CHAPTER 8

SQL Remote Design for Adaptive Server Enterprise

About this chapter

This chapter describes how to design a SQL Remote installation when the consolidated database is at an Adaptive Server Enterprise server.

Similar material for Adaptive Server Anywhere
Many of the principles of publication design are the same for Adaptive Server Anywhere and Adaptive Server Enterprise, but there are differences in commands and capabilities. There is a large overlap between this chapter and the corresponding chapter for Adaptive Server Anywhere users, "SQL Remote Design for Adaptive Server Anywhere" on page 113.

Contents

Topic	Page
Design overview	160
Creating publications	161
Publication design for Adaptive Server Enterprise	166
Partitioning tables that do not contain the subscription column	168
Sharing rows among several subscriptions	175
Managing conflicts	182
Ensuring unique primary keys	192
Creating subscriptions	198

Design overview

Designing a SQL Remote installation includes the following tasks:

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

All administration is at the consolidated database

Like all SQL Remote administrative tasks, design is carried out by a database administrator or system administrator at the consolidated database. The Sybase System Administrator should perform all SQL Remote configuration tasks.

For more information about the Adaptive Server Enterprise environment, see your Adaptive Server Enterprise documentation.

Creating publications

In this section

This section describes how to create simple publications consisting of whole tables, or of column-wise subsets of tables.

Simple publications are also discussed in the chapter "A Tutorial for Adaptive Server Enterprise Users" on page 65.

Creating publications for Adaptive Server Enterprise using Sybase Central

In Sybase Central, you can add a publication to a database from within the SQL Remote folder. The SQL Remote folder is displayed inside a database container.

❖ **To create a publication from Sybase Central:**

1 Click the Publications folder, which is inside the SQL Remote folder.

2 Double-click Add Publication. The Publication Wizard is displayed.

3 Follow the instructions in the Wizard.

For more information, see the Sybase Central online Help.

With Sybase Central, you do not need to know the SQL syntax in order to create publications. The remainder of this section discusses different kinds of publication that can be created. It describes the SQL syntax needed for these publications. However, each of the publications can also be created from Sybase Central.

Creating whole-table articles

The simplest type of article is one that includes all the rows and columns of a database table.

❖ **To create an article that includes all the rows and columns of a table:**

1 Mark the table for replication. You do this by executing the **sp_add_remote_table** procedure:

```
sp_add_remote_table table-name
```

2 Add the table to the publication. You do this by executing the **sp_add_article** procedure:

```
sp_add_article publication-name, table-name
```

Example

- The following commands add the table **SalesRep** to the **SalesRepData** publication:

```
sp_add_remote_table 'SalesRep'
sp_add_article 'SalesRepData', 'SalesRep'
go
```

Creating articles containing some of the columns in a table

To create an article that includes only some of the columns from a table, you need to list the columns that you wish to include, using **sp_add_article_col**. If no columns are listed, the article includes all columns of the table.

❖ **To create an article that includes some of the columns and all the rows of a table:**

1. Mark the table for replication. You do this by executing the **sp_add_remote_table** procedure:

```
sp_add_remote_table table-name
go
```

2. Add the table to the publication. You do this by executing the **sp_add_article** procedure:

```
sp_add_article publication-name, table-name
go
```

The **sp_add_article** procedure adds a table to a publication. By default, all columns of the table are added to the publication. If you wish to add only some of the columns, you must use the **sp_add_article_col** procedure to specify which columns you wish to include.

3. Add individual columns to the publication. You do this by executing the **sp_add_article_col** procedure for each column:

```
sp_add_article_col publication-name,
    table-name,
    column-name
go
```

Example

- The following commands add only the **rep_key** column of the table **SalesRep** to the **SalesRepData** publication:

```
sp_add_remote_table 'SalesRep'
sp_add_article 'SalesRepData',
    'SalesRep'
sp_add_article_col 'SalesRepData',
    'SalesRep',
```

```
   'rep_key'
go
```

Creating articles containing some of the rows in a table

There are two ways of including only some of the rows from a table in an article:

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

Allowed clauses

In SQL Remote for Adaptive Server Enterprise, the following limitations apply to each of these cases:

- **WHERE clause limitations** The only form of WHERE clause supported is the following:

  ```
  WHERE column-name IS NOT NULL.
  ```

- **Subscription column** SQL Remote for Adaptive Server Anywhere supports expressions other than column names. For Adaptive Server Enterprise, the subscription expression must be a column name.

When to use WHERE and SUBSCRIBE BY

You should use a subscription expression when different subscribers to a publication are to receive different rows from a table. The subscription expression is the most powerful method of partitioning tables.

Creating an article using a WHERE clause

The WHERE clause is used to exclude a set of rows from all subscriptions to a publication.

❖ **To create an article using a WHERE clause:**

1. If you have not already done so, mark the table for replication. You do this by executing the **sp_add_remote_table** procedure:

   ```
   sp_add_remote_table table_name
   ```

2. Add the table to the publication. You do this by executing the **sp_add_article** procedure: Specify the column name corresponding to the **WHERE** *column* **IS NOT NULL** clause in the third argument to the procedure:

```
sp_add_article publication_name,
   table_name,
   column_name
```

Do not specify **IS NOT NULL**; it is implicit. Specify the column name only.

3 If you wish to include only a subset of the columns in the table, specify the columns using the **sp_add_article_col** procedure. You must include the column specified in your WHERE clause in the article.

Example

♦ The following set of statements create a publication containing a single article, which includes only those rows of **test_table** for which column **col_1** is not null:

```
sp_create_publication test_pub
sp_add_remote_table test_table
sp_add_article test_pub, test_table, col_1
go
```

Creating an article using a subscription column

The subscription column is used when rows are to be shared among many remote databases.

❖ **To create an article using a subscription column:**

1 If you have not already done so, mark the table for replication. You do this by executing the **sp_add_remote_table** procedure:

sp_add_remote_table *table_name*

2 Add the table to the publication. You do this by executing the **sp_add_article** procedure: Specify the column name you wish to use as a subscription expression in the fourth argument to the procedure:

sp_add_article *publication_name, table_name,* **NULL**, *column_name*

You must include the NULL entry to avoid adding a WHERE clause.

3 If you wish to include only a subset of the columns in the table, specify the columns using the **sp_add_article_col** procedure. You must include the column specified in your subscription expression in the article.

Example

♦ The following set of statements create a publication containing a single article, which supports subscriptions based on the value of column **col_1**:

```
sp_create_publication test_pub
sp_add_remote_table test_table
sp_add_article test_pub,
```

```
    test_table,
    NULL,
    col_1
go
```

Notes on articles

- You can combine a WHERE clause and a subscription expression in an article.
- All columns in the primary key must be included in any article.
- You must not include a subset of columns in an article unless either:
  - The remaining columns have default values or allow NULLs.
  - No inserts are carried out at remote databases. Updates would not cause problems as long as they do not change primary key values.

  If you include a subset of columns in an article in situations other than these, INSERT statements at the consolidated database will fail.

Publication design for Adaptive Server Enterprise

Once you understand how to create simple publications, you must think about proper design of publications. This section describes the issues involved in designing publications, and how to take steps towards sound design.

Design issues overview

Each subscription must be a complete relational database

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

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

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

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

Transaction integrity must be maintained in the absence of locking

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

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

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

Conditions for valid articles

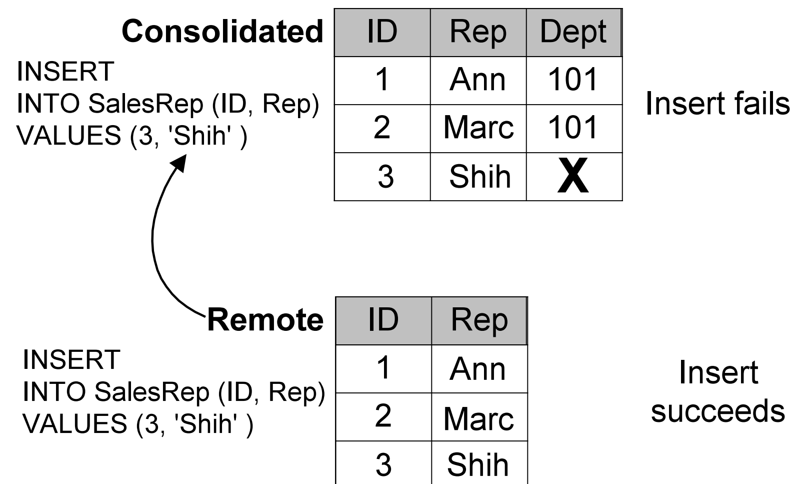
All columns in the primary key must be included in the article.

Supporting INSERTS at remote databases

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

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

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



Conditions on rows

There are two ways of including only some of the rows in a publication:

- **WHERE clause** You can use a WHERE clause to include a subset of rows in an article. All subscribers to the publication containing this article receive the rows that satisfy the WHERE clause.

  In SQL Remote for Adaptive Server Enterprise, the only supported WHERE clause is

  ```
  WHERE column-name IS NOT NULL
  ```

- **Subscription columns** You can use a subscription column to include a different set of rows in different subscriptions to publications containing the article.

For more information on restrictions on rows, see "Creating articles containing some of the rows in a table" on page 163.

Partitioning tables that do not contain the subscription column

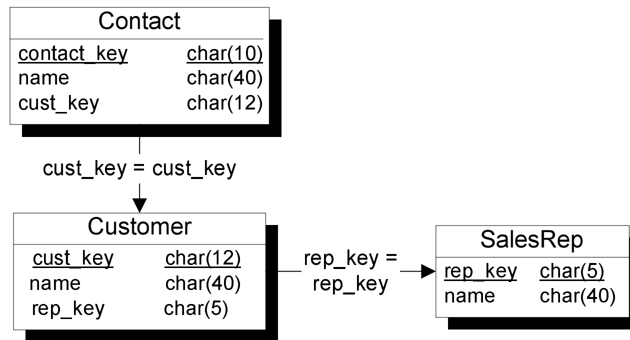
In many cases, the rows of a table need to be partitioned even when the subscription column does not exist in the table. This section describes how to handle this case, using an example.

The Contact example

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

Example

Here is a simple database that illustrates the problem. We call this database the Contact database, because it contains a Contact table in addition to the two tables described earlier in this chapter.



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

The tables in the database

The three tables are described in more detail as follows:

Table	Description

SalesRep	All sales representatives that work for the company. The SalesRep 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 SQL statement creating this table is as follows:
	`CREATE TABLE SalesRep (`
	`rep_key CHAR(12) NOT NULL,`
	`name CHAR(40) NOT NULL,`
	`PRIMARY KEY (rep_key)`
	`)`
	`go`

Customer	All customers that do business with the company. The Customer 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 SalesRep table.
	The SQL statement creating this table is as follows:
	`CREATE TABLE Customer (`
	`cust_key CHAR(12) NOT NULL,`
	`name CHAR(40) NOT NULL,`
	`rep_key CHAR(12) NOT NULL,`
	`FOREIGN KEY ( rep_key )`
	`REFERENCES SalesRep,`
	`PRIMARY KEY (cust_key)`
	`)`
	`go`

<table>
<tr><th>Table</th><th>Description</th></tr>
<tr><td>Contact</td><td>All individual contacts that do business with the company. Each contact belongs to a single customer. The Contact 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 Customer table.

The SQL statement creating this table is:
<pre>CREATE TABLE Contact (
      contact_key CHAR(12) NOT NULL,
      name CHAR(40) NOT NULL,
      cust_key CHAR(12) NOT NULL,
      FOREIGN KEY (cust_key)
      REFERENCES Customer,
      PRIMARY KEY (contact_key)
)
go</pre></td></tr>
</table>

Replication goals

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

- The complete SalesRep table.
- Those customers assigned to them, from the Customer table.
- Those contacts belonging to the relevant customers, from the Contact table.
- Maintenance of proper information when Sales Representative territories are realigned.

Territory realignment in the Contact example

In **territory realignment**, rows are reassigned among subscribers. In the current example, territory realignment involves reassigning customers among the sales representatives. It is carried out by updating the **rep_key** column of the **Customer** table.

The UPDATE is replicated as an INSERT or a or a DELETE to the old and new sales representatives, respectively, so that the customer row is properly transferred to the new sales representative.

No log entries for the Contact table when territories realigned

When a customer is reassigned, the **Contact** table is unaffected. There are no changes to the **Contact** table, and consequently no entries in the transaction log pertaining to the **Contact** table. In the absence of this information, SQL Remote cannot reassign the rows of the **Contact** table along with the **Customer**. This failure would cause referential integrity problems: the **Contact** table at the remote database of the old sales representative contains a **cust_key** value for which there is no longer a **Customer**.

In this section, we describe how to reassign the rows of the **Contact** table.

Partitioning the Customer table in the Contact example

The **Customer** table can be partitioned using the **rep_key** value as a subscription column. A publication that includes the **SalesRep** and **Customer** tables would be as follows:

```
exec sp_add_remote_table 'SalesRep'
exec sp_add_remote_table 'Customer'
go
exec sp_create_publication 'SalesRepData'
go
exec sp_add_article 'SalesRepData', 'SalesRep'
exec sp_add_article SalesRepData,
         Customer, NULL,
         'rep_key'
go
```

Adding a subscription-list column to the Contact table

The **Contact** table must also be partitioned among the sales representatives, but contains no reference to the sales representative **rep_key** value.

Add a subscription-list column

To solve this problem in Adaptive Server Enterprise, you must add a column to the **Contact** table containing a comma-separated list of subscription values to the row. (In the present case, there can only be a single subscription value.) The column can be maintained using triggers, so that applications against the database are unaffected by the presence of the column. We call this column a **subscription-list column**.

When a row in the **Customer** table is inserted, updated or deleted, a trigger updates rows in the **Contact** table. In particular, the trigger updates the subscription-list column. As the **Contact** table is marked for replication, the before and after image of the row is recorded in the log.

> **Log entries are values, not subscribers**
> Although in this case the values entered correspond to subscribers, it is not a list of subscribers that is entered in the log. The server handles only information about publications, and the Message Agent handles all information about subscribers. The values entered in the log are for comparison to the subscription value in each subscription. For example, if rows of a table were divided among sales representatives by state or province, the state or province value would be entered in the transaction log.

A **subscription-list column** is a column added to a table for the sole purpose of holding a comma-separated list of subscribers. In the present case, there can only be a single subscriber to each row of the **Contact** table, and so the subscription-list column holds only a single value.

☞ For a discussion of the case where the subscription-list column can hold many values, see "Sharing rows among several subscriptions" on page 175.

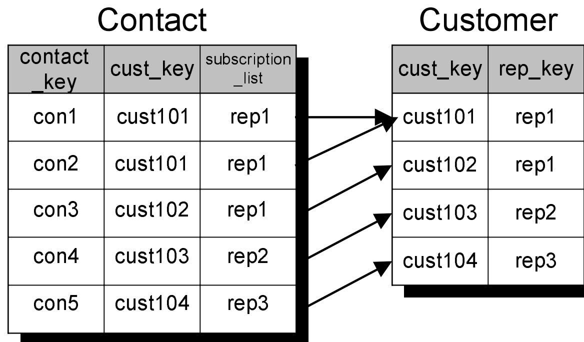
Contact table definition

In the case of the Contact table, the table definition would be changed to the following:

```
CREATE TABLE Contact (
   contact_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   subscription_list CHAR( 12 ) NULL,
   FOREIGN KEY ( cust_key )
   REFERENCES Customer ( cust_key ),
   PRIMARY KEY ( contact_key )
)
go
```

The additional column is created allowing NULL, so that existing applications can continue to work against the database without change.

The **subscription_list** column holds the **rep_key** value corresponding to the row with primary key value **cust_key** in the Customer table. A set of triggers handles maintenance of the **subscription_list** column.



For an Adaptive Server Anywhere consolidated database, the solution is different. For more information, see "Partitioning tables that do not contain the subscription expression" on page 125.

Maintaining the subscription-list column

In order to keep the **subscription_list** column up to date, triggers are needed for the following operations:

- ♦ INSERT on the Contact table.
- ♦ UPDATE on the Contact table.
- ♦ UPDATE on the Customer table.

The UPDATE of the Customer table addresses the **territory realignment** problem, where customers are assigned to different Sales Reps.

An INSERT trigger for the Contact table

The trigger for an INSERT on the Contact table sets the subscription_list value to the corresponding **rep_key** value from the Customer table:

```
CREATE TRIGGER set_contact_sub_list
ON Contact
FOR INSERT
AS
BEGIN
   UPDATE Contact
   SET Contact.subscription_list = (
         SELECT rep_key
         FROM Customer
         WHERE Contact.cust_key = Customer.cust_key )
   WHERE Contact.contact_key IN (
         SELECT contact_key
         FROM inserted
   )
END
```

The trigger updates the **subscription_list** column for those rows being inserted; these rows being identified by the subquery

```
SELECT contact_key
FROM inserted
```

An UPDATE trigger for the Contact table

The trigger for an UPDATE on the Contact table checks to see if the **cust_key** column is changed, and if it has updates the **subscription_list** column.

```
CREATE TRIGGER update_contact_sub_list
ON Contact
FOR UPDATE
AS
IF UPDATE ( cust_key )
   BEGIN
      UPDATE Contact
      SET subscription_list = Customer.rep_key
      FROM Contact, Customer
      WHERE Contact.cust_key=Customer.cust_key
   END
```

The trigger is written using a join; a subquery could also have been used.

An UPDATE trigger for the Customer table

The following trigger handles UPDATES of customers, transferring them to a new Sales Rep:

```
CREATE TRIGGER transfer_contact_with_customer
ON Customer
FOR UPDATE
AS
IF UPDATE ( rep_key )
BEGIN
   UPDATE Contact
   SET Contact.subscription_list = (
      SELECT rep_key
      FROM Customer
      WHERE Contact.cust_key = Customer.cust_key )
   WHERE Contact.contact_key IN (
         SELECT cust_key
      FROM inserted
   )
END
```

Sharing rows among several subscriptions

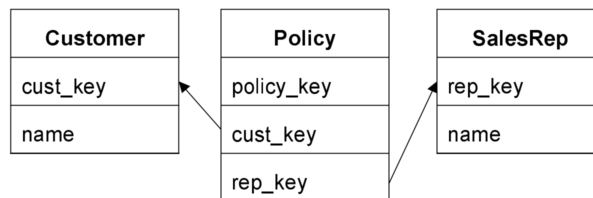
There are cases where a row may need to be included in several subscriptions. For example, if instead of the many-to-one relationship between customers and sales representatives that we had above, we may have a many-to-many relationship.

The Policy example

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

Example database

Here is a simple database that illustrates the problem.



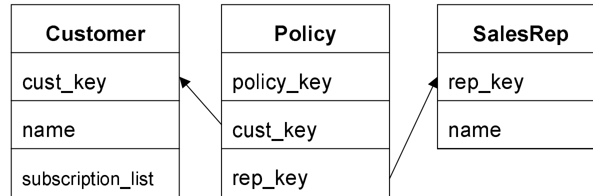
The Policy table has a row for each of a set of policies. Each policy is drawn up for a customer by a particular sales representative. There is a many-to-many relationship between customers and sales representatives, and there may be several policies drawn up between a particular rep/customer pair.

Any row in the Customer table may need to be shared with none, one, or several sales representatives.

Solving the problem

To support this case, you need to write triggers to build a comma-delimited list of values to store in a redundant subscription-list column of the Customer table, and include this column as the subscription column when adding the Customer table to the publication. The row is shared with any subscription for which the subscription value matches any of the values in the subscription-list column.

The database, with the subscription-list column included, is as follows:



Adaptive Server Enterprise VARCHAR columns are limited to 255 characters, and this limits the number of values that can be stored in the comma-delimited list.

Table definitions

The table definitions are as follows:

```
CREATE TABLE SalesRep (
   rep_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   PRIMARY KEY ( rep_key )
)
go
CREATE TABLE Customer (
   cust_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   subscription_list VARCHAR( 255 ) NULL,
   PRIMARY KEY ( cust_key )
)
go
CREATE TABLE Policy (
   policy_key   INTEGER NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   rep_key CHAR( 12 ) NOT NULL,
   FOREIGN KEY  ( cust_key )
   REFERENCES Customer (cust_key ),
   FOREIGN KEY (rep_key )
   REFERENCES SalesRep ( rep_key ),
   PRIMARY KEY  (policy_key)
)
```

Notes:

- The **subscription_list** column in the Customer table allows NULLs so that customers can be added who do not have any sales representatives in the **subscription_list** column.

The publication

The publication for this database can be created by the following set of statements:

```
//Mark the tables for replication
exec sp_add_remote_table 'SalesRep'
exec sp_add_remote_table 'Policy'
exec sp_add_remote_table 'Customer'
```

```
go

// Create an empty publication
exec sp_create_publication 'SalesRepData'

//Add the Sales Rep table to the publication
exec sp_add_article 'SalesRepData', 'SalesRep'

//Add the Policy table to the publication
exec sp_add_article 'SalesRepData', 'Policy', NULL,
'rep_key'

// Add the Customer table to the publication.
// Subscribe by the subscription_list column
// Exclude the subscription_list column
exec sp_add_article 'SalesRepData', 'Customer', NULL,
'subscription_list'
exec sp_add_article_col 'SalesRepData', 'Customer',
'cust_key'
exec sp_add_article_col 'SalesRepData', 'Customer',
'name'
go
```

The subscriptions

Subscriptions to this publication take the following form:

```
exec sp_subscription 'create', 'SalesRepData', 'userID',
'rep_key'
go
```

where *userID* identifies the subscriber, and *rep_key* is the subscription column, which is the value of the **rep_key** column in the **SalesRep** table.

Maintaining the subscription-list column

You need to write a procedure and a set of triggers to maintain the subscription-list column added to the Customer table. This section describes these objects.

Stored procedure

The following procedure is used to build the subscription-list column, and is called from the triggers that maintain the subscription_list column.

```
CREATE PROCEDURE SubscribeCustomer @cust_key CHAR(12)
AS
BEGIN
   --  Rep returns the list of reps for customer
@cust_key
   DECLARE Rep CURSOR FOR
      SELECT DISTINCT RTRIM( rep_key )
      FROM Policy
      WHERE cust_key = @cust_key
```

```
    DECLARE @rep_key    CHAR(12)
    DECLARE @subscription_list   VARCHAR(255)

    -- build comma-separated list of rep_key
    -- values for this Customer
    OPEN Rep
    FETCH Rep INTO @rep_key
    IF @@sqlstatus = 0 BEGIN
       SELECT @subscription_list = @rep_key
       WHILE 1=1 BEGIN
          FETCH Rep INTO @rep_key
          IF @@sqlstatus != 0 BREAK
          SELECT @subscription_list =
             @subscription_list + ',' + @rep_key
       END
    END
    ELSE BEGIN
       SELECT @subscription_list = ''
    END

    -- update the subscription_list in the
    -- Customer table
    UPDATE Customer
    SET subscription_list = @subscription_list
    WHERE cust_key = @cust_key
END
```

Notes:

- The procedure takes a Customer key as input argument.
- Rep is a cursor for a query that lists each of the Sales Representatives with which the customer has a contract.
- The WHILE loop builds a VARCHAR(255) variable holding the comma-separated list of Sales Representatives.
- The UPDATE **subscription_list** column of the Customer

Triggers

The following trigger updates the **subscription_list** column of the Customer table when a row is inserted into the Policy table.

```
CREATE TRIGGER InsPolicy
ON Policy
FOR INSERT
AS
BEGIN
   -- Cust returns those customers inserted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM inserted
   DECLARE @cust_key CHAR(12)

   OPEN Cust
   -- Update the rep list for each Customer
```

```
      -- with a new rep
      WHILE 1=1 BEGIN
         FETCH Cust INTO @cust_key
         IF @@sqlstatus != 0 BREAK
         EXEC SubscribeCustomer @cust_key
      END
   END
```

The following trigger updates the **subscription_list** column of the Customer table when a row is deleted from the Policy table.

```
CREATE TRIGGER DelPolicy
ON Policy
FOR DELETE
AS
BEGIN
   -- Cust returns those customers deleted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM deleted
   DECLARE @cust_key CHAR(12)

   OPEN Cust
   -- Update the rep list for each Customer
   -- losing a rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
```

Excluding the subscription-list column from the publication

The subscription-list column should be excluded from the publication, as inclusion of the column leads to excessive updates being replicated.

For example, consider what happens if there are many policies per customer. If a new Sales Representative is assigned to a customer, a trigger fires to update the subscription-list column in the Customer table. If the subscription-list column is part of the publication, then one update for each policy will be replicated to all sales reps that are assigned to this customer.

Triggers at the consolidated database only

The values in the subscription-list column are maintained by triggers. These triggers fire at the consolidated database when the triggering inserts or updates are applied by the Message Agent. The triggers must be excluded from the remote databases, as they maintain a column that does not exist.

You can use the **sp_user_extraction_hook** procedure to exclude only certain triggers from a remote database on extraction. The procedure is called as the final part of an extraction. By default, it is empty.

❖ **To customize the extraction procedure to omit certain triggers:**

1. Ensure the **quoted_identifier** option is set to ON:

```
set quoted_identifier on
go
```

2. Any temporary tables referenced in the procedure must exist, or the CREATE PROCEDURE statement will fail. The temporary tables referenced in the following procedure are available in the *ssremote.sql* script. Copy any required table definitions from the script and execute the CREATE TABLE statements, so they exist on the current connection, before creating the procedure.

3. Create the following procedure:

```
CREATE PROCEDURE sp_user_extraction_hook
AS
BEGIN
  -- We do not want to extract the INSERT and
  -- DELETE triggers created on the Policy table
  -- that maintain the subscription_list
  -- column, since we do not include that
  -- column in the publication.
  -- If these objects were extracted the
  -- INSERTs would fail on the remote database
  -- since they reference a column
  -- ( subscription_list ) that does not exist.
  DELETE FROM #systrigger
  WHERE table_id = object_id( 'Policy' )
  -- Do not create any procedures
  DELETE FROM #sysprocedure
  WHERE proc_name = 'SubscribeCustomer'
END
go
```

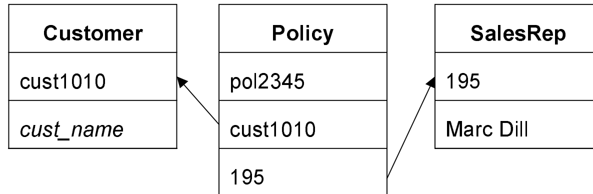
Using the Subscribe_by_remote option with many-to-many relationships

When the Subscribe_by_remote option is ON, operations that arrive from remote databases on rows with a subscribe by value of NULL or '' will assume the remote user is subscribed to the row. By default, the Subscribe_by_remote option is set to ON. In most cases, this setting is the desired setting.

The Subscribe_by_remote option solves a problem that otherwise would arise with publications including the Policy example. This section describes how the option automatically avoids the problem.

The database uses a subscription-list column for the Customer table, because each Customer may belong to several Sales Reps:

Marc Dill is a Sales Rep who has just arranged a policy with a new customer. He inserts a new Customer row and also inserts a row in the Policy table to assign the new Customer to himself. Assuming that the subscription-list column is not included in the publication, the operation at Marc's remote database is as follows:



As the INSERT of the Customer row is carried out by the Message Agent at the consolidated database, Adaptive Server Enterprise records the subscription value in the transaction log, at the time of the INSERT.

Later, when the Message Agent scans the log, it builds a list of subscribers to the new row, using the subscription value stored in the log, and Marc Dill is not on that list. 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 operation.

As long as Subscribe_by_remote is set to ON, the Message Agent assumes that, as the subscription-list column is NULL, the row belongs to the Sales Rep that inserted it. As a result, the INSERT is not replicated back to Marc Dill, and the replication system is intact.

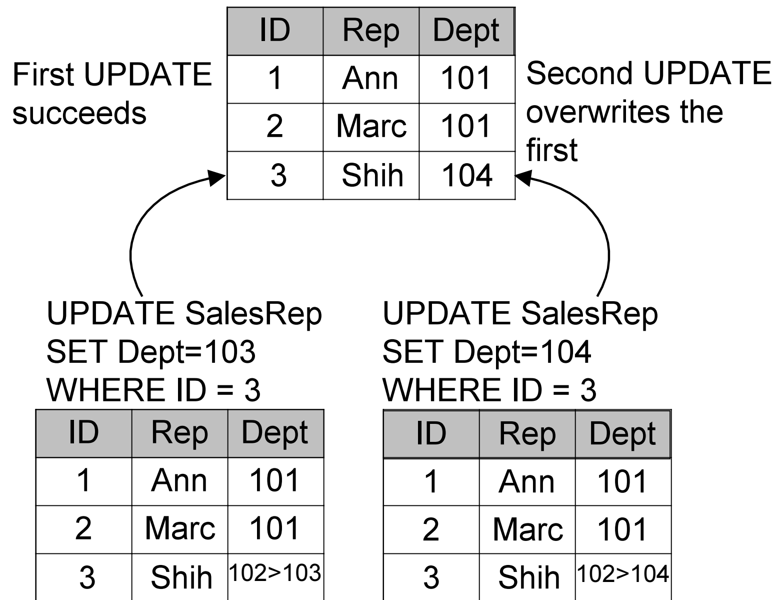
You can use a trigger, which executes after the INSERT, to maintain the subscription-list column.

Managing conflicts

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

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

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



Default conflict resolution

By default, the UPDATE still proceeds, so that the User 2 update (the last to reach the consolidated database) becomes the value in the consolidated database, and is replicated to all other databases subscribed to that row. In general, the default method of conflict resolution is that the most recent operation (in this case that from User 2) succeeds, and no report is made of the conflict. The update from User 1 is lost.

SQL Remote also allows custom conflict resolution, using a stored procedure to resolve conflicts in a way that makes sense for the data being changed.

Conflicts do not apply to primary keys

UPDATE conflicts do not apply to primary key updates. If the column being updated is a primary key, then when the update from User 2 arrives at the consolidated database, no row will be updated.

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

How SQL Remote handles conflicts

When a conflict is detected

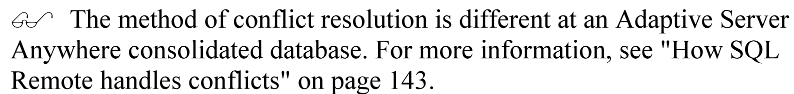
SQL Remote replication messages include UPDATE statements as a set of single row updates, each including the values prior to updating.

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

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

1. The UPDATE is applied.
2. Any conflict resolution procedures are called.

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



The method of conflict resolution is different at an Adaptive Server Anywhere consolidated database. For more information, see "How SQL Remote handles conflicts" on page 143.

Implementing conflict resolution

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

Required objects

For each table on which you wish to resolve conflicts, you must create three database objects to handle the resolution:

- **An old value table** To hold the values that were stored in the table when the conflicting message arrived.
- **A remote value table** To hold the values stored in the table at the remote database when the conflicting update was applied, as determined from the message.

- **A stored procedure** To carry out actions to resolve the conflict.

These objects need to exist only in the consolidated database, as that is where conflict resolution occurs. They should not be included in any publications.

Naming the objects

When a table is marked for replication, using the **sp_add_remote_table** or **sp_modify_remote_table** stored procedure, optional parameters specify the names of the conflict resolution objects.

The **sp_add_remote_table** and **sp_modify_remote_table** procedures take one compulsory argument, which is the name of the table being marked for replication. It takes three additional arguments, which are the names of the objects used to resolve conflicts. For example, the syntax for **sp_add_remote_table** is:

```
exec sp_add_remote_table table_name
    [ , resolve_procedure ]
    [ , old_row_table ]
    [ , remote_row_table ]
```

You must create each of the three objects *resolve_procedure, old_row_table*, and *remote_row_table*. These three are discussed in turn.

- **old_row_table** This table must have the same column names and data types as the table *table_name*, but should not have any foreign keys. When a conflict occurs, a row is inserted into *old_row_table* containing the values of the row in *table_name* being updated before the UPDATE was applied. Once *resolve_procedure* has been run, the row is deleted.

  As the Message Agent applies updates as a set of single-row updates, the table only ever contains a single row.

- **remote_row_table** This table must have the same column names and data types as the table *table_name*, but should not have any foreign keys. When a conflict occurs, a row is inserted into *remote_row_table* containing the values of the row in *table_name* from the remote database before the UPDATE was applied. Once *resolve_procedure* has been run, the row is deleted.

  As the Message Agent applies updates as a set of single-row updates, the table only ever contains a single row.

- **resolve_procedure** This procedure carries out whatever actions are required to resolve a conflict, which may include altering the value in the row or reporting values into a separate table.

Once these objects are created, you must run the **sp_add_remote_table** or **sp_modify_remote_table** procedure to flag them as conflict resolution objects for a table.

Limitations

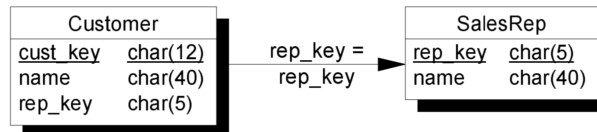
- At an Adaptive Server Enterprise database, conflict resolution will not work on a table with more than 128 columns while the VERIFY_ALL_COLUMNS option is set to ON. Even if VERIFY_ALL_COLUMNS is set to OFF, if an UPDATE statement updates more than 128 columns, conflict resolution will not work.

A first conflict resolution example

In this example, conflicts in the Customer table in the two-table example used in the tutorials are reported into a table for later review.

The database

The two-table database is as follows:



Goals of the conflict resolution

The conflict resolution will report conflicts on updates to the **name** column in the Customer table into a separate table named **ConflictLog**.

The conflict resolution objects

The conflict resolution tables are defined as follows:

```
CREATE TABLE OldCustomer(
   cust_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   rep_key CHAR( 5 ) NOT NULL,
   PRIMARY KEY ( cust_key )
)
```

```
CREATE TABLE RemoteCustomer(
   cust_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   rep_key CHAR( 5 ) NOT NULL,
   PRIMARY KEY ( cust_key )
)
```

Each of these tables has exactly the same columns and data types as the Customer table itself. The only difference in their definition is that they do not have a foreign key to the **SalesRep** table.

The conflict resolution procedure reports conflicts into a table named **ConflictLog**, which has the following definition:

```
CREATE TABLE ConflictLog (
   conflict_key numeric(5, 0) identity not null,
   lost_name char(40) not null ,
   won_name char(40) not null ,
   primary key ( conflict_key )
```

```
)
```

The conflict resolution procedure is as follows:

```
CREATE PROCEDURE ResolveCustomer
AS
BEGIN
   DECLARE @cust_key CHAR(12)
   DECLARE @lost_name CHAR(40)
   DECLARE @won_name CHAR(40)

   // Get the name that was lost
   // from OldCustomer
   SELECT @lost_name=name,
      @cust_key=cust_key
   FROM OldCustomer

   // Get the name that won
   // from Customer
   SELECT @won_name=name
   FROM Customer
   WHERE cust_key = @cust_key

   INSERT INTO ConflictLog ( lost_name, won_name )
   VALUES ( @lost_name, @won_name )
END
```

This resolution procedure does not use the **RemoteCustomer** table.

How the conflict resolution works

The stored procedure is the key to the conflict resolution. It works as follows:

1 Obtains the **@lost_name** value from the **OldCustomer** table, and also obtains a primary key value so that the real table can be accessed.

   The **@lost_name** value is the value that was overridden by the conflict-causing UPDATE.

2 Obtains the **@won_name** value from the Customer table itself. This is the value that overrode **@lost_name**. The stored procedure runs *after* the update has taken place, which is why the value is present in the Customer table. This behavior is different from SQL Remote for Adaptive Server Enterprise, where conflict resolution is implemented in a BEFORE trigger.

3 Adds a row into the **ConflictLog** table containing the **@lost_name** and **@won_name** values.

4 After the procedure is run, the rows in the **OldCustomer** and **RemoteCustomer** tables are deleted by the Message Agent. In this simple example, the **RemoteCustomer** row was not used.

Testing the example

❖ **To test the example:**

1. Create the tables and the procedure in the consolidated database, and add them as conflict resolution objects to the Customer table.
2. Insert and commit a change at the consolidated database. For example:

```
UPDATE Customer
SET name = 'Sea Sports'
WHERE cust_key='cust1'
go
COMMIT
go
```

3. Insert and commit a different change to the same line at the remote database. For example:

```
UPDATE Customer
SET name = 'C Sports'
WHERE cust_key='cust1'
go
COMMIT
go
```

4. Replicate the change from the remote to the consolidated database, by running the Message Agent at the remote database to send the message, and then at the consolidated database to receive and apply the message.
5. At the consolidated database, view the **Customer** table and the **ConflictLog** table. The Customer table contains the value from the remote database:

cust_key	name	rep_key
cust1	C Sports	rep1

The **ConflictLog** table has a single row, showing the conflict:

conflict_key	lost_name	won_name
1	Sea Sports	C Sports

A second conflict resolution example

This example shows a slightly more elaborate example of resolving a conflict, based on the same situation as the previous example, discussed in "A first conflict resolution example" on page 185.

Goals of the conflict resolution

In this case, the conflict resolution has the following goals:

- Disallow the update from a remote database. The previous example allowed the update.
- Report the name of the remote user whose update failed, along with the lost and won names.

The conflict resolution objects

In this case, the **ConflictLog** table has an additional column to record the user ID of the remote user. The table is as follows:

```
CREATE TABLE ConflictLog (
   conflict_key numeric(5, 0) identity not null,
   lost_name char(40) not null ,
   won_name char(40) not null ,
   remote_user char(40) not null ,
   primary key ( conflict_key )
)
```

The stored procedure is more elaborate. As the update will be disallowed, rather than allowed, the **lost_name** value now refers to the value arriving in the message. It is first applied, but then the conflict resolution procedure replaces it with the value that was previously present.

The stored procedure uses data from the temporary table **#remote**. In order to create a procedure that references a temporary table you first need to create that temporary table. The statement is as follows:

```
CREATE TABLE #remote (
   current_remote_user varchar(128),
   current_publisher varchar(128)
)
```

This table is created in TEMPDB, and exists only for the current session. The Message Agent creates its own #remote table when it connects, and uses it when the procedure is executed.

```
CREATE PROCEDURE ResolveCustomer
AS
BEGIN
   DECLARE @cust_key CHAR(12)
   DECLARE @lost_name CHAR(40)
   DECLARE @won_name CHAR(40)
   DECLARE @remote_user varchar(128)

   -- Get the name that was present before
   -- the message was applied, from OldCustomer
   -- This will "win" in the enx
   SELECT @won_name=name,
      @cust_key=cust_key
   FROM OldCustomer
```

```
      -- Get the name that was applied by the
      -- Message Agent from Customer. This will
      -- "lose" in the end
      SELECT @lost_name=name
      FROM Customer
      WHERE cust_key = @cust_key

      -- Get the remote user value from #remote
      SELECT @remote_user = current_remote_user
      FROM #remote

      -- Report the problem
      INSERT INTO ConflictLog ( lost_name,
         won_name, remote_user )
      VALUES ( @lost_name, @won_name, @remote_user )

      -- Disallow the update from the Message Agent
      -- by resetting the row in the Customer table
      UPDATE Customer
      SET name = @won_name
      WHERE cust_key = @cust_key

   END
```

Notes

There are several points of note here:

- The user ID of the remote user is stored by the Message Agent in the **current_remote_user** column of the temporary table **#remote**.
- The UPDATE from the Message Agent is applied before the procedure runs, so the procedure has to explicitly replace the values. This is different from the case in SQL Remote for Adaptive Server Anywhere, where conflict resolution is carried out by BEFORE triggers.

Testing the example

❖ **To test the example:**

1. Create the tables and the procedure in the consolidated database, and add them as conflict resolution objects to the Customer table.
2. Insert and commit a change at the consolidated database. For example:

   ```
   UPDATE Customer
   SET name = 'Consolidated Sports'
   WHERE cust_key='cust1'
   go
   COMMIT
   go
   ```

3. Insert and commit a different change to the same line at the remote database. For example:

   ```
   UPDATE Customer
   SET name = 'Field Sports'
   ```

```
WHERE cust_key='cust1'
go
COMMIT
go
```

4 Replicate the change from the remote to the consolidated database, by running the Message Agent at the remote database to send the message, and then at the consolidated database to receive and apply the message.

5 At the consolidated database, view the **Customer** table and the **ConflictLog** table. The Customer table contains the value from the consolidated database:

cust_key	name	rep_key
cust1	Consolidated Sports	rep1

The **ConflictLog** table has a single row, showing the conflict and recording the value entered at the remote database:

conflict_key	lost_name	won_name	remote_user
1	Field Sports	Consolidated Sports	field_user

6 Run the Message Agent again at the remote database. This receives the corrected update from the consolidated database, so that the name of the customer is set to Consolidated Sports here as well.

Designing to avoid referential integrity errors

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

Referential integrity errors stop replication
If a remote database receives a message that includes a statement that cannot be executed because of referential integrity constraints, no further messages can be applied to the database (because they come after a message that has not yet been applied), including passthrough statements, which would sit in the message queue.

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

Unreplicated referenced table errors

Consider the following **SalesRepData** publication:

```
exec sp_add_remote_table 'SalesRep'
exec sp_create_publication 'SalesRepData'
exec sp_add_article 'SalesRepData', 'SalesRep'
go
```

If the **SalesRep** table had a foreign key to another table (say, **Employee**) that was not included in the publication, inserts or updates to **SalesRep** would fail to replicate unless the remote database had the foreign key reference removed.

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

Ensuring unique primary keys

Users at physically distinct sites can each INSERT new rows to a table, so there is an obvious problem ensuring that primary key values are kept unique.

If two users INSERT a row using the same primary key values, the second INSERT to reach a given database in the replication system will fail. As SQL Remote is a replication system for occasionally-connected users, there can be no locking mechanism across all databases in the installation. It is necessary to design your SQL Remote installation so that primary key errors do not occur.

For primary key errors to be designed out of SQL Remote installations; the primary keys of tables that may be modified at more than one site must be guaranteed unique. There are several ways of achieving this goal. This chapter describes a general, economical and reliable method that uses a pool of primary key values for each site in the installation.

Overview of primary key pools

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

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

The primary key pool

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

```
CREATE TABLE KeyPool (
   table_name VARCHAR(40) NOT NULL,
   value INTEGER NOT NULL,
   location VARCHAR(6) NOT NULL,
   PRIMARY KEY (table_name, value),
)
go
```

The columns of this table have the following meanings:

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

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

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

Replicating the primary key pool

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

❖ To replicate the primary key pool:

1 Create a publication for the primary key pool data.

```
sp_create_publication 'KeyPoolData'
go
sp_add_remote_table 'KeyPool'
go
sp_add_article 'KeyPoolData', 'KeyPool',
   NULL, 'location'
go
```

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

```
sp_subscription 'create',
   KeyPoolData,
   field_user,
   rep1
go
```

The subscription argument is the location identifier.

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

Filling and replenishing the key pool

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

```
CREATE PROCEDURE ReplenishPool AS
BEGIN
   DECLARE @CurrTable  VARCHAR(40)
   DECLARE @MaxValue   INTEGER
   DECLARE EachTable   CURSOR FOR
            SELECT table_name, max(value)
            FROM KeyPool
            GROUP BY table_name
   DECLARE @CurrLoc    VARCHAR(6)
   DECLARE @NumValues  INTEGER
   DECLARE EachLoc     CURSOR FOR
            SELECT location, count(*)
            FROM KeyPool
            WHERE table_name = @CurrTable
            GROUP BY location
   OPEN EachTable
   WHILE 1=1 BEGIN
      FETCH EachTable INTO @CurrTable, @MaxValue
      IF @@sqlstatus != 0 BREAK
      OPEN EachLoc
      WHILE 1=1 BEGIN
         FETCH EachLoc INTO @CurrLoc, @NumValues
         IF @@sqlstatus != 0 BREAK
         -- make sure there are 10 values
         WHILE @NumValues < 10 BEGIN
            SELECT @MaxValue = @MaxValue + 1
            SELECT @NumValues = @NumValues + 1
            INSERT INTO KeyPool
                    (table_name, location, value)
            VALUES (@CurrTable, @CurrLoc, @MaxValue)
         END
      END
      CLOSE EachLoc
   END
   CLOSE EachTable
END
go
```

This procedure fills the pool for each user up to ten values. You may wish to use a larger value in a production environment. The value you need depends on how often users are inserting rows into the tables in the database.

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

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

```
INSERT INTO KeyPool VALUES( 'Customer', 40, 'rep1' )
INSERT INTO KeyPool VALUES( 'Customer', 41, 'rep2' )
INSERT INTO KeyPool VALUES( 'Customer', 42, 'rep3' )
INSERT INTO KeyPool VALUES( 'Customer', 43, 'Office')
EXEC ReplenishPool
go
```

Cannot use a trigger to replenish the key pool
You cannot use a trigger to replenish the key pool, as no actions are replicated to the remote database performing the original operation, including trigger actions.

Adding new customers

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

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

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

```
CREATE PROCEDURE NewKey
        @TableName VARCHAR(40),
        @Location VARCHAR(6),
        @Value INTEGER OUTPUT AS
    BEGIN
        DECLARE @NumValues INTEGER
        SELECT @NumValues = count(*),
               @Value = min(value)
        FROM KeyPool
```

```
         WHERE table_name = @TableName
         AND location = @Location
         IF @NumValues > 1
            DELETE FROM KeyPool
            WHERE table_name = @TableName
            AND value = @Value
         ELSE
            -- Never take the last value,
            -- because RestorePool will not work.
            -- The key pool should be kept large
            -- enough so this never happens.
            SELECT @Value = NULL
      END
```

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

```
CREATE PROCEDURE NewCustomer @name VARCHAR(40),
         @loc VARCHAR(6) AS
BEGIN
   DECLARE @cust  INTEGER
   DECLARE @cust_key  VARCHAR(12)

   EXEC NewKey 'Customer', @loc, @cust output
   SELECT @cust_key = 'cust' +
      convert( VARCHAR(12), @cust )
   INSERT INTO Customer (cust_key, name, rep_key )
   VALUES ( @cust_key, @name, @loc )
END
```

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

Testing the key pool

❖ **To test the primary key pool:**

1 Re-extract a remote database using the field_user user ID.

2 Try this sample INSERT at the remote and consolidated sites:

```
EXEC NewCustomer 'Great White North', rep1
```

Primary key pool summary

The primary key pool technique requires the following components:

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

Creating subscriptions

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

Subscriptions with no subscription column

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

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

The following statement creates a subscription for a user ID **SamS** to the **pub_orders_samuel_singer** publication, which was created without using a subscription column:

```
sp_subscription 'create',
   'pub_orders_samuel_singer',
   'SamS'
```

Subscriptions with a subscription column

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

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

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

```
sp_subscription create,
   pub_orders,
   SamS,
   '856'
```

Starting a subscription

In order to receive and apply updates properly, each subscriber needs to have an initial copy of the data. The synchronization process is discussed in "Synchronizing databases" on page 207.