;PWD=sql;DBF=c:cons\cons.db" "c:\remote1\reload.sql"
  UserName
  ```

The reload.sql file

The *reload.sql* file contains the SQL statements to build the database schema, which includes the commands to create:

- Publishers, remote, and consolidated users
- Publications and subscriptions
- Message types
- Tables
- Views
- Triggers
- Procedures

Editing of reload.sql may be needed
The Extraction utility (dbxtract) 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. See "Editing the reload.sql file" on page 75.

Use the following procedure to create a remote database using a reload.sql file.

To create a remote database from the reload.sql file (command line)

1. Use the Extraction utility (dbxtract) to extract the database schema and data to files. For example, run the following command:

   ```
   dbxtract -c "UID=DBA;PWD=sql;DBF=c:cons\cons.db" "c:\remote\reload.sql"
   UserName
   ```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

By default, subscriptions for the specified remote user are started automatically.

2. Edit the *reload.sql*, if required. See "Editing the reload.sql file" on page 75.

3. Create an empty SQL Anywhere database.

   For example, run the following command:

```
dbinit c:\rem1\rem1.db
```

4. Connect to the database from Interactive SQL, and run the *reload.sql* command file.

   For example, run the following command:

```
READ remote\reload.sql
```

   The new remote database, *rem1.db*, is created with the appropriate schema, remote users, publications, subscriptions, and triggers. However, the Extraction utility (dbxtract) does not start the SQL Remote Message Agent, so no messages are exchanged. See "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

See also

- "Automatically extract remote databases" on page 72

Editing the reload.sql file

You should edit the *reload.sql* command file as needed when creating remote databases. For example, you must edit the *reload.sql* file in the following cases:

Adding unreplicated tables to remote database

Remote databases can have tables that are not present at their consolidated database as long as these tables do not take part in replication. The Extraction utility (dbxtract) and **Extract Database Wizard** cannot extract unreplicated tables from a consolidated database.

After extracting the database, you should edit *reload.sql* to add such tables.

Extracting procedures, triggers, and views

By default, the Extraction utility (dbxtract) and **Extract Database Wizard** extract all stored procedures, triggers, and views from the database. While some of these views and procedures are likely to be required on the remote site, others may not be required. For example, a procedure could refer to parts of the database that are not included in the remote site.

After extracting the database, you should edit *reload.sql* to remove unnecessary procedures, triggers, and views.

Using the Extraction utility (dbxtract) in multi-tiered systems

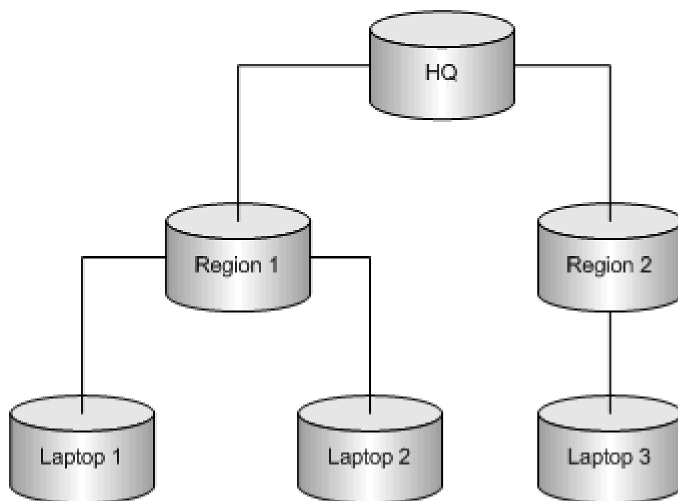
See "Extracting databases for a multi-tier hierarchy system" on page 76.

See also

- "Creating multiple remote databases" on page 77
- "Extracting remote databases to a reload file" on page 73

Extracting databases for a multi-tier hierarchy system

To understand the role of the Extraction utility (dbxtract) and the **Extract Database Wizard** in multi-tiered arrangements, consider a three-tiered SQL Remote system. This system is illustrated in the following diagram.



To create the remote databases for a three-tiered system

1. Use the Extraction utility (dbxtract) on the top-level, consolidated database, HQ, to create the second-level databases Region 1 and Region 2.

2. Use the Extraction utility (dbxtract), on the second-level databases, Region 1 and Region 2, to create the third-level databases for users Laptop 1, Laptop 2, and Laptop 3. The second-level databases are remote databases to the first-level database, HQ, and are consolidated databases to the third-level databases, Laptop 1, Laptop 2, and Laptop 3.

Re-extracting databases in a multi-tier hierarchy system

If you have to re-extract the *schema* for the second-level database from the top-level consolidated database, the Extraction utility (dbxtract) deletes the remote users (Laptop 1, Laptop 2, and Laptop 3) along with their subscriptions and permissions. As a result, you must recreate those third-level users and their subscriptions manually.

If you only have to re-extract the *data* from the second-level databases from the top-level consolidated database, the Extraction utility (dbxtract) does not affect the remote users. See the -d option "Extraction utility (dbxtract)" on page 140.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Fully qualified publication definitions

Fully qualified publication definitions contain WHERE and SUBSCRIBE BY clauses. Usually you do not need to extract fully qualified publication definitions for a remote database, since the remote database typically replicates all rows back to the consolidated database.

See also

- "Creating multiple remote databases" on page 77
- "Extracting remote databases to a reload file" on page 73

Creating multiple remote databases

Use the following steps to increase efficiency when creating more than one remote database.

To create multiple remote databases

1. Make a copy of the consolidated database and start the subscriptions for the remote users from the consolidated database. For example:

   a. Start the subscriptions and then immediately shut down the consolidated database and the SQL Remote Message Agent (if it is running).

      The subscriptions must be started at the same time that the consolidated database copy is made. Any operations that take place between copying the database and starting the subscriptions can be lost, and can lead to errors at remote databases. Starting subscriptions on the consolidated database allows messages to be packaged and sent to subscribers, even if the subscriber databases do not exist yet. See "START SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*] and "Start subscriptions" on page 129.

      To start several subscriptions within a single transaction, use the REMOTE RESET statement. See "REMOTE RESET statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   b. Copy the consolidated database.

      By default, both the Extraction utility (dbxtract) and the **Extract Database Wizard** run at isolation level 3. This isolation level ensures that data in the extracted database is consistent with data on the database server; however, it can prevent other users from using the database. It is recommended that you extract your remote database against a copy of the consolidated database.

   c. Re-start the consolidated database, and if it was running, re-start the SQL Remote Message Agent on the consolidated database.

2. Extract the remote database schema from the copy of the consolidated database. As the database is a copy, there are no locking and concurrency problems; nevertheless, for a large number of remote databases, this process can take a while.

   When extracting the remote database schema, choose the following options:

   a. Extract only the schema for the remote database.

      By default, both the Extraction utility (dbxtract) and the **Extract Database Wizard** extract one database at a time, including the schema and data for each user. However, in most deployment scenarios the remote databases use the same schema but different data. Using the Extraction utility

(dbxtract) or the **Extract Database Wizard** to extract both schema and data for each user results in repeatedly extracting the same schema. See the -n option "Extraction utility (dbxtract)" on page 140.

b. Order the data by primary key.

By default, the data in each table is ordered by primary key. Loading data into the remote database is faster when the data is ordered by primary key. See the -u option "Extraction utility (dbxtract)" on page 140.

3. Create an empty remote database using the *reload.sql* file. Copy this database file to create the required number of remote databases. See "Extracting remote databases to a reload file" on page 73.

4. For each remote database, define the SQL Remote definitions specific to each remote user. See "User permissions" on page 18.

5. For each remote user, extract only their corresponding data from the consolidated database. See the -d option "Extraction utility (dbxtract)" on page 140.

6. Load the data for each remote user into the corresponding remote database.

   As each remote database is created, its data is out of date with the live consolidated database.

   However, when you run the SQL Remote Message Agent (dbremote), each user can receive and apply messages that have been sent from the live consolidated database to bring themselves up to date. See "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

See also

- "Extracting databases for a multi-tier hierarchy system" on page 76
- "Editing the reload.sql file" on page 75

Understanding the SQL Remote Message Agent (dbremote)

The SQL Remote Message Agent (dbremote) is a key component in SQL Remote replication. It must be installed and run on the every database in the system. The SQL Remote Message Agent (dbremote) handles both sending and receiving messages.

It carries out the following functions:

- **SQL Remote Message Agent (dbremote) tasks when sending messages**

  - It scans the transaction log at each publisher database and translates the transaction log entries into messages for subscribers.

  - It sends the messages to subscribers.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

  - When it receives a request to resend messages, the SQL Remote Message Agent (dbremote) resends the messages to the database that made the request.
  - It maintains message information in the system tables, and manages the Guaranteed Message Delivery System.

  See "Sending message tasks" on page 90.

- **SQL Remote Message Agent (dbremote) tasks when receiving messages**
  - It processes incoming messages, and applies them in the proper order to the database.
  - It requests that missing messages be resent.
  - It maintains the message information in the system tables, and manages the Guaranteed Message Delivery System.

  See "Receiving message tasks" on page 84.

Connections

The SQL Remote Message Agent (dbremote) uses several connections to the database server. These are:

- **One global connection** This connection is active all the time the SQL Remote Message Agent (dbremote) is running.
- **One connection for scanning the transaction log** This connection is active during the scan phase only.
- **One connection for executing commands from the transaction log-scanning thread** This connection is active during the scan phase only.
- **One connection for processing synchronize subscription requests** This connection is active during the send phase only.
- **One connection for each worker thread** These connections are alive during the receive phase only.

Choosing the SQL Remote Message Agent (dbremote) mode

The SQL Remote Message Agent (dbremote) can be run in one of two modes:

- **Continuous mode** In continuous mode, the SQL Remote Message Agent (dbremote) periodically sends messages, at times specified by the send frequency properties of each remote user. When it is not sending messages, it receives messages as they arrive.

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

  See "Run the SQL Remote Message Agent (dbremote) in continuous mode" on page 80.

- **Batch mode** In batch mode, the SQL Remote Message Agent (dbremote) receives and processes incoming messages, scans the transaction log once, creates and sends the outgoing messages, and then stops.

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

  See "Run the SQL Remote Message Agent (dbremote) in batch mode" on page 82.

SQL Remote Message Agent (dbremote) requirements

SQL Remote is very flexible. Within a system, you can run the SQL Remote Message Agent (dbremote) in both modes, on multiple devices, and on multiple operating systems. However, SQL Remote has the following requirements:

- **REMOTE DBA authority or DBA authority required** The SQL Remote Message Agent (dbremote) must be run by a user with REMOTE DBA authority or DBA authority. See "Granting REMOTE DBA authority" on page 28.

- **The maximum message length must be the same for each SQL Remote Message Agent (dbremote) in the system** This length can be restricted by operating system memory allocation limits. Received messages that are longer than the limit are deleted as corrupt messages. The default value is 50000 bytes. This length is configurable, using the SQL Remote Message Agent (dbremote) -l option. See "SQL Remote Message Agent utility (dbremote)" on page 131.

Run the SQL Remote Message Agent (dbremote) in continuous mode

Typically, the consolidated database is run in continuous mode. For information about continuous mode and its alternative, batch mode, see "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

To run the SQL Remote Message Agent (dbremote) in continuous mode

1. Ensure that every user with REMOTE authority either has a SEND AT or SEND EVERY frequency specified.

   In continuous mode, the SQL Remote Message Agent (dbremote) sends messages at times specified by the SEND AT or SEND EVERY frequency in the properties of each remote user. See "Setting the send frequency" on page 81.

2. Start the SQL Remote Message Agent (dbremote) *without* the -b option.

   On Windows, the SQL Remote Message Agent (dbremote) is named *dbremote.exe*. On Unix, the name is dbremote. On Mac OS X, you can also use SyncConsole to start the SQL Remote Message Agent (dbremote). See "Run the SQL Remote Message Agent (dbremote) on Mac OS X" on page 83 and "Run the SQL Remote Message Agent (dbremote) on Unix" on page 84.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

For example, the following statement runs dbremote in continuous mode on a database file named *c:\mydata.db*, connecting with user name ManagerSteve and password sql:

```
dbremote -c "UID=ManagerSteve;PWD=sql;DBF=c:\mydata.db" -l 40000
```

The user name, ManagerSteve, must have either REMOTE DBA authority or DBA authority. The maximum message length, as defined by the -l option, must be the same for all databases in the system. See "SQL Remote Message Agent (dbremote) requirements" on page 80.

For a complete list of dbremote options you can specify, see "SQL Remote Message Agent utility (dbremote)" on page 131.

Running the SQL Remote Message Agent (dbremote) as a service in continuous mode

When you run the SQL Remote Message Agent (dbremote) in continuous mode, you can choose to keep the SQL Remote Message Agent (dbremote) running whenever the database server is running. You can do this by running the SQL Remote Message Agent (dbremote) as a Windows service. A service can be configured to keep running even when the current user logs out and to start when the operating system is started.

For a full description of running programs as services, see "Service utility (dbsvc) for Windows" [*SQL Anywhere Server - Database Administration*].

Setting the send frequency

To run the SQL Remote Message Agent (dbremote) in continuous mode, for example on the consolidated database, you must ensure that every REMOTE user either has a send frequency specified. In continuous mode, the SQL Remote Message Agent (dbremote) sends messages at the times specified with the SEND AT or SEND EVERY property.

The SQL Remote Message Agent (dbremote) supports the following send frequency values:

- **SEND EVERY** Specifies a length of time to wait between sending messages.

  When any user with SEND EVERY set is sent messages, all users with the same frequency are sent messages at the same time. For example, all remote users who receive updates every twelve hours are sent updates at the same time, 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.

  A send frequency can be specified in hours, minutes, and seconds in the format ***HH*:*MM*:*SS***.

- **SEND AT** Specifies a time of day at which messages are sent.

  Updates are sent daily at the specified time. You should use as few distinct times as possible rather than staggering the send times. You should choose times when the database is not busy.

- **Default setting (no SEND clause)** If any user has no SEND AT or SEND EVERY clause specified, the SQL Remote Message Agent (dbremote) runs in batch mode, sending messages every

time it is run, and then stopping. See "Run the SQL Remote Message Agent (dbremote) in batch mode" on page 82.

Sending messages too frequently

If you send messages frequently, there is a greater 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 should avoid using very small send frequency periods.

To set the send frequency (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, select the **SQL Remote Users** directory.
3. Right-click a user and choose **Properties**.
4. Click the **SQL Remote** tab.
5. Select either **Send Every** or **Send Daily At** and specify a time.

See also

- "GRANT REMOTE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

Run the SQL Remote Message Agent (dbremote) in batch mode

Use the following procedure to run SQL Remote in batch mode. For information about batch mode and its alternative, continuous mode, see "Understanding the SQL Remote Message Agent (dbremote)" on page 78.

To run the SQL Remote Message Agent (dbremote) in batch mode

1. Ensure that at least one remote user has neither a SEND AT nor a SEND EVERY option in their remote properties.

   If **all** of your remote users have a SEND AT or a SEND EVERY clause defined, and you want to send and receive messages and then shut down, you must start the SQL Remote Message Agent (dbremote) using the -b option.

2. Start the SQL Remote Message Agent (dbremote).

   On Windows, the SQL Remote Message Agent (dbremote) is named *dbremote.exe*. On Unix, the name is dbremote. On Mac OS X, you can also use **SyncConsole** to start the SQL Remote Message Agent (dbremote). See "Run the SQL Remote Message Agent (dbremote) on Mac OS X" on page 83 and "Run the SQL Remote Message Agent (dbremote) on Unix" on page 84.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

For example, the following statement runs dbremote in batch mode on a database file named *c:\mydata.db*, connecting with user name ManagerSteve and password sql:

```
dbremote -c "UID=ManagerSteve;PWD=sql;DBF=c:\mydata.db"
```

The SQL Remote Message Agent (dbremote) receives and processes incoming messages, scans the transaction log once, creates and sends the outgoing messages, and then stops.

The username, ManagerSteve, must have either REMOTE DBA authority or DBA authority. The maximum message length, as defined by the -l option, must be the same for all databases in the system. See "SQL Remote Message Agent (dbremote) requirements" on page 80.

See also

- "SQL Remote Message Agent utility (dbremote)" on page 131

Run the SQL Remote Message Agent (dbremote) on Mac OS X

SQL Anywhere includes an application called SyncConsole that can be used to start the SQL Remote Message Agent (dbremote) on Mac OS X. You can also start the SQL Remote Message Agent on Mac OS X using the dbremote utility.

To start SyncConsole

1. In the Finder, go to */Applications/SQLAnywhere12*.
2. Double-click **SyncConsole**.
3. Choose **File** » **New** » **SQL Remote**.

   A client options window appears.
4. Specify the connection information for the SQL Remote Message Agent (dbremote).

   For example, the following connection parameter uses an ODBC data source for the SQL Anywhere sample database:

   ```
   DSN="SQL Anywhere 12 Demo"
   ```

   The user name must have either REMOTE DBA authority or DBA authority. The message length, as defined by the -l option, must be the same on all databases in the system. See "SQL Remote Message Agent (dbremote) requirements" on page 80.

   For a complete list of dbremote options you can specify in the **Options** field, see "SQL Remote Message Agent utility (dbremote)" on page 131.

Run the SQL Remote Message Agent (dbremote) on Unix

On Unix platforms, you run the SQL Remote Message Agent (dbremote) as a daemon by supplying the -ud option. See -ud option "SQL Remote Message Agent utility (dbremote)" on page 131.

The user name must have either REMOTE DBA authority or DBA authority. The maximum message length, as defined by the -l option, must be the same on all databases in the system. See "SQL Remote Message Agent (dbremote) requirements" on page 80.

For a complete list of dbremote options you can specify, see "SQL Remote Message Agent utility (dbremote)" on page 131.

Improving SQL Remote performance

Each time a row in a table is inserted, deleted, or updated, a message is created for those users 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.

The task of determining who gets what is divided between the database server and the SQL Remote Message Agent (dbremote).

The database server

The database server handles publications. See "The database server handles publications" on page 32.

The SQL Remote Message Agent (dbremote)

The **SQL Remote Message Agent (dbremote)** handles subscriptions.

The SQL Remote Message Agent (dbremote) 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 SQL Remote Message Agent (dbremote) sends the correct operations to each subscriber.

While a large number of subscribers do not have any effect on database server performance, they can affect SQL Remote Message Agent (dbremote) performance. The work in matching subscription values against large numbers of subscription values, and the work in sending the messages, can be demanding.

Receiving message tasks

The SQL Remote Message Agent (dbremote) performs the following tasks when it receives messages:

- **Polling for incoming messages** To check for new messages that have arrived at a database, the SQL Remote Message Agent (dbremote) polls for new messages. See "Adjusting the polling interval to check for new messages" on page 86.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- **Reading the messages** When messages arrive, they are read and stored in cache memory by the SQL Remote Message Agent (dbremote) until they can be applied. See "Adjusting throughput by caching received messages" on page 86.

  If a message is missing and the SQL Remote Message Agent (dbremote) is running in continuous mode, then the SQL Remote Message Agent (dbremote) waits for the message to arrive in a subsequent poll. The number of polls that SQL Remote Message Agent (dbremote) waits is referred to as its **patience**, and is specified by the -rp option.

  - If the missing message arrives before the SQL Remote Message Agent (dbremote) patience expires, then the missing message is added, in the correct order, to the cache.

  - If the missing message does not arrive and the SQL Remote Message Agent (dbremote) patience expires, then the SQL Remote Message Agent (dbremote) sends a request to re-send the message from the publisher database.

  Messages continue to be read and added to the cache until the cache memory usage is exceeded. When the cache memory usage specified using the -m option is exceeded, messages are deleted.

  See "Adjusting the requests to resend messages" on page 87.

- **Applying the messages** The SQL Remote Message Agent (dbremote) applies the messages, in the correct order, to the subscriber database. See "Adjusting the number of worker threads" on page 89.

- **Waiting for confirmation that the messages are applied on the subscriber databases** Once the message has been received and applied at the subscribed database, confirmation is sent back to the publisher. When the publisher SQL Remote Message Agent (dbremote) receives the confirmation, it keeps track of the confirmation in a system table. See "Understanding the Guaranteed Message Delivery System" on page 93.

Improving performance when receiving messages

The major bottleneck for total throughput in a SQL Remote system is generally receiving messages from many remote databases and applying them to the database. To reduce this lag time, when running a SQL Remote Message Agent (dbremote) in continuous mode, you can adjust the following variables:

- How often the SQL Remote Message Agent (dbremote) checks for incoming messages. See "Adjusting the polling interval to check for new messages" on page 86.

- How much memory is used by the SQL Remote Message Agent (dbremote) for holding messages to be sent. See "Adjusting throughput by caching received messages" on page 86.

- How long the SQL Remote Message Agent (dbremote) waits for an out-of-order message to arrive before requesting that the message be resent. See "Adjusting the requests to resend messages" on page 87.

- How many worker threads are used to process the received messages. See "Adjusting the number of worker threads" on page 89.

Adjusting the polling interval to check for new messages

To check for new messages that have arrived at a database, the SQL Remote Message Agent (dbremote) polls for new messages. The default polling interval from the end of one poll to the start of another is 1 minute. You can configure the polling interval using the -rd option, but the default is generally sufficient.

Decreasing the polling interval

You can poll more frequently by using a value in seconds. For example, the following command polls every thirty seconds:

```
dbremote -c "DSN=SQL Anywhere 12 Demo" -rd 30s
```

In general, do not use a small polling interval unless you have a specific reason for requiring a very quick response time for messages. Setting a very small interval can have a detrimental effect on overall system throughput because:

- You can waste resources polling when no messages are in the queue. For example, if you are using email, each poll of the mail server places a load on your message system. Too frequent polling may affect your message system and produce no benefits.

- You can overload your system with resend requests. When adjusting the polling interval, you should also adjust the SQL Remote Message Agent (dbremote) patience. The patience is the number of polls the SQL Remote Message Agent (dbremote) waits for an out-of-sequence message to arrive before requesting that it be sent again. See "Adjusting the requests to resend messages" on page 87.

Increasing the polling interval

You can poll less frequently, as in the following command, which polls every five minutes:

```
dbremote -c "DSN=SQL Anywhere 12 Demo" -rd 5
```

Setting larger polling intervals can provide a better overall throughput of messages in your system, it can increase the time it takes to apply an individual messages. For example, if your polling period for incoming messages is too long, compared to the frequency with which messages are arriving, you can end up with messages sitting in the queue, waiting to be processed.

See also

- "Improving performance when receiving messages" on page 85
- "Adjusting throughput by caching received messages" on page 86
- "Adjusting the requests to resend messages" on page 87
- "Adjusting the number of worker threads" on page 89
- "Improving performance when sending messages" on page 91

Adjusting throughput by caching received messages

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

When messages arrive, the SQL Remote Message Agent (dbremote) reads the messages and then stores them in cache memory until they are applied. This caching of messages prevents:

- Rereading of out-of-order messages from the message system, which may lower performance on large systems. Caching of messages is useful when messages are being read over a WAN (such as Remote Access Services or POP3 through a modem).
- Contention between database worker threads reading messages (a single threaded task) because the message contents are cached.

How messages are cached

Messages are stored in memory until they are applied by the SQL Remote Message Agent (dbremote) when one of the following conditions occurs:

- The transactions are so large that they require multi-part messages.
- The messages arrive out of order.

Specifying the message cache size

Use the SQL Remote Message Agent (dbremote) -m option to specify the size of the message cache. The -m option specifies the maximum amount of memory to be used by the SQL Remote Message Agent (dbremote) for holding messages. The allowed size can be specified as *n* (in bytes), *n*K, or *n*M. The default is 2048K (2M). When the specified cache memory usage is exceeded, messages are deleted.

The -m option is useful when you have a single consolidated database and a large number of remote databases. See the -m option in "SQL Remote Message Agent utility (dbremote)" on page 131.

Example

The following command starts a SQL Remote Message Agent (dbremote) using 12 MB of memory as a message cache:

```
dbremote -c "DSN=SQL Anywhere 12 Demo" -m 12M
```

See also

- "Improving performance when receiving messages" on page 85
- "Adjusting the polling interval to check for new messages" on page 86
- "Adjusting the requests to resend messages" on page 87
- "Adjusting the number of worker threads" on page 89
- "Improving performance when sending messages" on page 91

Adjusting the requests to resend messages

When a message is missing from a sequence, the SQL Remote Message Agent (dbremote) waits a specified number of polls before requesting that the missing message be resent. The number of polls that the SQL Remote Message Agent (dbremote) waits is referred to as its patience. By default, the SQL Remote Message Agent (dbremote) has a patience of 1.

If the SQL Remote Message Agent (dbremote) has a patience of 1 and it expects to receive message 6 but it receives message 7, the SQL Remote Message Agent (dbremote) takes no action. Instead, the SQL Remote Message Agent (dbremote) waits for the results of the next poll. If after the next poll, Message 6 is still missing, then the SQL Remote Message Agent (dbremote) issues a resend request for Message 6.

Increasing the resend patience

Suppose you have a very small polling interval, and a message system that does not preserve the order in which messages arrive. It may be common for out-of-sequence messages to arrive after two or three polls have been completed. In this example, it is recommended that you use the -rp option to increase the SQL Remote Message Agent (dbremote) patience so that a large number of unnecessary resend requests are not sent. The -rp option is often used in conjunction with the -rd option that sets the polling interval. See "Adjusting the polling interval to check for new messages" on page 86.

Example

There are two remote users, named user1 and user2, both of which run the SQL Remote Message Agent (dbremote) with a polling interval of 30 seconds and a patience of 3 polls. For example, they use the following command to run their SQL Remote Message Agents (dbremote):

```
dbremote -c "DSN=SQL Anywhere 12 Demo"  -rd 30s -rp 3
```

In the following sequence of operations, messages are marked as *userX.n* where X is the user name and *n* is the message number. For example, user1.5 is the fifth message from user1. The SQL Remote Message Agent (dbremote) expects messages to start at number 1 for both users.

At time 0 seconds:

1. The SQL Remote Message Agent (dbremote) reads user1.1, user2.4
2. The SQL Remote Message Agent (dbremote) applies user1.1
3. The SQL Remote Message Agent (dbremote) 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 SQL Remote Message Agent (dbremote) reads: no new messages
2. The SQL Remote Message Agent (dbremote) applies: nothing
3. The SQL Remote Message Agent (dbremote) patience is now user1: N/A, user2: 2

At time 60 seconds:

1. The SQL Remote Message Agent (dbremote) reads: user1.3
2. The SQL Remote Message Agent (dbremote) applies: no new messages
3. The SQL Remote Message Agent (dbremote) patience: user1: 3, user2: 1

At time 90 seconds:

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

1. The SQL Remote Message Agent (dbremote) reads: user1.4

2. The SQL Remote Message Agent (dbremote) applies: none

3. The SQL Remote Message Agent (dbremote) patience user1: 3, user2: 0

4. The SQL Remote Message Agent (dbremote) issues resend to user2

When a user receives a new message, it resets the SQL Remote Message Agent (dbremote) patience even if that message is not the one expected.

At time 120 seconds:

1. The SQL Remote Message Agent (dbremote) reads: user1.2 and user2.2

2. The SQL Remote Message Agent (dbremote) applies user1.2, user1.3, user1.4, and user2.2

3. The SQL Remote Message Agent (dbremote) patience user1: N/A, user2: N/A

See also

- “Improving performance when receiving messages” on page 85
- “Adjusting the polling interval to check for new messages” on page 86
- “Adjusting throughput by caching received messages” on page 86
- “Adjusting the number of worker threads” on page 89
- “Improving performance when sending messages” on page 91

Adjusting the number of worker threads

The following steps describe how the SQL Remote Message Agent (dbremote) applies incoming messages:

1. It reads the messages. Messages are read and the header information is examined (to determine the correct order of application). Reading messages from the message system is single-threaded.

2. It applies the messages. Read messages are passed off to database worker threads to be applied.

   On remote databases, the messages are usually applied serially. In a multi-tier system, a remote database can also be a consolidated database for other remotes. On this type of a remote database, the messages are applied as on a consolidated database.

   On the consolidated database, the default is to apply the messages serially. You can use additional database worker threads to apply incoming messages from remote users in parallel. See the -w option in “SQL Remote Message Agent utility (dbremote)” on page 131.

   When database worker threads are used on a consolidated database:

   - 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 are applied by a single worker thread in the correct order.

Advantages of using database worker threads

Using database worker threads on the consolidated database can improve throughput by allowing messages to be applied in parallel rather than serially. The performance advantage is significant when the database server is on a system with a striped drive array.

Disadvantages of using database worker threads

Using database worker threads on the consolidated database, can decrease throughput if they cause a lot of locking between users.

A deadlock is handled by re-applying the rolled-back transaction later.

To set the number of database worker threads

- On the consolidated database, use the -w option to set the number of database worker threads.

  For example, the following command sets the number of worker threads to 5:

  ```
  dbremote -c "DSN=SQL Anywhere 12 Demo"  -w 5
  ```

See also

- “Run the SQL Remote Message Agent (dbremote) in continuous mode” on page 80
- “Improving performance when receiving messages” on page 85
- “Adjusting the polling interval to check for new messages” on page 86
- “Adjusting throughput by caching received messages” on page 86
- “Adjusting the requests to resend messages” on page 87
- “Improving performance when sending messages” on page 91

Sending message tasks

The SQL Remote Message Agent (dbremote) performs the following tasks to send messages:

- **Scanning the publisher transaction log** The SQL Remote Message Agent (dbremote) scans the transaction log of the publisher database and translates the transaction log entries into messages for subscribers. The maximum message length, as defined by the -l option, must be the same on all databases in the system.

  For large transactions, the SQL Remote Message Agent (dbremote) creates multi-part messages. These messages each contain a sequence number that keeps track of their place in the transaction. The SQL Remote Message Agent (dbremote) on the subscriber database uses the sequence number to ensure that the messages are applied in the correct order and that no message is lost.

- **Sending messages to the remote databases** The SQL Remote Message Agent (dbremote) sends messages at times specified by the send frequency properties of each remote user. See “Adjusting the send delay” on page 91.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

The SQL Remote Message Agent (dbremote) sends messages earlier if its cache memory exceeds the set value. The SQL Remote Message Agent (dbremote) stores its messages in cache memory. When the cache memory being used exceeds the specified value, messages are sent. See "Adjusting throughput by caching sent messages" on page 92.

- **Processing resend requests from remote databases** When a user requests that a message be resent, the SQL Remote Message Agent (dbremote) on the publisher database interrupts the regular message sending process to process the resend request.

  You control the urgency with which these resend requests are processed with the -ru option. See "Adjusting how quickly resent requests are processed" on page 93.

- **Sending confirmations to the publisher database** Once a message has been received and applied at the subscribed database, confirmation is sent back to the publisher. See "Understanding the Guaranteed Message Delivery System" on page 93.

Improving performance when sending messages

The major performance issue for sending messages is the turnaround time between when the data is entered at one site to when it appears at other sites. To reduce this lag time, when sending messages with the SQL Remote Message Agent (dbremote), you can adjust the following variables:

- How often messages are sent to remote databases. See "Adjusting the send delay" on page 91.
- The size of the messages. See "Improving performance when receiving messages" on page 85.
- How quickly resend requests are processed. See "Adjusting how quickly resent requests are processed" on page 93.

Adjusting the send delay

To create messages to send, the SQL Remote Message Agent (dbremote) polls for new data from the transaction log. The send delay is the time to wait between polls for more transaction log data to send. The default polling interval from the end of one poll to the start of another is 1 minute. You can configure the send delay using the -sd option, but the default is generally sufficient. The send delay should be less than or equal to the remote users' send frequency.

Decreasing the send delay

You can poll more frequently by using a value in seconds. For example, the following command polls every thirty seconds:

```
dbremote -c "DSN=SQL Anywhere 12 Demo"  -sd 30s ...
```

Increasing the send delay

You can poll less frequently, as in the following command, which polls every 60 minutes:

```
dbremote -c "DSN=SQL Anywhere 12 Demo"  -sd 60
```

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Typically, larger send intervals mean that the SQL Remote Message Agent (dbremote) does most of the message creation work before sending the messages. Smaller intervals are generally preferred as they spread out the message creation work.

See "SQL Remote Message Agent utility (dbremote)" on page 131.

See also

- "Adjusting throughput by caching sent messages" on page 92
- "Adjusting how quickly resent requests are processed" on page 93

Adjusting throughput by caching sent messages

The SQL Remote Message Agent (dbremote) caches messages to be sent in a configurable area of memory.

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 is receiving the same operations. Messages are sent when:

- The send frequency is reached.
- When the cache memory being used exceeds the -m value.
- When the size of the message reaches its maximum size (as specified by the -l option).

Specifying the message cache size

The size of the message cache is specified on the SQL Remote Message Agent (dbremote) command, using the -m option.

The -m option specifies the maximum amount of memory to be used by the SQL Remote Message Agent (dbremote) for building messages. The allowed size can be specified as *n* (in bytes), *n*K, or *n*M. The default is 2048K (2M).

The -m option is useful when you have a single consolidated database and a large number of remote databases. See the -m option "SQL Remote Message Agent utility (dbremote)" on page 131.

Example

The following command starts a SQL Remote Message Agent (dbremote) using 12 MB of memory as a message cache:

```
dbremote -c "DSN=SQL Anywhere 12 Demo" -m 12M
```

See also

- "Adjusting the send delay" on page 91
- "Adjusting how quickly resent requests are processed" on page 93

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Adjusting how quickly resent requests are processed

Because resending a message interrupts the regular message sending process, the SQL Remote Message Agent (dbremote) delays processing resend requests. By default, the SQL Remote Message Agent (dbremote) waits for a time that is half of the send frequency of the remote user who requested the resend.

To resend a message, the SQL Remote Message Agent (dbremote) does the following tasks:

- It stops scanning the transaction log and stops building new messages.
- It deletes the current messages that are stored in its cache waiting to be sent. All of the work that the SQL Remote Message Agent (dbremote) did in reading the transaction log and building those messages is lost.
- It re-reads the transaction log from the offset requested in the resend request. The SQL Remote Message Agent (dbremote) builds the messages and stores them in its cache.
- It waits until the next send frequency time occurs and then sends the messages.

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. To delay processing resend requests until more have arrived, set this option to a longer time. For example, the following command waits one hour before processing a resend request:

```
dbremote -c "DSN=SQL Anywhere 12 Demo" -ru 1h
```

See “SQL Remote Message Agent utility (dbremote)” on page 131.

See also

- “Adjusting the send delay” on page 91
- “Adjusting throughput by caching sent messages” on page 92
- “Adjusting the requests to resend messages” on page 87

Understanding the Guaranteed Message Delivery System

The Guaranteed Message Delivery System ensures that:

- All replicated operations are applied in the correct order. See “Ensuring that operations are applied in the correct order” on page 95.
- No replicated operations are missed. See “Resending lost or corrupt messages” on page 97.
- No replicated operation is applied twice. See “Ensuring that messages are applied only once” on page 97.

The Guaranteed Message Delivery System uses the following information:

- **The status information maintained in the SYSREMOTEUSER system table** This table contains a row for each subscriber, with status information for messages sent to and received by that subscriber. For example:

  - On the consolidated database, the SYSREMOTEUSER system table contains a row for each remote user.

  - On each remote database, the SYSREMOTEUSER system table contains a single row with information about the consolidated database.

  The SYSREMOTEUSER system table is maintained by the SQL Remote Message Agent (dbremote).

  On the subscriber database, the SQL Remote Message Agent (dbremote) sends a confirmation to the publisher database to ensure that the SYSREMOTEUSER system table is maintained correctly at each end of the subscription.

  See "ISYSREMOTEUSER system table" [*SQL Anywhere Server - SQL Reference*].

- **The information in the header of the messages** The SQL Remote Message Agent (dbremote) reads the header information in the messages and uses this information to update the SYSREMOTEUSER system table. A message includes the following information in its header:

  - **Its resend_count** A counter that keeps track of the number of times the receiving database has lost messages.

    In the following example, the resend_count is 1.

    ```
    Current message's header: (1-0000942712-0001119170-0)
    ```

  - **The transaction log offset of the last COMMIT in the previous message** In the following example, the transaction log offset of the last commit in the previous message is 0000942712.

    ```
    Previous message's header:(0-0000923357-0000942712-0)
    Current message's header: (0-0000942712-0001119170-0)
    ```

  - **The transaction log offset of the last COMMIT in the current message** In the following example, the last commit in the current message is 0001119170:

    ```
    Current message's header: (0-0000942712-0001119170-0)
    ```

    If a transaction spans several messages, both transaction log offsets can be identical until the final message contains a COMMIT.

    In the following example, the COMMIT does not occur until the fourth message:

    ```
    (0-0000942712-0000942712-0)
    (0-0000942712-0000942712-1)
    (0-0000942712-0000942712-2)
    (0-0000942712-0001119170-3)
    ```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- **A sequence number** When a transaction spans several messages, this sequence number is used to order the messages correctly.

  A sequence number of zero can indicate that:

  - The message is not part of a multi-part message if the transaction log offsets are different.

    In the following example, the messages are not part of a multi-part message:

    ```
    (0-0000923200-0000923357-0)
    (0-0000923357-0000942712-0)
    ```

  - The message is the first of a multi-part message if the transaction log offsets are the same.

    In the following example, the first message is part of a multi-part message:

    ```
    (0-0000942712-0000942712-0)
    (0-0000942712-0000942712-1)
    (0-0000942712-0000942712-2)
    (0-0000942712-0001119170-3)
    ```

Ensuring that operations are applied in the correct order

To ensure that the replicated statements are applied in the correct order, the Guaranteed Message Delivery System uses the transaction log offsets 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 transaction log offset values.

Each message includes the following transaction log offsets:

- The transaction log offset of the last COMMIT in the previous message. If a transaction spans several messages, there is a sequence number within the transaction to order the messages correctly. "Ensuring that operations are applied in the correct order" on page 95.

- The transaction log offset of the last COMMIT in the message.

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, a sequence number within the transaction is used to order the messages correctly.

Sending messages

The log_sent column in the SYSREMOTEUSER system table holds the local transaction log offset for the last message sent to the subscriber.

The following describes how the SYSREMOTEUSER system tables are updated when messages are sent.

1. When the publisher SQL Remote Message Agent (dbremote) sends a message to a subscriber, it also sets the log_sent value to the transaction log offset value of the last COMMIT in the sent message.

For example, the publisher sends the following message to user1.

```
(0-0000923200-0000923357-0)
```

In the publisher's SYSREMOTEUSER system table, the publisher sets the log_sent value to 0000923357 for user1.

2. When the message is received and applied at the subscriber database, a confirmation is sent to the publisher. The confirmation includes the last transaction log offset that was applied by the subscriber database.

   For example, the message confirms that user1 applied all of the transactions up to and including the transaction log offset 0000923357.

3. When the publisher SQL Remote Message Agent (dbremote) receives the confirmation, it sets the confirm_sent column to the value of the confirmation offset for the user in the SYSREMOTEUSER system table.

   For example, the publisher sets the confirm_sent column to 0000923357 for user1 in the publisher's SYSREMOTEUSER system table.

Both the log_sent and confirm_sent values contain transaction log offsets of the publisher's transaction log. The confirm_sent value cannot be a later offset than log_sent value.

Receiving messages

The following describes how the SYSREMOTEUSER system tables are updated when messages are received.

1. When the SQL Remote Message Agent (dbremote) at a subscriber database receives and applies a replication update, it updates the log_received column in the SYSREMOTEUSER system table with the offset of the last COMMIT in the message.

   For example, when the subscriber receives and applies the following message, the log_received value in the SYSREMOTEUSER system table is set to 0000923357.

   ```
   (0-0000923200-0000923357-0)
   ```

   The log_received column at any subscriber database contains a transaction log offset that exists in the publisher database transaction log.

2. When the operations are received and applied, the subscriber SQL Remote Message Agent (dbremote) sets the confirm_received value in its SYSREMOTEUSER system table, and then sends confirmation to the publisher database.

See also

- “Resending lost or corrupt messages” on page 97
- “Ensuring that messages are applied only once” on page 97

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Resending lost or corrupt messages

The SYSREMOTEUSER system table contains two columns that manage resending messages:

- **resend_count column** A counter that keeps track of the number of times that the subscriber database has lost messages.
- **rereceive_count column** A counter that keeps track of the number of times the SQL Remote Message Agent (dbremote) has determined that messages from a publisher user have been lost.

When messages are received in the proper order at a subscriber database:

1. The subscriber SQL Remote Message Agent (dbremote) applies the messages in the correct order and updates its SYSREMOTEUSER system table.
2. The subscriber SQL Remote Message Agent (dbremote) sends a confirmation message to the publisher.
3. When the publisher receives the confirmation, its SQL Remote Message Agent (dbremote) updates its SYSREMOTEUSER system table.

When messages are not received in the proper order:

1. The subscriber SQL Remote Message Agent (dbremote) sends a resend request and increments the rereceive_count value in its SYSREMOTEUSER system table.
2. When the publisher receives the resend request, it increments the resend_count value in its SYSREMOTEUSER system table for the subscriber.
3. In the publisher's SYSREMOTEUSER system table, the log_sent value is set to the value in the confirm_sent column. Resetting of the log_sent value causes operations to be resent.

See also

- “Ensuring that operations are applied in the correct order” on page 95
- “Ensuring that messages are applied only once” on page 97

Ensuring that messages are applied only once

The subscriber SQL Remote Message Agent (dbremote) compares the resend_count value in a messages header with the rereceive_count in its local SYSREMOTEUSER system table. If the resend_count value is smaller than rereceive_count, the message is not applied; it is deleted. This behavior ensures that operations are not applied more than once.

See also

- “Ensuring that operations are applied in the correct order” on page 95
- “Resending lost or corrupt messages” on page 97

Controlling message size

The following section discusses the message encoding and compression scheme in SQL Remote Message Agent (dbremote).

The SQL Remote Message Agent provides the following encoding and compression features:

- **Compatibility** The system can be set to be compatible with previous versions of SQL Anywhere. See "Upgrading SQL Remote" [*SQL Anywhere 12 - Changes and Upgrading*].
- **Compression** You can select a level of compression for your messages.

  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. However, compression can take a significant amount of time. See "compression option [SQL Remote]" [*SQL Anywhere Server - Database Administration*].
- **Encoding** SQL Remote encodes messages to ensure that they pass through message systems uncorrupted. The encoding scheme can be customized to provide extra features. See "Encoding: Preventing message corruption" on page 98.

For information about SQL Remote requirements for the maximum message length, see "SQL Remote Message Agent (dbremote) requirements" on page 80.

Encoding: Preventing message corruption

SQL Remote encodes messages to ensure that they pass through message systems uncorrupted. The default message-encoding behavior of SQL Remote is as follows:

- If the message system can use binary message formats, the messages are not encoded.
- If the message system, for example SMTP, requires text-based message formats, then an encoding DLL (*dbencod12.dll*) translates the messages into a text format before sending. The message format is unencoded at the receiving end using the same DLL.

  You can customize the encoding scheme to provide extra features. See "Creating custom encoding schemes" on page 98.
- If the compression database option is set to -1, then a version 5 compatible encoding is used for all message systems. See "Upgrading SQL Remote" [*SQL Anywhere 12 - Changes and Upgrading*].

Creating custom encoding schemes

To implement a custom encoding scheme, you can build a custom encoding DLL. You can use this custom DLL to apply special features required for a particular messages system, or to collect statistics, such as how many messages are sent to each user.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

The header file *dbrmt.h* is installed in the *install-dir* directory. This file includes an application programming interface that you can use to build a custom encoding scheme.

To use your custom DLL, set the message control parameter encode_dll to a value that is the full path to the custom DLL. For example:

```
SET REMOTE FTP OPTION "Public"."encode_dll" = 'c:\\sqlany12\\Bin32\
\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.

See also

- "SET REMOTE OPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]

SQL Remote message systems

In SQL Remote replication, a message system is a protocol for exchanging messages between the consolidated database and a remote database. SQL Remote exchanges data among databases using one or more underlying message systems. SQL Remote supports the following message systems:

- **File sharing** A simple system requiring no extra software. See "The FILE message system" on page 103.
- **FTP** Internet file transfer protocol. See "The FTP message system" on page 105.
- **SMTP/POP** Internet email protocol. See "The SMTP message system" on page 106.

You choose a message system when you assign REMOTE or CONSOLIDATE permission to a user. See "Grant REMOTE permission" on page 23 and "Grant CONSOLIDATE permission" on page 26.

Each message system that is used in a SQL Remote system has control parameters and other settings that must be set up.

Not all message systems are supported on all operating systems. For a complete list of supported operating systems, see "Supported platforms" [*SQL Anywhere 12 - Introduction*].

Setting up a message system

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

Each message type definition includes the message system type name (**FILE**, **FTP**, or **SMTP**) and the address of the publisher under that message type.

The address supplied with a message type definition is closely tied to the publisher ID of the database.

Extraction utility (dbxtract)

The publisher address at a consolidated database is used by the Extraction utility (dbxtract) and the **Extract Database Wizard** as a return address when creating remote databases. It is also used by the SQL Remote Message Agent (dbremote) to identify the location of incoming messages for the FILE system.

See also

- "The FILE message system" on page 103
- "The FTP message system" on page 105
- "The SMTP message system" on page 106

Create message types

To add a message type (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.

2. In the left pane, expand the **SQL Remote Users** directory.

3. In the right pane, click the **Message Types** tab.

4. Choose **File** » **New** » **Message Type**.

5. In the **What Do You Want To Name The New Message Type** field, type a name for the message type. The name should correspond to a message-type DLL already installed in your SQL Anywhere installation directory. Click **Next**.

6. In the **What Is The Publisher Address** field, type a publisher address. Click **Finish**.

To create a message type (SQL)

1. Verify that you have created an address for the publisher under the message type.

2. Execute a CREATE REMOTE MESSAGE TYPE statement.

   ```
   CREATE REMOTE MESSAGE TYPE message-type ADDRESS publisher-address
   ```

   For example:

   ```
   CREATE REMOTE MESSAGE TYPE FILE
   ADDRESS 'company';
   ```

Create a remote message type on Windows Mobile

If you have Windows Mobile services installed, you can set up SQL Remote for ActiveSync synchronization from Sybase Central. This option sets your directory for FILE message link messages to be the Microsoft ActiveSync directory. When you dock your Windows Mobile device to your desktop computer, Microsoft ActiveSync keeps the files in your desktop computer's Microsoft ActiveSync directory synchronized with those in the Windows Mobile ActiveSync directory.

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

To set up SQL Remote ActiveSync synchronization

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.

2. From **Tools**, choose **SQL Anywhere 12** » **Edit Windows Mobile Message Type**.

See also

- "CREATE REMOTE MESSAGE TYPE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Alter message types" on page 101
- "Delete message types" on page 101

Alter message types

To change address of a publisher, alter its 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 remote message type (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.

2. In the left pane, expand the **SQL Remote Users** directory for a database.

3. In the right pane, click the **Message Types** tab.

4. In the right pane, right-click the message type you want to alter and choose **Properties**.

5. Update the message type properties and click **OK**.

To alter a remote message type (SQL)

1. Connect to a database as a user with DBA authority.

2. Make sure you have decided on a new address for the publisher under the message type.

3. Execute an ALTER REMOTE MESSAGE TYPE statement.

See also

- "ALTER REMOTE MESSAGE TYPE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Create message types" on page 100
- "Delete message types" on page 101

Delete message types

Deleting a message type removes the publisher address from the definition.

To delete a message type (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, expand the **SQL Remote Users** directory for a database.
3. In the right pane, click the **Message Types** tab.
4. In the right pane, right-click the message type you want to remove and choose **Delete**.
5. Click **Yes**.

To delete a message type (SQL)

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

See also

- "DROP REMOTE MESSAGE TYPE statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Create message types" on page 100
- "Alter message types" on page 101

Set remote message type control parameters

The message control parameters are held in the database. Use the following procedure to set the control parameter.

To set a message control parameter (SQL)

- Execute a SET REMOTE OPTION statement.

  For example, the following statement sets the FTP host to *ftp.mycompany.com* for the FTP link for user myuser:

  ```
  SET REMOTE FTP OPTION myuser.host = 'ftp.mycompany.com';
  ```

  See "SET REMOTE OPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

To view the message link parameters (SQL)

- Query the SYSREMOTEOPTION system view.

  For example:

  ```
  SELECT * from SYSREMOTEOPTION;
  ```


Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

See “SYSREMOTEOPTION system view” [*SQL Anywhere Server - SQL Reference*].

Message link parameters stored on disk

Earlier versions of SQL Remote stored the message link parameters outside the database. The external storage of message link parameters is not recommended.

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

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

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

  The parameters for each message link are stored in a key under the SQL Remote key, with the name of the message link (4, *SMTP*, for example).

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

The SQL Remote environment variable stores a path that can be used as an alternative to one of the control parameters for the FILE messaging system.

When the SQL Remote Message Agent (dbremote) loads a message link, the link uses the setting of the current publisher or, if a setting is not specified, of groups to which the publisher belongs.

On Windows, when the SQL Remote Message Agent (dbremote) that supports storing the message link parameters in the database is run for the first time, it copies the link options from the registry to the database.

See also

- “SET REMOTE OPTION statement [SQL Remote]” [*SQL Anywhere Server - SQL Reference*]

The FILE message system

SQL Remote can be used even if you do not have an email or FTP 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, 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 a Windows service, the account under which the SQL Remote Message Agent (dbremote) is running must have permissions to read from and write to all necessary directories. If the correct permissions are not assigned, the SQL Remote Message Agent is unable to access network drives.

Root directory for addresses

Typically, the FILE message system addresses are subdirectories of a shared directory that is available to all SQL Remote users, whether by modem or 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 computers. You can use a simple file transfer mechanism to exchange files and complete replication.

FILE message control parameters

The FILE message system uses the following control parameters:

- **Directory**   The directory under which the messages are stored. The setting is an alternative to the SQLREMOTE environment variable.
- **Debug**   The setting for this parameter is either YES or NO. The default is NO. When set to YES, all FILE system calls made by the FILE link are displayed.
- **Encode_dll**   If you are using a custom encoding scheme, you must set this parameter to the full path of the custom encoding DLL that you created. See "Controlling message size" on page 98.
- **invalid_extensions**   A comma-separated list of file extensions that you do not want the SQL Remote Message Agent (dbremote) to use when generating files in the messaging system.
- **Unlink_delay**   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 set 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.

Windows Mobile and Microsoft ActiveSync

The SQL Remote Message Agent (dbremote) searches in *C:\My Documents\Synchronized Files* for the FILE link. On the consolidated database computer, the SQLREMOTE environment variable or directory message link parameter for the FILE link should be set to the following directory where *userid* and *Windows-mobile-device-name* are set to the appropriate values:

```
%SystemRoot%\Profiles\userid\Personal\Windows-mobile-device-name
\Synchronized Files
```

With this system, Microsoft ActiveSync automatically synchronizes the message files between the consolidated database computer and the Windows Mobile device.

To verify that FILE synchronization is activated, check **Mobile Devices** » **Tools** » **ActiveSync Options**.

For information on setting message link parameters, see "The FILE message system" on page 103.

See also

- "The FTP message system" on page 105

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

The FTP message system

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 platforms" [*SQL Anywhere 12 - Introduction*].

FTP message control parameters

The FTP message system uses the following control parameters:

- **host** The host name of the computer where the FTP server is running. This parameter 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** The IP port number used for the FTP connection. This parameter is usually not required.
- **debug** This parameter is set either to YES or NO. The default is NO. When set to YES, debugging output is displayed.
- **active_mode** This parameter is set either to YES or NO. The default is 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. When you set this parameter, only reconnections are affected. 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. When you set this parameter, only reconnections are affected. The initial connection made by the FTP link is not affected.
- **suppress_dialogs** This parameter is set to TRUE or FALSE. If set to TRUE, the **Connect** window 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.
- **encode_dll** If you have implemented a custom encoding scheme, you must set this parameter to the full path of the custom encoding DLL that you created.

See also

- "Controlling message size" on page 98

Troubleshooting FTP problems

Most problems with the FTP message link are caused by network system issues. This section provides a list of tests you can use to troubleshoot problems.

- **Set the DEBUG message control parameter** Review the debug output to determine whether you are connecting to the FTP server. If you are connecting, the debug output should indicate which FTP commands are failing.

- **Ping the FTP server** If the FTP link is not able to connect to the FTP server, test your system network configuration. For example, run the following command:

  ```
  ping FTP-server-name
  ```

  The IP address of the FTP server and the ping (round trip) time to the FTP server should be returned. If you cannot ping the FTP server, then you have a network configuration problem, and you should contact you 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 FTP server initiates all data connections. If your FTP server is behind an incorrectly configured firewall, you may not be able to use the default passive transfer mode because 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 or download a file. If the client you are using cannot transfer the file without using active mode then you should 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 behind an IP masquerading gateway incoming connections may fail depending on your 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 required directories 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 appear, the root_directory control parameter may be set incorrectly 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 errors are returned, your FTP server permissions are set up incorrectly.

The SMTP message system

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

With the SMTP system, SQL Remote sends messages using Internet mail. The messages are encoded in 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.

For a list of operating systems for which SMTP is supported, see "Supported platforms" [*SQL Anywhere 12 - Introduction*].

SMTP addresses and user IDs

To use SQL Remote and an SMTP message system, each database participating in the system requires an 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 email server.

> **Separate email account recommended**
> It is recommended you use a separate POP email account to send and receive SQL Remote messages. See "Sharing SMTP/POP addresses" on page 108.

SMTP message control parameters

Before the SQL Remote Message Agent (dbremote) connects to the message system to send or receive messages, the user must have a set of control parameters already set on their computer, or the user is prompted to specify needed information. This information is needed only on the first connection. It is saved and used as the default entries for subsequent connections.

The SMTP message system uses the following control parameters:

- **local_host** The name of the local computer. It is useful on computers 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. When you set the TOP_supported parameter to NO, SQL Remote uses the RETR command, which is less efficient but works with all POP servers. The default is YES.

- **smtp_authenticate** Determines whether the SMTP link authenticates the user. The default value is YES. Set this parameter to NO to turn off SMTP authentication.

- **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** The name of the computer on which the SMTP server is running. It corresponds to the SMTP host field in the SMTP/POP3 login window.

- **pop3_host** The name of the computer on which the POP host is running. Typically, it is the same name as the SMTP host. It corresponds to the POP3 host field in the SMTP/POP3 login window.
- **pop3_userid** The user ID used to retrieve mail. The POP user ID corresponds to the user ID field in the SMTP/POP3 login window. You must obtain a user ID from your POP host administrator.
- **pop3_password** The password used to retrieve mail. It corresponds to the password field in the SMTP/POP3 login window.
- **Debug** When set to YES, all SMTP and POP3 commands and responses are displayed. This information can be used for troubleshooting SMTP/POP support problems. The default is NO.
- **Suppress_dialogs** When this parameter is set to true, the **Connect** window does not appear after failed attempts to connect to the mail server. Instead, an error is generated.
- **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 "Controlling message size" on page 98.

Sharing SMTP/POP addresses

You should use a separate email account for SQL Remote messages. It is not recommended that you send and receive SQL Remote messages through the same email account that you use for personal or business email messages.

If you need to share the same email account for SQL Remote messages and regular email messages, then you must ensure that your email program does not download and delete all messages from the mail server, including SQL Remote email and personal messages. You must configure the email program so that it does not alter or delete SQL Remote messages when it downloads your regular email messages. SQL Remote messages contain the subject ---SQL Remote--.

Troubleshooting SMTP Link

If you cannot get the SMTP Link to work, connect to the SMTP/POP3 server from the same computer on which the SQL Remote Message Agent (dbremote) is running, using the same account and password. Use an Internet email program that supports SMTP/POP3. Disable this program once the SMTP message link is working.

Check that email is working properly

If SQL Remote messages are not being sent and received properly and you are using an email message system, you should confirm that email is working properly between the two computers.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Backing up SQL Remote systems

SQL Remote replication depends on access to operations in the transaction log, and access to old transaction logs. Any backup strategy that you implement must incorporate the maintenance of the transaction logs for SQL Remote.

The SQL Remote Message Agent (dbremote) must have access to the current and old transaction logs until they are no longer needed.

A consolidated database no longer needs its transaction logs when all remote databases have received and have confirmed that the messages contained in the transaction logs have been successfully applied.

A remote database no longer needs its transaction logs when the consolidated database has received and confirmed that it has successfully applied the messages contained in the transaction logs.

Backing up remote databases

For your remote databases, you need to decide whether to:

- **Rely on replication to the consolidated database as a backup method** Backup procedures are not as crucial on remote databases as on the consolidated database. You can rely on replication to the consolidated database as a data backup method.

  If you choose this method, then you *should* create a strategy for maintaining the remote database transaction logs. See "Maintaining transaction logs for remote databases" on page 110.

- **Create a backup strategy for the remote database** If the changes made on the remote databases are crucial, then you need to create a backup strategy for the remote databases that includes the maintenance of the transaction logs. See "Back up remote databases" on page 110.

Backing up consolidated databases

You *must* have a backup strategy for your consolidated database that includes the maintenance of the transaction logs. See "Protecting the consolidated database from media failures" on page 111.

The Backup utility (dbbackup) and the SQL Remote Message Agent (dbremote) -x option

On a database, you should never run both the SQL Remote Message Agent (dbremote) with the -x option and the Backup utility (dbbackup).

The -x option is used to manage transaction logs for replication. The -x option ensures that SQL Remote Message Agent has access to old transaction logs and deletes the transaction logs when they are no longer needed. The -x option does not back up the transaction log.

The Backup utility (dbbackup) is used to back up the current transaction log. When the Backup utility (dbbackup) is run with the -r and -n options, it backs up the current transaction log to a backup directory and renames and restarts the current transaction log. The Backup utility (dbbackup) assumes that the current transaction log is the same transaction log that it renamed and restarted after the previous back up.

If you try to run both the SQL Remote Message Agent -x option and the Backup utility (dbbackup) on the same database, they interfere with each other. You can lose transaction logs when both are running.

Only run the SQL Remote Message Agent (dbremote) with the -x option on a remote database that is not being backed up.

Maintaining transaction logs for remote databases

Use the following procedure to maintain your remote database transaction logs when you are relying on replication to the consolidated database to back up your remote databases. That is, you are *not* running the Backup utility (dbbackup) on the remote databases and transaction logs.

> **Caution**
> Do *not* run the SQL Remote Message Agent (dbremote) with the -x option on a database that is being backed up.

To maintain remote database transaction logs

1. On the remote database, run the SQL Remote Message Agent (dbremote) with the -x option and specify a size for the transaction log. This option causes the SQL Remote Message Agent (dbremote) to rename and restart the transaction log when the transaction log exceeds the specified size.

   The following deletes the transaction log when it is larger than 1 MB:

   ```
   dbremote -x 1M  -c "UID=ManagerSteve;PWD=sql;DBF=c:\mydata.db"
   ```

2. On the remote database, set the delete_old_logs option to On. Setting the delete_old_logs_option causes the old transaction log files to be deleted automatically by the SQL Remote Message Agent (dbremote) when they are no longer needed for replication.

   A transaction log is no longer needed when all subscribers have confirmed that they have received and successfully applied all the changes recorded in that transaction log file. You can set the delete_old_logs option either for the PUBLIC group or just for the user contained in the SQL Remote Message Agent (dbremote) connection string.

   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';
   ```

Back up remote databases

Use the following procedure to back up your remote databases. This procedure includes a maintenance strategy for the use of the transaction logs by SQL Remote. Do *not* use this procedure and run the SQL Remote Message Agent (dbremote) with the -x option.

To back up remote databases using the Backup utility (dbbackup)

1. Make a full backup of the remote database.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

a. Connect to the database as a user with DBA authority.

b. Run dbbackup with the -r and -n options.

For example, assume that the backup directory is *e:\live*, the database file is located in the *c:\live* directory and its corresponding transaction log file is located in the *d:\live* directory:

```
dbbackup -r -n -c "UID=DBA;PWD=sql;DBF=c:\live\remotedatabase.db" e:
\archive
```

The transaction logs in the *d:\live* directory are not altered by the full backup.

c. Copy the backup files located in the *e:\archive* directory to an off-site drive or to a DVD.

d. Run the SQL Remote Message Agent (dbremote) with access to the current transaction log files using the following command:

```
dbremote -c "UID=DBA;PWD=sql;DBF=c:\live\database.db" d:\live
```

Caution
Do *not* run the SQL Remote Message Agent (dbremote) with the -x option on a database that is being backed up.

2. Set up the Backup utility (dbbackup) to make incremental backups of the remote database's transaction log.

a. Connect to the database as a user with DBA authority.

b. Run dbbackup with the -r, -n, and -t options.

For example:

```
dbbackup -r -n -t -c "UID=DBA;PWD=sql;DBF=c:\live\remotedatabase.db" e:
\archive
```

c. Run the SQL Remote Message Agent (dbremote) with access to the current transaction log files using the following command:

```
dbremote -c "UID=DBA;PWD=sql;DBF=c:\live\remotedatabase.db" d:\live
```

Protecting the consolidated database from media failures

To protect your SQL Remote replication system against media failure:

- **Replicate only backed-up transactions** Send messages that contain only backed-up transactions. By sending only backed-up transactions, the replication system is protected against media failure on the transaction log. You can accomplish this by:
  - Running the SQL Remote Message Agent (dbremote) with the -u option. When the SQL Remote Message Agent (dbremote) is run with the -u option, only committed transactions that have been backed up are packaged into messages to be sent.

    The -u option provides additional protection against total site failure, if backups are carried out to another site. See “SQL Remote Message Agent utility (dbremote)” on page 131.

- **Use a transaction log mirror** Using a transaction log mirror protects against media failure on the transaction log device. See "Transaction log mirrors" [*SQL Anywhere Server - Database Administration*].

- **Do not run SQL Remote Message Agent (dbremote) with the -x option on the consolidated database** Never run the SQL Remote Message Agent (dbremote) with the -x option against a database that is being backed up. The -x option maintains the transaction logs for replication, not for backup or recovery. To back up your consolidated database, see "Back up the consolidated database" on page 112.

Back up the consolidated database

Back up your consolidated database by making a full backup of the consolidated database and transaction log, and then make incremental backups of the transaction log.

To back up SQL Remote consolidated databases

1. Make a full back up of the consolidated database and its transaction log.
   a. Connect to the database as a user with DBA authority.
   b. Run dbbackup with the -r and -n options.

      For example:

      ```
      dbbackup -r -n -c "UID=DBA;PWD=sql;DBF=c:\live\database.db" e:\archive
      ```

2. Make incremental backups of the consolidated database transaction log. When backing up the transaction log, choose to rename and restart the transaction log.
   a. Connect to the database as a user with DBA authority.
   b. Run dbbackup with the -r and -n and -t options.

      For example:

      ```
      dbbackup -r -n -t -c "UID=DBA;PWD=sql;DBF=c:\live\database.db" e:
      \archive
      ```

3. Run the SQL Remote Message Agent (dbremote) with access to the current transaction log.

   For example:

   ```
   dbremote -c "UID=DBA;PWD=sql;DBF=c:\live\database.db" d:\live
   ```

> **Caution**
> Do *not* run the SQL Remote Message Agent (dbremote) with the -x option on a database that is being backed up.

The figure below illustrates a database named *database.db* in the *c:\live* directory, with a transaction log named *database.log* in the *d:\live* directory.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

When you back up the transaction log to a backup directory *e:\archive* using the -r and -n options to rename and restart the transaction log the Backup utility (dbbackup) carries out the following tasks:

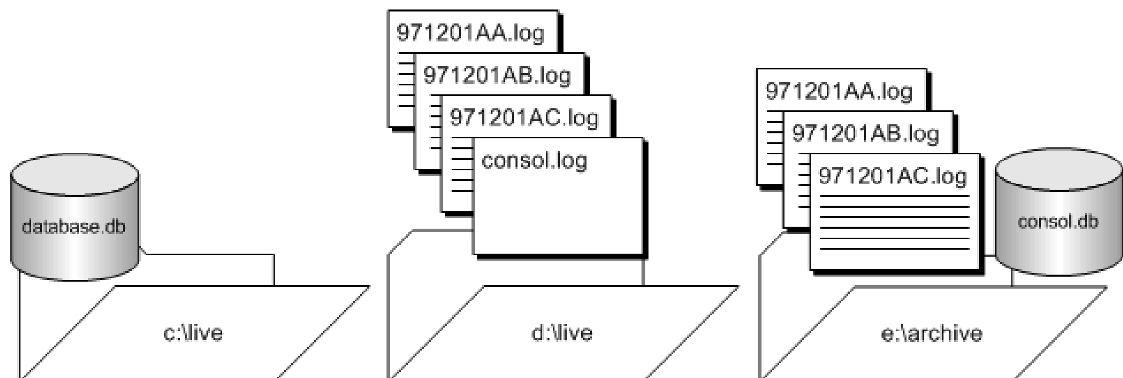
1. Renames the current transaction log file to *971201xx.log*, where *xx* are sequential characters ranging from *AA* to *ZZ*.

2. Backs up the transaction log file to the backup directory, creating a backup file named *971201xx.log*.

Old transaction log names before 8.0.1
Before the release of SQL Anywhere 8.0.1, the old transaction log files were named *yymmdd01.log*, *yymmdd02.log*, and so on. The name change was introduced to allow more old transaction logs to be stored. As the SQL Remote Message Agent (dbremote) scans all the files in the specified directory, regardless of their names, the name change should not affect existing applications.

3. Starts a new transaction log, as *database.log*.

After several backups, the live directory and the archive directory contain a set of sequential transaction logs.



See also

- "Protecting the consolidated database from media failures" on page 111
- "Rename the backup copy of the transaction log during backup" [*SQL Anywhere Server - Database Administration*]
- "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*]

Recover consolidated databases manually

The following procedure describes how to recover a consolidated database by applying each transaction log to the database. To have the SQL Anywhere database server automatically recover the consolidated database, see "Recover consolidated databases automatically" on page 115.

To recover the database using the -a option

1. Make a copy of the database and transaction log file. This procedure assumes that previous backups of the database file have been made and are available, for example on tape.

2. Create a temporary directory.

3. Restore the most recent back up of the database (*.db*) file, *not* the transaction log file, from tape into a temporary directory.

   In the temporary directory:

   a. Start the backup copy of the database.
   b. Apply the old transaction logs using the -a option.
   c. Shut down the database.
   d. Start the database using the current transaction log and the -a option to apply the transactions and bring the database file up to date.
   e. Shut down the database.
   f. Back up the database.

4. Copy the database to the production directory.

5. Start the database.

   Any new activity is appended to the current transaction log.

Example: Applying transaction logs individually

Suppose you have a consolidated database file named *c:\dbdir\cons.db*, a transaction log file *c:\dbdir\cons.log*, and a transaction log mirror file *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 -n -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 run the SQL Remote Message Agent (dbremote) with the optional directory to point to the renamed transaction log files. For example:

```
dbremote -c "UID=DBA;PWD=sql" c:\dbdir
```

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

On the third day following the weekly backup, the database file is corrupted because of a bad disk block.

Use the following procedure to recover from a media failure.

To recover from media failure on the C drive

1. Back up the transaction log mirror file *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:

   ```
   dbeng12 -a c:\dbdir\dateAA.log c:\recover\cons.db
   dbeng12 -a c:\dbdir\dateAB.log c:\recover\cons.db
   ```

5. Copy the current transaction log, *d:\mirdir\cons.log* to the recovery directory, giving *c:\recover\cons.log*.
6. Start the database using the following command:

   ```
   dbeng12 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.

See also

- "-a dbeng12/dbsrv12 database option" [*SQL Anywhere Server - Database Administration*]

Recover consolidated databases automatically

The following procedure describes how to automatically recover a consolidated database. To apply the transaction logs manually, see "Recover consolidated databases manually" on page 114.

To recover the database using the -ad option

1. Make a copy of the database and transaction log file. This procedure assumes that previous backups of the database file have been made and are available, for example on tape.

2. Restore the most recent backed up copy of the database (*.db*) file, *not* the transaction log file, from tape into a temporary directory.

3. In the temporary directory:

   a. Start the database, applying the transaction logs using the -ad option.

      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 transaction log offsets.

   b. Copy the current transaction log, to the temporary directory.

   c. Start the database and apply the current transaction log.

   d. Shut down the database server.

   e. Back up the database and transaction log.

4. Copy database and transaction log files to the appropriate production directories.

5. Restart your system as normal.

   Any new activity is appended to the current transaction log.

Example

Suppose you have a consolidated database file named *c:\dbdir\cons.db*, a transaction log file *c:\dbdir\cons.log*, and a transaction log mirror file *d:\mirdir\cons.mlg*.

Assume that you perform full backups weekly using the following command:

```
dbbackup -c "UID=DBA;PWD=sql" -r -n e:\backdir
```

Assume that you also perform incremental backups daily using the following command:

```
dbbackup -c "UID=DBA;PWD=sql" -r -n -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 run the SQL Remote Message Agent (dbremote) with the optional directory to point to the renamed transaction log files. For example:

```
dbremote -c "UID=DBA;PWD=sql" c:\dbdir
```

On the third day following the weekly backup, the database file is 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 transaction log mirror file *d:\mirdir\cons.mlg*.

3. Create a temporary directory to perform the recovery in. In this example, it is called *c:\recover*.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

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:

```
dbeng12 c:\recover\cons.db -ad c:\dbdir
```

7. Copy the current transaction log, *d:\mirdir\cons.log* to the recovery directory, giving *c:\recover\cons.log*.

8. Start the database using the following command:

```
dbeng12 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.

See also

- "-ad dbeng12/dbsrv12 database option" [*SQL Anywhere Server - Database Administration*]

Reporting and handling replication errors

The following errors can occur on a SQL Remote system:

- **Row not found errors** See "Row not found errors" on page 47.

- **Referential integrity errors** See "Referential integrity errors" on page 47.

- **Duplicate primary key errors** See "Duplicate primary key errors" on page 50.

By default, when an error occurs, the SQL Remote Message Agent (dbremote) prints the error in its log output window. The SQL Remote Message Agent (dbremote) can print more information in the output messages file than in the messages window.

The SQL Remote Message Agent (dbremote) messages log file includes the following information:

- Applied messages
- Failed SQL statements
- Other errors

To print an error to the output log file, run the SQL Remote Message Agent (dbremote) with the -o option. See “SQL Remote Message Agent utility (dbremote)” on page 131.

When an error occurs, you can configure SQL Remote to:

- **Run an error-handling procedure** By default, no procedure is called. However, you can use the replication_error database option to specify a stored procedure to be called by the SQL Remote Message Agent (dbremote) when an error occurs. See “Run an error-handling procedure” on page 118.

  For example, you can configure SQL Remote to:

  - Send portions of a remote database's output log to the consolidated database and written to a file. See “Collect errors from the remote database” on page 118.
  - Send an email notification when an error occurs at a remote database. See “Receive email notification about remote database errors” on page 119.

- **Ignore the error** There might be instances when you do not want to the SQL Remote Message Agent (dbremote) to report an error. For example, you can choose to ignore an error when you know the conditions under which the error occurs and you are sure that the error does not produce inconsistent data. See “Ignoring replication errors” on page 122.

Run an error-handling procedure

Set the replication_error option to call a procedure when a SQL error occurs. By default, no procedure is called when a SQL error occurs.

The procedure that is called must have a single argument type of 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. See “replication_error option [SQL Remote]” [*SQL Anywhere Server - Database Administration*].

To set the replication_error option

- Execute the following statement. The *remote-user* is the publisher name on the SQL Remote Message Agent (dbremote) command and *procedure-name* is the procedure called when a SQL error is detected.

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

Collect errors from the remote database

Use the following procedure to send portions of a remote database's output log to the consolidated database. The information is written to a file and the file can contain output logging information from some or all remote databases in the system.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

To configure SQL Remote to collect output log information from remote databases

1. Configure the remote databases to send output log information to the consolidated database.

   a. Use the SET REMOTE statement with the output_log_send_on_error option to send log information when an error occurs.

      On the remote database, execute the following command:

      ```
      SET REMOTE link-name OPTION
      PUBLIC.output_log_send_on_error = 'Yes';
      ```

      When the SQL Remote Message Agent (dbremote) reads any messages that start with the error indicator **E**, it sends the output log information to the consolidated database. See "SET REMOTE OPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   b. This step is optional. Set the SET REMOTE statement with the output_log_send_limit option to limit the amount of information that is sent to the consolidated database. The output_log_send_limit option specifies the number of bytes at the end of the output log (that is, the most recent entries) that are sent to the consolidated database. The default is 5K.

      If you supply an output_log_send_limit value that exceeds the maximum message size, SQL Remote overrides the output_log_send_limit value and sends only what can fit within the maximum message size.

      On the remote database, execute the following command:

      ```
      SET REMOTE link-name OPTION
      PUBLIC.output_log_send_limit = '7K';
      ```

      See "SET REMOTE OPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

2. Configure the consolidated database to receive log information.

   On the consolidated database, run the SQL Remote Message Agent (dbremote) with either the -ro or the -rt options.

   See "SQL Remote Message Agent utility (dbremote)" on page 131.

3. This step is optional. To test your configurations, set the output_log_send_now option to send output log information to the consolidated database.

   On the remote database, set the output_log_send_now option to YES.

   On the next poll, the remote database sends the output log information and then resets the output_log_send_now option to NO.

See also

- "Receive email notification about remote database errors" on page 119

Receive email notification about remote database errors

Use the following procedure to send email notification when an error occurs at a remote database. You can use email or a paging system to receive the notifications.

To set up SQL Remote to send email notification of errors (SQL)

1. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as user Cons.

2. Create a stored procedure that notifies the DBA user by email that an error has occurred.

   For example, execute the following to create the sp_LogReplicationError procedure:

```
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 SQL Remote Message Agent is passing to the
procedure.
 IF CURRENT PUBLISHER = 'cons' THEN
  CALL sp_notify_DBA( error_text );
 END IF
END;
```

3. Create a stored procedure that manages the sending of email.

   For example, execute the following statement to create the sp_notifiy_DBA procedure:

```
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; Smith, Elton',
   subject='SQL Remote Error',
   "message"=msg);
 //If mail sent successfully, stop
  IF rc=0 THEN
   call xp_stopmail()
  END IF
 END IF
END;
```

4. Set the replication_error database option to call the procedure that notifies the DBA by email that an error occurs.

   For example, execute the following statement to call the sp_LogReplicationError procedure when an error occurs:

```
SET OPTION PUBLIC.replication_error =
 'cons.sp_LogReplicationError';
```

5. Create an audit table.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

For example, execute the following to create the replication_audit table:

```
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 following table describes the columns of the replication_audit table:

Column	Description
pub	Current publisher of the database (identifies the database in which the publisher was inserted).
remoteuser	Remote user applying the message (identifies the database from which the remote user came from).
errormsg	Error message passed to the replication_error procedure.

6. Test your procedures.

   For example, insert a row on the consolidated database that uses the same primary key as a row on a remote database. When this row from the consolidated database is replicated to the remote database, a primary key conflict error occurs and:

   - The remote database SQL Remote Message Agent (dbremote) prints the following message to its output log:

     ```
     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 following INSERT statement is sent to the consolidated database:

     ```
     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;
     ```

   - An email is sent to John Doe and Elton Smith with the following message:

```
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')
```

See also

- "Collect errors from the remote database" on page 118

Ignoring replication errors

To ignore replication errors (SQL)

- Create a BEFORE trigger on the action that causes the known error. This trigger should signal the REMOTE_STATEMENT_FAILED SQLSTATE (5RW09) or SQLCODE (-288) value.

  For example, if you want to ignore INSERT statement errors that occur when a table is missing a referenced column, create a BEFORE INSERT trigger that signals REMOTE_STATEMENT_FAILED SQLSTATE when the referenced column does not exist. The INSERT statement fails, but this failure is not reported in the SQL Remote Message Agent (dbremote) output log. See "Remote statement failed" [*Error Messages*].

Security

Use the following features are available to protect your data.

- **REMOTE DBA authority** It is recommended that you connect to the SQL Remote Message Agent (dbremote) with a user that has REMOTE DBA authority. See "Granting REMOTE DBA authority" on page 28.
- **Database encryption** You can encrypt your database using the -ek option. See "Extraction utility (dbxtract)" on page 140.
- **Message encryption** The SQL Remote Message Agent (dbremote) uses a simple encryption algorithm to protect the messages against casual snooping. However, this encryption scheme is not intended to provide full protection against determined efforts to decipher them. For information about database encryption, see "Encrypting and decrypting a database" [*SQL Anywhere Server - Database Administration*].

Upgrading and resynchronization

Use caution when upgrading a SQL Remote system. You can upgrade a SQL Remote system in any of the following ways:

- **Upgrading software** For information about upgrading SQL Remote, see "Upgrading SQL Remote" [*SQL Anywhere 12 - Changes and Upgrading*].

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- **Changing the database schemas** To make changes to the database schema, you can:
  - **Use passthrough mode** The passthrough mode allows schema changes to be sent to some or all databases in a SQL Remote system, but it requires careful planning and execution. See "SQL Remote passthrough mode" on page 124.
  - **Re-synchronize subscriptions** Re-synchronization involves copying new copies of the data to the remote databases. When there are many remote databases, resynchronization can be a time-consuming process involving work interruptions and possible data loss. See "Resynchronizing subscriptions" on page 126.

Changes to avoid on a running system

The following changes should not be made to a deployed and running SQL Remote system, except under the conditions stated:

- **Changing publishers** Problems can occur if you change the publisher user name on a consolidated database of a deployed and running SQL Remote system. If you need to change the consolidated database publisher user name, you must shut down the SQL Remote system and resynchronize all remote users.

  Changing the user name of a publisher at a remote database causes problems for any subscriptions that the remote database is involved in, including the loss of information. If you need to change a remote database publisher user name, shut down the remote database and resynchronize the remote user. See "Resynchronizing subscriptions" on page 126.

- **Making restrictive changes to tables** You cannot make restrictive changes to tables. For example, do not drop a column or alter a column to disallow NULL values because messages can exist in the system that reference these columns.

- **Making permissive changes to tables** You can make permissive changes using passthrough mode. Use passthrough mode to make the changes to the remote database schema and publications. Permissive changes include adding a new table or column, adding new users, resynchronizing users, dropping users, and changing the address, message type, or send frequency for a remote user. See "SQL Remote passthrough mode" on page 124.

- **Altering publications** Publication definitions must be maintained on both the consolidated and the remote databases. Altering publications in a running SQL Remote system can cause replication errors and can lead to a loss of data in the replication system.

- **Deleting subscriptions** You can delete a subscription, but you must use passthrough mode to remove the data on the remote database. See "SQL Remote passthrough mode" on page 124.

- **Unloading and reloading databases** You must ensure that the transaction log is properly maintained. See "Rebuild databases involved in synchronization or replication" [*SQL Anywhere Server - SQL Usage*].

- **Making changes in a multi-tier hierarchy** For information about re-extracting database schemas in a multi-tier hierarchy, see "Extracting databases for a multi-tier hierarchy system" on page 76.

  For information about making schema changes using the passthrough mode, see "Passthrough mode limitations" on page 124.

SQL Remote passthrough mode

Use passthrough mode to pass standard SQL statements to a remote database where they can be executed.

You can use passthrough mode to complete the following tasks on a running SQL Remote system:

- Add new users.
- Resynchronize users.
- Delete users from the system.
- Change the address, message type, or frequency for a remote user.
- Add a column to a table.

Caution

- SQL Remote relies on each database in the system having the same objects; when a table is altered at some sites but not at others, attempts to replicate data changes fail. Additional schema changes executed on a running SQL Remote system might cause problems. See "Changes to avoid on a running system" on page 123.

- Always test your passthrough operations on a copy of the consolidated database with a copy of a remote database subscribed. Never run untested passthrough scripts on a production database.

- You should always qualify object names with the owner name. PASSTHROUGH statements are not executed on remote databases from the same user name. Object names without the owner name qualifier may not be resolved correctly.

Passthrough mode limitations

- **Passthrough works on only one level of a hierarchy** In a multi-tier SQL Remote system, it is important that passthrough statements work immediately below the current level. In a multi-tier system, passthrough statements must be entered at each consolidated database, for the level beneath it.

- **Calling procedures** When a stored procedure is called in passthrough mode using a CALL or EXEC statement:

  - The procedure must exist in the consolidated database that calls the passthrough command, even if the procedure is not executed on the consolidated database. See "PASSTHROUGH statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

  - The procedure must also exist on the remote database. The CALL or EXEC statement is replicated, but none of the statements inside the procedure is replicated. It is assumed that the procedure on the replicated database has the correct effect.

- **Control statements** Control statements such as IF and LOOP and any cursor operations, are not replicated in passthrough mode. Any statements within the loop or control structure *are* replicated. See "Control statements" [*SQL Anywhere Server - SQL Usage*].

- **Cursor operations** Operations on cursors are not replicated.

- **SQL SET OPTION statements** Static embedded SQL SET OPTION statements are not replicated. However, dynamic SQL statements are replicated. See "Static and dynamic SQL" [*SQL Anywhere Server - Programming*].

  For example, 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 ... "
  ```

- **Batch statements** Batch statements (a group of statements surrounded with a BEGIN and END) are not replicated in passthrough mode. If you try to use batch statements in passthrough mode, an error occurs.

See also

- "Resynchronizing subscriptions" on page 126

Start and stop passthrough mode

Passthrough mode is started using the PASSTHROUGH statement and it is stopped using the PASSTHROUGH STOP statement. A passthrough session refers to the statements entered between the PASSTHROUGH statements. Statements entered in a passthrough session:

- Are checked for syntax errors.

- Are executed at the consolidated database unless you supply the ONLY keyword. When ONLY is specified, the statements are sent to the remote database without being executed on the consolidated database.

  The following statement starts a passthrough session, which passes the statements to a list of two named subscribers, without being executed at the current database:

  ```
  PASSTHROUGH ONLY
  FOR userid_1, userid_2;
  ```

- Are passed to the identified subscriber database. Passthrough statements are replicated in sequence with normal replication messages, in the order in which the statements are recorded in the transaction log.

- Are executed at the subscriber database.

Direct passthrough statements

The following statement starts a passthrough session that passes the statements to all users who are subscribed to the pubname publication:

```
PASSTHROUGH ONLY
FOR SUBSCRIPTION TO [owner].pubname statement1;
```

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 stops a passthrough session for all remote users:

```
PASSTHROUGH STOP;
```

Data modification language (DML)

Passthrough mode is commonly used to send data modification statements. In this case, replicated DML statements use the *before* schema before the passthrough and the *after* schema following the passthrough.

The following example drops a table on the remote database and the consolidated database.

```
-- Drop a table on the remote database
-- and at the consolidated database
PASSTHROUGH TO Joe_Remote;
DROP TABLE CrucialData;
PASSTHROUGH STOP;
```

The following example drops a table on the remote database only.

```
-- Drop a table on the remote database only
PASSTHROUGH ONLY TO Joe_Remote;
DROP TABLE CrucialData;
PASSTHROUGH STOP;
```

See also

- "PASSTHROUGH statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "Data modification statements" [*SQL Anywhere Server - SQL Usage*]

Resynchronizing subscriptions

When you create a remote database, you extract both the schema and data from the consolidated database and use them to build the remote database. This process ensures that each database has an initial copy of the data. After deployment, you might consider resynchronizing subscriptions in the following circumstances:

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

- **After you complete significant maintenance to the consolidated database** For example, you make changes to the consolidated database, which updates every row in the database. By default, SQL Remote creates and sends update messages to each subscribed remote. These update messages could include the UPDATE, DELETE, and INSERT statements for each row.

  If you chose to synchronize the subscription using a SYNCHRONIZE SUBSCRIPTION statement, you only send the statements required to delete all the rows in the subscribed tables and the INSERT statements to insert all new rows.

- **When a remote database is out-of-step with the consolidated database** If a remote database becomes out-of-step with the consolidated database, you can try to use passthrough mode. See "SQL Remote passthrough mode" on page 124.

  If using passthrough mode doesn't work, you can synchronize the subscriptions. When you synchronize subscriptions, you force the remote database into step with the consolidated database. A SYNCHRONIZE SUBSCRIPTION statement includes statements to delete the contents of the subscribed tables in the remote database and statements to insert the rows of the subscription from the consolidated database to the remote database.

Limitations

- **Synchronization applies to an entire subscription** You cannot synchronize a single table.

- **Data loss on synchronization** Any data on the remote database that is part of the subscription, which has not been replicated to the consolidated database, is lost.

  Before synchronizing the database, use the **Unload Database Wizard** in Sybase Central or the Unload utility (dbunload) to unload or back up the remote database. See "Export data with the Unload Database Wizard" [*SQL Anywhere Server - SQL Usage*] and "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

Synchronize

It is recommended that you use either the Extraction utility (dbxtract) or the **Extract Database Wizard** to extract the data for the specified remote database and then manually load the data into the remote database.

> **Caution**
> Do not run the SQL Remote Message Agent (dbremote) when running the Extraction utility (dbxtract) or the **Extract Database Wizard**.

To synchronize a subscription (Sybase Central)

1. Shut down the SQL Remote Message Agent on the remote database and the consolidated database.

2. Use the **SQL Anywhere 12** plug-in to connect to the consolidated database as a user with DBA authority.

3. In the left pane, expand the **Publications** directory.

4. Select a publication.

5. In the right pane, click the **SQL Remote Subscriptions** tab.

6. Manually synchronize subscriptions:

   a. Right-click the user in the **Subscribers** list and choose **Properties**.

   b. Click the **Advanced** tab.

   c. Click **Synchronize Now**.

      The subscriptions are affected when you click the **Synchronize Now** button. Subsequently clicking **Cancel** on the properties window does *not* cancel the synchronize action.

7. Click **OK**.

Synchronize using the SQL Remote Message Agent (dbremote)

It is recommended that you use the Extraction utility (dbxtract) or the **Extract Database Wizard** to synchronize subscriptions. See "Synchronize" on page 127.

Extracting a large number of subscriptions, or synchronizing subscriptions to large, frequently-used tables, can slow database access. You can use the SEND AT clause to specify a time to synchronize when the consolidated database is not in heavy use. See "Setting the send frequency" on page 81.

To synchronize a subscription with the message system (Interactive SQL)

1. Connect to the consolidated database as a user with DBA authority.

2. Execute a SYNCHRONIZE SUBSCRIPTION statement. See "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

   The SQL Remote Message Agent (dbremote) on the consolidated database sends a copy of all rows in the subscription to the subscriber. The SQL Remote Message Agent (dbremote) assumes that an appropriate database schema is in place at the remote databases.

   The SQL Remote Message Agent (dbremote) on the subscriber database receives the synchronization message and it *replaces* the current contents of the subscribed tables with the new copy.

Cautions

- **Do not execute a SYNCHRONIZE SUBSCRIPTION statement on a remote database** Execute SYNCHRONIZE SUBSCRIPTION statements at the consolidated database.

- **Large volume of messages may result** Synchronizing databases over a message system can require large volumes of messages. Also, the size of the message can exceed the size of the remote database. Synchronizing many subscriptions over a message link can increase the amount of message traffic.

  Often, it is recommended that you extract the remote databases and then manually load the data.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Start subscriptions

To start a subscription (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, expand the **Publications** directory.
3. Select a publication.
4. In the right pane, click the **SQL Remote Subscriptions** tab.
5. Manually synchronize subscriptions:
   a. Right-click the user in the **Subscribers** list and choose **Properties**.
   b. Click the **Advanced** tab.
   c. Click **Synchronize Now**.

      The subscriptions are affected when you click the **Synchronize Now** button. Subsequently clicking **Cancel** on the properties window does *not* cancel the synchronize action.

To start a subscription (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a START SUBSCRIPTION statement. See "START SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

To start several subscriptions within a single transaction, use the REMOTE RESET statement. See "REMOTE RESET statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

Stop subscriptions

To stop a subscription (Sybase Central)

1. Use the **SQL Anywhere 12** plug-in to connect to the database as a user with DBA authority.
2. In the left pane, expand the **Publications** directory.
3. Select the desired publication.
4. In the right pane, click the **SQL Remote Subscriptions** tab.
5. To manually synchronize subscriptions, select the user in the Subscribers list and choose **Properties**.

   Click the **Advanced** tab. On this tab, click **Stop Now** to stop subscriptions.

   The subscriptions are affected when you click the **Stop Now** button. Subsequently clicking **Cancel** on the properties window does *not* cancel your stop synchronization action.

SQL Remote reference

This section provides reference material for SQL Remote.

SQL Remote utilities and options reference

SQL Remote Message Agent utility (dbremote)

Sends and applies SQL Remote messages, and maintains the message tracking system to ensure message delivery.

Syntax

dbremote [*options*] [*directory*]

Option	Description

@data	Reads in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used. 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. See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].
	
	The environment variable can 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 SQL Remote process that starts with a cache size of 4 MB, receives messages only, and connects to a database named **field** on a database server named **myserver**. The SET statement should be entered all on one line:
	
	`SET envvar=-m 4096 -r`
	` -c`
	`"Server=myserver;DBN=field;UID=sa;PWD=sysadmin"`
	`dbremote @envvar`
	
	The configuration file contains line breaks, and can contain any set of options. For example, the following command file holds a set of options for a SQL Remote Message Agent that starts with a cache size of 4 MB, sends messages only, and connects to a database named **field** on a database server named **myserver**:
	
	`-m 4096`
	`-s`
	`-c "Server=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	Processes 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	Runs in batch mode. In this mode, the SQL Remote Message Agent processes incoming messages, scans the transaction log once, processes outgoing messages, and then stops.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-c *"keyword=value; ..."*	Specifies connection parameters. If this option is not specified, the environment variable SQLCONNECT is used. For example, the following statement runs dbremote on a database file located at *c:\mydata.db*, connecting with user ID **DBA** and password **sql**: `dbremote -c "UID=DBA;PWD=sql;DBF=c:\mydata.db"` The SQL Remote Message Agent must be run by a user with REMOTE DBA authority or DBA authority. See "Granting REMOTE DBA authority" on page 28. The SQL Remote Message Agent supports the full range of SQL Anywhere connection parameters. See "Connection parameters" [*SQL Anywhere Server - Database Administration*].
-dl	Displays messages in the SQL Remote Message Agent window or at the command prompt and in the log file if specified.
-ek *key*	Specifies that you want to be prompted at the command prompt for the encryption key for strongly encrypted databases. 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 fails if you do not specify a key for a strongly encrypted database.
-ep	Specifies that you want to be prompted for the encryption key. This option causes a window 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 fails if you do not specify a key for a strongly encrypted database.
-g *n*	Instructs the SQL Remote Message Agent to group transactions containing fewer 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.

Option	Description
-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 SQL Remote Message Agent for building messages and caching incoming messages. The allowed size can be specified as *n* (in bytes), *n*K, or *n*M. 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 SQL Remote Message Agent until they are applied. This caching of messages prevents rereading 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.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-ml directory</td><td>Specifies the location of offline transaction log mirror files. This option makes it possible for dbremote to delete old transaction log mirror files when either of the following two circumstances occur:

● the offline transaction log mirror is located in a different directory from the transaction log mirror

● dbremote is run on a different computer from the remote database server

In a typical setup, the active transaction log mirror and renamed transaction log mirrors are located in the same directory, and dbremote is run on the same computer as the remote database, so this option is not required and old transaction log mirror files are automatically deleted. Transaction logs in this directory are only affected if the delete_old_logs database option is set to a value other than Off.</td></tr>
<tr><td>-o file</td><td>Prints messages to an output log file. The default is to print output to the screen.</td></tr>
<tr><td>-os size</td><td>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 an output log 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.dbs, where xx are sequential characters ranging from AA to ZZ, and yymmdd represents the current year, month, and date.

If the SQL Remote Message Agent is running in continuous mode for a long time, this option allows you to manually delete old output log files and free up disk space.</td></tr>
<tr><td>-ot file</td><td>Truncates the output log file and then appends output messages to it. The default is to send output to the screen.</td></tr>
<tr><td>-p</td><td>Does not purge messages.</td></tr>
<tr><td>-q</td><td>Starts the SQL Remote Message Agent with a minimized window. This option applies for Windows operating systems only.</td></tr>
<tr><td>-qc</td><td>Closes SQL Remote window on completion.</td></tr>
</table>

Option	Description
-r	Receives messages. If neither -r or -s is specified, the SQL Remote Message Agent executes both phases. Otherwise, only the indicated phases are executed. The SQL Remote Message Agent runs in continuous mode if it is run with the -r option. To have the SQL Remote Message Agent shut down after receiving messages, use the -b option in addition to -r.
-rd *minutes*	Specifies the polling frequency for incoming messages. By default, the SQL Remote 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, the following command polls every thirty seconds. `dbremote -rd 30s` The -rd option is often used in conjunction with the -rp option that sets the number of polls for which the SQL Remote Message Agent waits before requesting that a missing message be resent. See "Improving performance when receiving messages" on page 85.
-ro *filename*	Logs remote output to file. 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. See "Collect errors from the remote database" on page 118.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-rp *number*	Specifies the number of receive polls before a message is assumed lost. When running in continuous mode, the SQL Remote Message Agent polls at certain intervals for messages. After polling a set number of times (by default, one), if a message is missing, the SQL Remote Message Agent assumes it has been lost and requests that the message be resent. On slow message systems, this behavior can result in many unnecessary resend requests. You can set the number of polls before a resend request is issued using this option to minimize the number of resend requests. For more information on configuring this option, see "Improving performance when receiving messages" on page 85. The -rp option is often used in conjunction with the -rd option that sets the polling frequency for incoming messages.
-rt *filename*	Truncates the output log file on startup, and then appends the log output from the remote database to the file. This option is for use at consolidated sites. It is identical to the -ro option except that the file is truncated on startup.
-ru *time*	Specifies the waiting period to re-scan log on receipt of a resend. Control the **resend urgency**. This is the time between detection of a resend request and when the SQL Remote Message Agent starts fulfilling the request. Use this option to help the SQL Remote Message Agent collect resend requests from multiple users before rescanning the log. The time unit can be s (seconds), m (minutes), h (hours), or d (days).
-s	Sends messages. If neither -r or -s is specified, the SQL Remote Message Agent executes both phases. Otherwise, only the indicated phases are executed.
-sd *time*	Controls the delay between polls of the database transaction log. The -sd option is only used when running in continuous mode. Controls the **send delay** that is the time to wait between polls for more transaction log data to send.

Option	Description
-t	Replicates all triggers. 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. “Using the CURRENT REMOTE USER special value” on page 44.
-u	Processes only transactions that exist in offline transaction logs. This option prevents the SQL Remote Message Agent from processing transactions since the latest backup. Using this option, outgoing transactions and confirmation of incoming transactions are not sent until they exist in offline transaction logs. This means that only transactions from renamed logs are processed.
-ud	Runs the SQL Remote Message Agent as a daemon on Unix platforms. If you run the SQL Remote Message Agent as a daemon, you must also supply the -o or -ot option to log output information. If you run the SQL Remote 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 SQL Remote Message Agent does not prompt the user for these options when running as a daemon. For information on message link parameters, see “Set remote message type control parameters” on page 102. When you start the SQL Remote Message Agent as a daemon, its permissions are controlled by the current user's umask setting. It is recommended that you set the umask value before starting the SQL Remote Message Agent to ensure that the it has the appropriate permissions.
-ui	When this option is used, SQL Remote Message Agent tries to start with X Windows. If this fails, it starts in shell mode. When -ui is specified, SQL Remote Message Agent attempts to find a usable display. If it cannot find one, for example because the X window server isn't running, then SQL Remote Message Agent starts in shell mode.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-ux	Opens the SQL Remote messages window on Solaris and Linux. 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 window server is not running, dbremote fails to start. On Windows, the SQL Remote messages window appears automatically.
-v	Displays verbose output. This option displays the SQL statements contained in the messages to the messages window and, if the -o or -ot option is used, to a log file.
-w *n*	Specifies the number of database worker threads to apply incoming messages. This option is not supported on Windows Mobile. 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 maximum number of database worker threads is 50. 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), can improve throughput of incoming messages. Multiple processors in the computer running the SQL Remote Message Agent could also improve throughput of incoming messages. The -w option does not improve performance significantly on hardware that cannot perform many concurrent operations. Incoming messages from a single remote database are never applied on multiple threads. Messages from a single remote database are always applied serially in the correct order.
-x [*size*]	Renames and restarts 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. 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), *n*K, or *n*M. The default is 0.

Option	Description
Directory	Specifies the directory in which old transaction logs are held. The optional *directory* parameter specifies a directory in which old transaction logs are held, so that the SQL Remote Message Agent has access to events from before the current log was started.
-y	Overwrites the command file without confirmation. Without specifying this option, you are prompted to confirm the replacement of an existing command file.

Remarks

The SQL Remote Message Agent sends and applies messages for SQL Remote replication, and maintains the message tracking system to ensure message delivery.

You can also run the SQL Remote 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 SQL Remote Message Agent command must have either REMOTE DBA or DBA authority. See “Granting REMOTE DBA authority” on page 28.

The SQL Remote Message Agent uses several database connections.

- **Message system control parameters** SQL Remote uses several registry settings to control aspects of message link behavior.

  On Windows, the message link control parameters are stored in the registry, at the following location:

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

  For a listing of registry settings, see the section for each message system in “SQL Remote message systems” on page 99.

Extraction utility (dbxtract)

Extracts a remote database from a consolidated SQL Anywhere database.

Syntax

dbxtract [*options*] [*directory*] *subscriber*

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
@data	Reads in options from a configuration file. See "@data dbeng12/ dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]. 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. 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. See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].
-ac *"keyword=value; ..."*	Connects to the database specified in the connection string to do the reload. 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 (entered all on one line) loads a copy of the data for the field_user subscriber into an existing database file named *c:\field.db*: `dbxtract -c "UID=DBA;PWD=sql;DBF=c:\cons.db" -ac "UID=DBA;PWD=sql;DBF=c:\field.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.
-al *filename*	Specifies the transaction log file name for the new database if using the -an option.

Option	Description

-an *database*	Creates a database file with the same settings as the database being extracted and automatically reload it.
	
	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 (entered all on one line) creates a new database file named *c:\field.db* and copies the schema and data for the field_user subscriber of *c:\cons.db* into it:
	
	`dbxtract -c "UID=DBA;PWD=sql;DBF=c:\cons.db"`
	`-an c:\field.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.

-ap *size* [**k**]	Sets the page size of the new database. This option is ignored unless -an is used. The page size for a database can be (in bytes) 2048, 4096, 8192, 16384, or 32768, with the default being the page size of the original database. Use **k** to specify units of kilobytes (for example, -ap 4k). If there are already databases running on the database server, the server's page size (set with the -gp option) must be large enough to handle the new page size. See "-gp dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].
-b	Does not start subscriptions. If this option is specified, 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. See "START SUBSCRIPTION statement [SQL Remote]" [*SQL Anywhere Server - SQL Reference*].

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-c *"keyword=value; ..."*	Specifies database connection parameters, in a string: 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 (entered all on one line) extracts a database for remote user ID joe_remote from the sample database running on the sample_server database server, connecting as user ID DBA with password sql. The data is unloaded into the *c:\extract* directory. `dbxtract -c "Server=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.
-d	Extracts data only. If this option is specified, the schema definition is not unloaded and publications and subscriptions are not created at the remote database. This option is used when a remote database already exists with the proper schema, and only needs to be filled with data.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-ea alg</td><td>
Specifies the encryption algorithm for the new database. 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.
For greater security, specify AES or AES256 for 128-bit or 256-bit strong encryption, respectively. Specify AES_FIPS or AES256_FIPS for 128-bit or 256-bit FIPS-approved encryption, respectively. For strong encryption, you must also specify the -ek or -ep option. For more information about strong encryption, see “Strong encryption” [SQL Anywhere Server - Database Administration].
To create a database that is not encrypted, specify -ea none, or do not include the -ea option (and do not specify -e, -et, -ep, or -et).
If you do not specify the -ea option, the default behavior is as follows:

-ea none, if -ek, -ep, or -et is not specified
-ea AES, if -ek or -ep is specified (with or without -et)
-ea simple, if -et is used without -ek or -ep

Algorithm names are case insensitive.
Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.
See “Separately licensed components” [SQL Anywhere 12 - Introduction].
</td></tr>
<tr><td>-ek key</td><td>
Specifies the encryption key for the new database. 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
For strongly encrypted databases, be sure to store a copy of the key in a safe location. If you lose the encryption key, there is no way to access the data, even with the assistance of technical support. The database must be discarded and you must create a new database.
</td></tr>
</table>

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-ep</td><td>Prompts for the encryption key for the new database. This option specifies that you want to create a strongly encrypted database by typing the encryption key in a window. 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. See “Strong encryption” [SQL Anywhere Server - Database Administration].</td></tr>
<tr><td>-er</td><td>Removes encryption from encrypted tables during an unload procedure.

When extracting from a database that has table encryption enabled, you must specify either -er or -et to indicate whether the new database has table encryption enabled, otherwise you get an error when attempting to load the data into the new database.

The following command extracts a database (cons.db) that has encrypted tables, into a new database (field.db) that does not have table encryption enabled, removing encryption from any encrypted tables:

<code>dbxtract -an c:\field.db -er -c "UID=DBA;PWD=sql;DBF=c:\cons.db;DBKEY=29bN8cj1z field_user"</code></td></tr>
<tr><td>-et</td><td>Enables database table encryption in the new database (-an or -ar must also be specified). If you specify the -et option without the -ea option, the AES algorithm is used. If you specify the -et option, you must also specify -ep or -ek. You can change the table encryption settings for the new database to be different from those of the database you are unloading.

When rebuilding a database that has table encryption enabled, you must specify either -er or -et to indicate whether the new database has table encryption enabled, otherwise you get an error when attempting to load the data into the new database.

The following example unloads a database (cons.db) that has tables encrypted with the simple encryption algorithm, into a new database (field.db) that has table encryption enabled, and uses AES_FIPS encryption with the key 34jh:

<code>dbxtract -an c:\field.db -et -ea AES_FIPS -ek 34jh -c "UID=DBA;PWD=sql;DBF=c:\cons.db field_user"</code></td></tr>
</table>

Option	Description
-f	Extracts fully qualified publications. Usually 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. 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.
-g	● **Materialized views** By default, materialized views defined as MANUAL REFRESH are not initialized after a reload. If you want these materialized views to be initialized as part of the reload process, specify the -g option. Specifying -g causes the database server to execute the sa_refresh_materialized_views system procedure. See "sa_refresh_materialized_views system procedure" [*SQL Anywhere Server - SQL Reference*]. When deciding whether to use the -g option, consider that initializing all materialized views may cause the reload process to take significantly longer to complete. On the other hand, not using the -g option means that the first query that attempts to use an uninitialized materialized view must wait while the database server initializes the view, which may cause an unexpected delay. If you do not use the -g option, you can also manually initialize materialized views after the reload completes. See "Initialize materialized views" [*SQL Anywhere Server - SQL Usage*]. ● **Text indexes** By default, text indexes defined as MANUAL REFRESH are not initialized after a reload. If you want the text indexes initialized as part of the reload process, specify the -g option. Specifying -g causes the database server to execute the sa_refresh_text_indexes system procedure. See "sa_refresh_text_indexes system procedure" [*SQL Anywhere Server - SQL Reference*].
-ii	Performs an internal unload and internal reload. 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 take the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the database server.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-ix	Performs an internal unload and external reload. 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 take the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the database server.
-l *level*	Performs all extraction operations at specified isolation level. The default setting is an isolation level of 0. If you are extracting a database from an active database server, you should run it at isolation level 3 to ensure that data in the extracted database is consistent with data on the database server. Increasing the isolation level may result in large numbers of locks being used by the Extraction utility (dbxtract), and may restrict database use by other users. See "Extraction utility (dbxtract)" on page 140.
-n	Extracts the schema definition only. With this definition, none of the data is unloaded. The reload file contains SQL statements to build the database schema 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. `dbxtract -c "UID=DBA;PWD=sql;DBF=c:\remote\cons\cons.db" -n "c:\remote\reload.sql" UserName`
-nl	Extracts the structure (the same behavior as the -n option), but the resulting *reload.sql* file also includes LOAD TABLE or INPUT statements for each table. No user data is extracted when this option is used. When you specify -nl, you must also include a data directory so that the LOAD/INPUT statements can be generated, even though no files are written to the directory. This option allows you to generate a reload script without unloading data. You can extract the data by specifying -d. If a database contains a table whose data should not be unloaded, you can avoid unloading the data for that table by using `dbxtract -d -e` *`table-name`*.
-o *file*	Outputs messages to an output log file.
-p *character*	Specifies an escape character. The default escape character (\) can be replaced by another character using this option.

Option	Description
-q	Operates quietly: does not display messages or show windows. When this option is specified, -y must also be specified or the operation fails. This option is available only for the command line utility.
-r *file*	Specifies the name of the generated reload Interactive SQL command file. 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.
-u	Does not order data during the unload. By default, the data in each table is ordered by primary key. Unloads are faster with the -u option, but loading the data into the remote database is slower.
-v	Displays verbose messages. The name of the table being unloaded, the number of rows unloaded, and the SELECT statement used.
-xf	Excludes foreign keys. You can use this option if the remote database contains a subset of the consolidated database schema, and some foreign key references are not present in the remote database.
-xh	Excludes procedure hooks.
-xi	Performs an external unload and internal reload. 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 on the client. External operations take the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the database server.
-xp	Does not extract stored procedures from the database.
-xt	Does not extract triggers from the database.
-xv	Does not extract views from the database.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Description
-xx	Performs an external unload and an external load. 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 OUT-PUT and INPUT. External operations take the path of the data files relative to the current working directory of dbxtract, while internal statements take the path relative to the database server.
directory	Specifies the directory the files are written to. This is not needed if you specify -an or -ac .
subscriber	Specifies the subscriber for whom the database is being extracted.

Remarks

By default, the Extraction utility (dbxtract) runs at isolation level zero. If you are extracting a database from an active database server, you should run it at isolation level 3 to ensure that data in the extracted database is consistent with data on the database server. Running at isolation level 3 may hamper others' turnaround time on the database server because of the large number of locks required. It is recommended that you run the Extraction utility (dbxtract) when the database server is not busy, or run it against a copy of the database.

The Extraction utility (dbxtract) 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.

When using the Extraction utility (dbxtract) or the **Extract Database Wizard** with a version 10.0.0 or later database, the version of dbxtract used must match the version of the database server used to access the database. If an older version of dbxtract is used with a newer database server, or vice versa, an error is reported.

The Extraction utility (dbxtract) and **Extract Database Wizard** do 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 (dbxtract), and so these objects are preserved.

See also

- “Extracting remote databases” on page 72
- “Extracting databases” [*SQL Anywhere Server - SQL Usage*]

SQL Remote options

Replication options are database options included to provide control over replication behavior.

Syntax

SET [**TEMPORARY**] **OPTION**
[*userid*. | **PUBLIC.**]*option-name* **=** [*option-value*]

Parameter	Description
option-name	The name of the option being changed.
option-value	A string containing the setting for the option.

Remarks

These options are used by the SQL Remote Message Agent, and should be set for the user ID specified in the SQL Remote Message Agent command. They can also be set for general public use.

Option	Values	Default
“blob_threshold option [SQL Remote]” [*SQL Anywhere Server - Database Administration*]	Integer (in bytes)	256
“compression option [SQL Remote]” [*SQL Anywhere Server - Database Administration*]	Integer, from -1 to 9	6
“delete_old_logs option [SQL Remote]” [*SQL Anywhere Server - Database Administration*]	On, Off, Delay, *n* days	Off
“external_remote_options [SQL Remote]” [*SQL Anywhere Server - Database Administration*]	On, Off	Off
“qualify_owners option [SQL Remote]” [*SQL Anywhere Server - Database Administration*]	On, Off	On

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Option	Values	Default
"quote_all_identifiers option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	On, Off	Off
"replication_error option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	Stored procedure name	(no procedure)
"replication_error_piece option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	Stored procedure name	(no procedure)
"save_remote_passwords option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	On, Off	On
"sr_date_format option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	*date-string*	*YYYY/MM/DD*
"sr_time_format option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	*time-string*	*HH*:*NN*:*SS.SSSSSS*
"sr_timestamp_format [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	*timestamp-string*	*YYYY/MM/DD HH*:*NN*:*SS.SSSSSS*
"subscribe_by_remote option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	On, Off	On
"verify_all_columns option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	On, Off	Off
"verify_threshold option [SQL Remote]" [*SQL Anywhere Server - Database Administration*]	Integer (in bytes)	1000

SQL Remote system procedures

The following stored procedure names and arguments provide the interface for customizing replication at SQL Remote databases.

Notes

Unless otherwise stated, the following conditions 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 SQL Remote 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 SQL Remote 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 SQL Remote Message Agent.

sp_hook_dbremote_begin system procedure

Use this system 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.

Remarks

If a procedure of this name exists, it is called when the SQL Remote Message Agent starts.

sp_hook_dbremote_end system procedure

Use this system procedure to add custom actions just before the SQL Remote Message Agent exits.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

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.

Remarks

If a procedure of this name exists, it is called as the last event before the SQL Remote Message Agent shuts down.

sp_hook_dbremote_shutdown system procedure

Use this system procedure to initiate a SQL Remote 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 SQL Remote Message Agent is shut down.

Remarks

If a procedure of this name exists, it is called when the SQL Remote Message Agent is neither sending nor receiving messages, and permits a hook-initiated shutdown of the SQL Remote Message Agent.

sp_hook_dbremote_receive_begin system procedure

Use this system procedure to perform actions before the start of the receive phase of replication.

Rows in #hook_dict

None

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

sp_hook_dbremote_receive_end system procedure

Use this system procedure to perform actions after the end of the receive phase of replication.

Rows in #hook_dict

None

sp_hook_dbremote_send_begin

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

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

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

Use this stored procedure to perform actions when the SQL Remote Message Agent has determined that one or more messages is missing from a remote user.

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

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

Use this stored procedure to perform actions just before the SQL Remote 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

Use this stored procedure to perform actions just after the SQL Remote 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.

SQL Remote system objects

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*]
- "SYSREMOTEUSER system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSSUBSCRIPTION system view" [*SQL Anywhere Server - SQL Reference*]

SQL Remote SQL statements

Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

Following are the SQL statements used for executing SQL Remote commands:

- "ALTER PUBLICATION statement [MobiLink] [SQL Remote]" [*SQL Anywhere Server - SQL Reference*]
- "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*]

Index

Symbols

#hook_dict table
SQL Remote Message Agent utility (dbremote), 152
SQL Remote unique primary keys, 69
-a option
SQL Remote Message Agent utility (dbremote), 131
-ac option
SQL Remote extraction utility (dbxtract), 140
-al option
SQL Remote extraction utility (dbxtract), 140
-an option
SQL Remote extraction utility (dbxtract), 140
-ap option
SQL Remote extraction utility (dbxtract), 140
-b option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-c option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-d option
SQL Remote extraction utility (dbxtract), 140
-dl option
SQL Remote Message Agent utility (dbremote), 131
-ea option
SQL Remote extraction utility (dbxtract), 140
-ek option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-ep option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-er option
SQL Remote extraction utility (dbxtract), 140
-et option
SQL Remote extraction utility (dbxtract), 140
-f option
SQL Remote extraction utility (dbxtract), 140
-g option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-ii option
SQL Remote extraction utility (dbxtract), 140
-ix option
SQL Remote extraction utility (dbxtract), 140
-l option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 80, 131
-m option
SQL Remote Message Agent utility (dbremote), 131
-ml option
SQL Remote Message Agent utility (dbremote), 131
-n option
SQL Remote extraction utility (dbxtract), 140
-nl option
SQL Remote extraction utility (dbxtract), 140
-o option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-os option
SQL Remote Message Agent utility (dbremote), 131
-ot option
SQL Remote Message Agent utility (dbremote), 131
-p option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-q option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-qc option
SQL Remote Message Agent utility (dbremote), 131
-r option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131

-rd option
SQL Remote Message Agent utility (dbremote), 131
-ro option
SQL Remote Message Agent utility (dbremote), 131
-rp option
SQL Remote Message Agent utility (dbremote), 131
-rt option
SQL Remote Message Agent utility (dbremote), 131
-ru option
SQL Remote Message Agent utility (dbremote), 131
-s option
SQL Remote Message Agent utility (dbremote), 131
-sd option
SQL Remote Message Agent utility (dbremote), 131
-t option
SQL Remote Message Agent utility (dbremote), 131
-u option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-ud option
SQL Remote Message Agent utility (dbremote), 131
-ui option
SQL Remote Message Agent utility (dbremote), 131
-ux option
SQL Remote Message Agent utility (dbremote), 131
-v option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131
-w option
SQL Remote Message Agent utility (dbremote), 131
-x option
SQL Remote Message Agent utility (dbremote), 131
-xf option
SQL Remote extraction utility (dbxtract), 140
-xh option
SQL Remote extraction utility (dbxtract), 140
-xi option
SQL Remote extraction utility (dbxtract), 140
-xp option
SQL Remote extraction utility (dbxtract), 140
-xt option
SQL Remote extraction utility (dbxtract), 140
-xv option
SQL Remote extraction utility (dbxtract), 140
-xx option
SQL Remote extraction utility (dbxtract), 140
-y option
SQL Remote extraction utility (dbxtract), 140
@data option
SQL Remote extraction utility (dbxtract), 140
SQL Remote Message Agent utility (dbremote), 131

A

ActiveSync
SQL Remote synchronization for Windows Mobile, 104
addresses
SQL Remote FILE sharing, 104
SQL Remote FTP, 105
administering
SQL Remote, 71
administering SQL Remote
about, 71
ALTER REMOTE MESSAGE TYPE statement
using, 101
articles
INSERT statements, 49
SQL Remote creation, 10
authorities
SQL Remote revoking REMOTE DBA, 29

B

backups
SQL Remote transaction log management, 110
batch mode
SQL Remote Message Agent utility (dbremote), 82
BEFORE triggers
SQL Remote ignoring errors, 118
BLOBs

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

SQL Remote, 37
bugs
providing feedback, viii

C

cache
SQL Remote received messages, 86
SQL Remote sent messages, 92
ccMail
SQL Remote, 99
command prompts
conventions, vii
curly braces, vii
environment variables, vii
parentheses, vii
quotes, vii
semicolons, vii
command shells
conventions, vii
curly braces, vii
environment variables, vii
parentheses, vii
quotes, vii
confirm_received column
SQL Remote, 96
conflict resolution
SQL Remote approaches, 41, 47
SQL Remote triggers, 41
conflicts
SQL Remote, 39
SQL Remote management, 41
connections
SQL Remote Message Agent, 131
CONSOLIDATE permissions
SQL Remote, 25
SQL Remote granting, 26
consolidated databases
setting, 26
SQL Remote, 6
contacts SQL Remote example
about, 56
continuous mode
SQL Remote Message Agent utility (dbremote), 80
conventions
command prompts, vii
command shells, vii
documentation, v
file names in documentation, vi
operating systems, v
Unix , v
Windows, v
Windows CE, v
Windows Mobile, v
create article wizard
SQL Remote adding articles in , 17
create publication wizard
SQL Remote, 10
create SQL Remote message type wizard
adding message types in Sybase Central, 100
create SQL Remote subscription wizard
using, 30
CREATE SUBSCRIPTION statement
SQL Remote, 30
CURRENT REMOTE USER special value
SQL Remote using, 44

D

daemon
SQL Remote Message Agent utility (dbremote), 131
data exchange
SQL Remote, 1
data movement technologies
SQL Remote replication, 1
data recovery
SQL Remote, 110
data types
SQL Remote replicating, 37
database extraction utility (dbxtract)
SQL Remote syntax, 140
databases
setting a consolidated database, 26
dbo user
SQL Remote system objects, 140
dbremote utility
options, 131
SQL Remote #hook_dict table, 152
SQL Remote about, 78
SQL Remote daemon, 131
SQL Remote introduction, 4
SQL Remote security, 122
starting on Mac OS X, 83
syntax, 131
dbunload utility

SQL Remote, 123
dbxtract utility
SQL Remote about, 126
SQL Remote options, 140
SQL Remote sp_hook_dbxtract_begin procedure, 69
SQL Remote syntax, 140
DCX
about, v
debug control parameter
SQL Remote FILE message type, 104
SQL Remote FTP message type, 105
SQL Remote SMTP message type, 107
Deleting Corrupt Message error
SQL Remote, 98
deploying
SQL Remote databases, 72
design overview
SQL Remote, 31
developer centers
finding out more and requesting technical support, ix
developer community
newsgroups, viii
directory control parameter
SQL Remote FILE message type, 104
directory option
SQL Remote (dbremote), 131
SQL Remote extraction utility (dbxtract) syntax, 140
DLL
SQL Remote replicating, 36
DocCommentXchange (DCX)
about, v
document_in_progress table
SQL Remote usage, 38
documentation
conventions, v
SQL Anywhere, v
dropping
SQL Remote message types, 102
SQL Remote publications, 17

E

encode_dll control parameter
SQL Remote FILE message type, 104
SQL Remote FTP message type, 105
encoding
SQL Remote custom, 98
encoding and compressing messages
SQL Remote, 98
encoding scheme
SQL Remote, 98
encryption
SQL Remote, 122
environment variables
command prompts, vii
command shells, vii
errors
SQL Remote default handling, 117
SQL Remote reporting by SQL Remote Message Agent (dbremote), 117
event hooks
sp_hook_dbremote_message_sent stored procedure, 154
SQL Remote about, 151
SQL Remote sp_hook_dbremote_begin stored procedure, 152
SQL Remote sp_hook_dbremote_end , 152
SQL Remote sp_hook_dbremote_message_apply_begin stored procedure, 155
SQL Remote sp_hook_dbremote_message_apply_end stored procedure, 155
SQL Remote sp_hook_dbremote_message_missing stored procedure, 154
SQL Remote sp_hook_dbremote_receive_begin stored procedure, 153
SQL Remote sp_hook_dbremote_receive_end stored procedure, 154
SQL Remote sp_hook_dbremote_send_begin stored procedure, 154
SQL Remote sp_hook_dbremote_send_end stored procedure, 154
SQL Remote sp_hook_dbremote_shutdown stored procedure, 153
extracting
SQL Remote deploying databases, 72
SQL Remote reload files, 73
extraction utility (dbxtract)
SQL Remote options, 140
SQL Remote synchronizing databases, 126
SQL Remote syntax, 140

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

F

feedback
documentation, viii
providing, viii
reporting an error, viii
requesting an update, viii
FILE message type
SQL Remote, 103
SQL Remote control parameters, 104
SQL Remote using, 99
finding out more and requesting technical assistance
technical support, viii
frequency
SQL Remote about, 81
FTP message system
about, 105
FTP message type
SQL Remote, 105
SQL Remote control parameters, 105
SQL Remote troubleshooting, 106
SQL Remote using, 99

G

getting help
technical support, viii
global autoincrement
SQL Remote, 50
global_database_id option
SQL Remote, 69
GRANT PUBLISH statement
SQL Remote about, 20
GRANT REMOTE DBA statement
SQL Remote using, 28
guaranteed message delivery system
SQL Remote , 93

H

handling lost or corrupt messages
SQL Remote, 97
help
technical support, viii
hooks
SQL Remote about, 151
host
SQL Remote FTP message type, 105

I

iAnywhere developer community
newsgroups, viii
install-dir
documentation usage, vi
invalid_extensions parameter
SQL Remote FILE message type, 104
SQL Remote FTP message type, 105

L

log_received column
SQL Remote, 96
log_sent column
SQL Remote, 95
Lotus Notes
SQL Remote supported message types, 99

M

Mac OS X
running dbremote, 83
many-to-many relationships
SQL Remote publication design, 61
media failures
SQL Remote, 110
message
SQL Remote administration, 71
message agent
SQL Remote Message Agent, v
Message Agent (dbremote)
syntax, 131
message systems
about, 99
message type control parameters
SQL Remote, 102
message types
SQL Remote , 99
SQL Remote dropping, 102
SQL Remote FILE sharing, 104
SQL Remote FTP, 105
SQL Remote SMTP, 107
SQL Remote using, 99
messages
SQL Remote caching, 86, 92
SQL Remote synchronizing databases, 128
multi-level hierarchies
SQL Remote passthrough mode limitation, 124
SQL Remote permissions, 19

multi-tier hierarchy
SQL Remote worker threads, 89
multi-tiered installations
SQL Remote permissions, 19
mutli-level hierarchies
SQL Remote database extraction, 76

N

newsgroups
technical support, viii
Notes
(*see also* Lotus Notes)
SQL Remote, 99

O

online books
PDF, v
operating systems
Unix, v
Windows, v
Windows CE, v
Windows Mobile, v
options
SQL Remote, 150
output_log_send_limit remote option
SQL Remote troubleshooting, 118
output_log_send_now remote option
SQL Remote troubleshooting, 118
output_log_send_on_error remote option
SQL Remote troubleshooting, 118

P

partitioning
SQL Remote, 10
passthrough mode
SQL Remote, 124
PASSTHROUGH statement
SQL Remote, 124
password control parameter
SQL Remote FTP message type, 105
patience
SQL Remote, 84
PDF
documentation, v
performance
SQL Remote Message Agent utility (dbremote), 85
SQL Remote publications, 10
permissions
SQL Remote granting CONSOLIDATE, 26
SQL Remote management, 25
SQL Remote multi-tier installations, 19
policy example
SQL Remote publications, 61
pop3_host control parameter
SQL Remote SMTP message type, 107
pop3_password control parameter
SQL Remote SMTP message type, 107
pop3_userid control parameter
SQL Remote SMTP message type, 107
port control parameter
SQL Remote FTP message type, 105
primary key pools
SQL Remote, 51
primary keys
SQL Remote, 47
SQL Remote primary key pools, 51
SQL Remote unique values, 50
publications
SQL Remote alteration, 16
SQL Remote creation, 10
SQL Remote design, 10
SQL Remote dropping, 17
SQL Remote many-to-many relationships, 61
SQL Remote replication, 30
PUBLISH permission
SQL Remote, 25
publishing
SQL Remote, 10

R

reconnect_pause parameter
SQL Remote FTP message type, 105
reconnect_retries parameter
SQL Remote FTP message type, 105
recovery
SQL Remote, 110
referential integrity
SQL Remote, 47
reload files
SQL Remote database extraction, 73
reload.sql
SQL Remote, 73
remote message types
SQL Remote altering, 101

SQL Remote creating, 100
REMOTE permission
SQL Remote, 25
replication
(*see also* SQL Remote)
about SQL Remote, 1
publication design, 10
SQL Remote backup procedures, 110
SQL Remote backups, 110
SQL Remote BLOBs, 37
SQL Remote conflicts, 39
SQL Remote data definition statements, 36
SQL Remote data recovery, 110
SQL Remote data types, 37
SQL Remote dbremote, 131
SQL Remote files, 37
SQL Remote Message Agent, 131
SQL Remote passthrough mode, 124
SQL Remote primary key errors, 47
SQL Remote primary keys, 50
SQL Remote procedures, 35
SQL Remote publications, 30
SQL Remote referential integrity errors , 47
SQL Remote subscriptions, 30
SQL Remote transaction log management, 110
SQL Remote triggers, 35
SQL Remote triggers designing, 36
SQL statements, 124
replication conflicts
SQL Remote management, 41, 47
replication errors
SQL Remote, 39
replication errors and conflicts
SQL Remote, 39
replication_error option
SQL Remote error handling procedures, 118
tracking SQL errors, 118
reporting errors
SQL Remote Message Agent utility (dbremote), 117
rereceive_count column
SQL Remote, 97
resend requests
SQL Remote, 87
resend_count column
SQL Remote, 97
RESOLVE UPDATE
example, 45
RESOLVE UPDATE trigger
custom conflict resolution, 44
REVOKE PUBLISH statement
SQL Remote about, 20
REVOKE REMOTE DBA statement
SQL Remote using, 29
revoking consolidate permissions
SQL Remote, 27
revoking REMOTE DBA authority
SQL Remote, 29
revoking remote permissions
SQL Remote, 25
root control parameter
SQL Remote FTP message type, 105

S

samples
SQL Remote policy example, 61
samples-dir
documentation usage, vi
selecting a send frequency
SQL Remote about, 81
SEND AT clause
SQL Remote frequency setting, 81
SEND EVERY clause
SQL Remote frequency setting, 81
send frequency
SQL Remote Message Agent utility (dbremote), 80, 82
SQL Remote selection, 81
services
SQL Remote Message Agent utility (dbremote), 81
SMTP message type
SQL Remote, 106
SQL Remote control parameter, 107
SQL Remote using, 99
SMTP/POP
SQL Remote addresses, 108
smtp_authenticate control parameter
SQL Remote SMTP message type, 107
smtp_host control parameter
SQL Remote SMTP message type, 107
smtp_password control parameter
SQL Remote SMTP message type, 107
smtp_userid control parameter
SQL Remote SMTP message type, 107
sp_hook_dbremote_begin stored procedure

SQL Remote syntax, 152
sp_hook_dbremote_end stored procedure
SQL Remote syntax, 152
sp_hook_dbremote_message_apply_begin stored procedure
SQL Remote syntax, 155
sp_hook_dbremote_message_apply_end stored procedure
SQL Remote syntax, 155
sp_hook_dbremote_message_missing stored procedure
syntax, 154
sp_hook_dbremote_message_sent stored procedure
SQL Remote syntax, 154
sp_hook_dbremote_receive_begin stored procedure
SQL Remote syntax, 153
sp_hook_dbremote_receive_end stored procedure
SQL Remote syntax, 154
sp_hook_dbremote_send_begin stored procedure
SQL Remote syntax, 154
sp_hook_dbremote_send_end stored procedure
SQL Remote syntax, 154
sp_hook_dbremote_shutdown stored procedure
SQL Remote syntax, 153
sp_hook_dbxtract_begin procedure
SQL Remote, 69
specifying a consolidated database
about, 26
SQL Anywhere
documentation, v
SQL Anywhere Developer Centers
finding out more and requesting technical support, ix
SQL Anywhere Tech Corner
finding out more and requesting technical support, ix
SQL Remote
(*see also* replication)
about, 1
ActiveSync and Windows Mobile, 104
administering, 71
administration overview, 71
backup procedures, 110
backup procedures at remote databases, 110
components, 4
concepts, 1
dbxtract utility, 140
deployment overview, 72
design principles, 31
event hooks, 151
extracting remote databases, 73
guaranteed message delivery system, 93
mobile workforces, 1
publications, 30
receiving message tasks, 84
reload.sql, 74
replicating data types, 37
replicating dates, 38
replicating DDL statements, 36
replicating deletes, 33
replicating inserts, 33
replicating procedures, 35
replicating times, 38
replicating triggers, 35
replicating updates, 33
replication system recovery procedures, 110
resolving date conflicts, 44
running as a service, 81
SQL Anywhere system tables, 155
SQL Remote Message Agent (dbremote) performance, 85
SQL Remote Message Agent introduction, 4
SQL Remote trigger replication, 36
SQL statements, 156
subscribers, 1
subscriptions, 30
supported message systems, 99
system objects, 155
transaction log management, 110
unloading databases, 123
users removing, 25, 27
utilities and options reference, 131
Windows Mobile and ActiveSync, 104
SQL Remote administration
about, 71
SQL Remote components
about, 4
SQL Remote concepts
about, 1
SQL Remote Message Agent (dbremote)
guaranteed message delivery system, 93
running on Mac OS X, 83
SQL Remote about, 78
SQL Remote administration, 71
SQL Remote backup procedures, 110
SQL Remote batch mode, 82

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0

SQL Remote continuous mode, 80
SQL Remote daemon, 131
SQL Remote introduction, 4
SQL Remote Message Agent utility (dbremote), 131
SQL Remote output , 117
SQL Remote performance, 85
SQL Remote reporting errors, 117
SQL Remote running as a service, 81
SQL Remote security, 122
SQL Remote settings, 80
SQL Remote transaction log management about, 110
SQL Remote tuning throughput, 89
syntax, 131
SQL Remote options
about, 150
SQL statements
SQL Remote list, 156
SQLANY.INI
SQL Remote, 102
SQLREMOTE environment variable
alternative to, 104
setting message control parameters, 102
stable queue
SQL Remote cleaning, 131
statements
SQL Remote, 156
subscriber option
SQL Remote extraction utility (dbxtract), 140
subscription expressions
SQL Remote cost of evaluating, 32
SQL Remote using, 14
subscriptions
SQL Remote creation, 30
SQL Remote replication, 30
support
newsgroups, viii
suppress_dialogs control parameter
SQL Remote SMTP message type, 107
suppress_dialogs parameter
SQL Remote FTP message type, 105
Sybase Central
setting a consolidated database, 26
SyncConsole
starting dbremote, 83
synchronization
SQL Remote, 126
synchronizing
SQL Remote synchronizing databases, 126
synchronizing data over a message system
SQL Remote, 128
SYSREMOTEUSER
confirm_received column, 96
log_received column, 96
log_sent column, 95
rereceive_count column, 97
resend_count column, 97
SQL Remote, 93
system objects
SQL Remote, 155
SQL Remote dbo user, 140
system tables
SQL Remote, 155

T

tech corners
finding out more and requesting technical support, ix
technical support
newsgroups, viii
territory realignment
SQL Remote foreign keys, 58
SQL Remote many-to-many relationships, 65
SQL Remote UPDATES, 33
testing
SQL Remote deployments, 71
tracking SQL errors
SQL Remote, 118
transaction log
SQL Remote guaranteed message delivery, 95
SQL Remote Message Agent, 131
SQL Remote offsets, 94
SQL Remote publications, 84
transaction log mirror
SQL Remote, 110
triggers
SQL Remote, 36
SQL Remote designing, 61
troubleshooting
newsgroups, viii
SQL Remote errors, 117

U

unique column values

SQL Remote, 50
Unix
documentation conventions, v
operating systems, v
SQL Remote supported message types, 99
unlink_delay control parameter
SQL Remote FILE message type, 104
unloading
SQL Remote consolidated databases, 123
UPDATE conflicts
SQL Remote, 41, 47
UPDATE statement
SQL Remote territory realignment, 33
user control parameter
SQL Remote FTP message type, 105
utilities
SQL Remote extraction (dbxtract), 140
SQL Remote Message Agent (dbremote) syntax, 131

V

verify_all_columns option
about, 43

W

Windows
documentation conventions, v
operating systems, v
SQL Remote supported message types, 99
Windows Mobile
documentation conventions, v
operating systems, v
SQL Remote, 104
Windows CE, v

 Copyright © 2010, iAnywhere Solutions, Inc. - SQL Anywhere 12.0.0