

CHAPTER 7

SQL Remote Design for Adaptive Server Anywhere

About this chapter

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

Similar material for Adaptive Server Enterprise

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 Enterprise users, "SQL Remote Design for Adaptive Server Enterprise" on page 159.

Contents

Topic	Page
Design overview	114
Creating publications	115
Publication design for Adaptive Server Anywhere	122
Partitioning tables that do not contain the subscription expression	125
Sharing rows among several subscriptions	133
Managing conflicts	142
Ensuring unique primary keys	152
Creating subscriptions	157

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 Adaptive Server Anywhere Database Administrator should perform all SQL Remote configuration tasks.

Creating publications

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 Anywhere Users" on page 37.

Creating publications for Adaptive Server Anywhere using Sybase Central

You can add a publication to a database from within the SQL Remote folder of Sybase Central.

❖ To create a publication from Sybase Central:

- 1 Open the SQL Remote folder for your database, which is inside the database container.
- 2 Click the Publications folder.
- 3 Double-click Add Publication. The Publication Wizard is displayed.
- 4 Follow the instructions in the Wizard.

✍ For more information on how to use Sybase Central, 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.

To send SQL statements to an Adaptive Server Anywhere database, you can use the Interactive SQL utility.

Publishing a whole table

The simplest publication you can make consists of a single article, which consists of a single, entire, table.

❖ **To create a publication that includes all the rows and columns of a single table:**

- ◆ Execute a create publication statement specifying the table you wish to publish. The syntax is as follows:

```
CREATE PUBLICATION publication_name (
    TABLE table_name
)
```

Example

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

```
CREATE PUBLICATION pub_customer (
    TABLE customer
)
```

Publishing some of the columns in a table

Partitioning tables

An article that contains only some of the columns of a table can be called a **column-wise partitioning** of the table. An article that contains only some of the rows of a table can be called a **row-wise partitioning** of the table. Articles can be both column-wise and row-wise partitions of a table.

☞ Only column-wise partitioning is described in this section. For information on row-wise partitioning of tables, see "Publishing some of the rows from a table" on page 117.

You create a publication that contains all the rows, but only some of the columns, of a table by specifying a list of columns in the CREATE PUBLICATION statement.

☞ Any article must conform to the requirements listed in "Conditions for valid articles" on page 123.

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

- ◆ Enter a CREATE PUBLICATION statement that includes the column names you wish to include. The syntax is as follows:

```
CREATE PUBLICATION publication_name (
    TABLE table_name ( column_name, ... )
)
```

Example

- ◆ The following statement creates a publication that publishes all rows of the **id**, **company_name**, and **city** columns of the **customer** table:

```
CREATE PUBLICATION pub_customer (
    TABLE customer (
        id,
```

```

        company_name,
        city )
    )

```

Publishing a set of tables

When publishing a set of tables, a separate article is required for each table in the publication. The following statement creates a publication including all columns and rows in each of a set of tables from the Adaptive Server Anywhere sample database:

```

CREATE PUBLICATION sales (
    TABLE customer,
    TABLE sales_order,
    TABLE sales_order_items,
    TABLE product
)

```

Not all the tables in the database have been published. For example, subscribers receive the **sales_order** table, but not the **employee** table.

Notes

- ◆ In the sample database, the **sales_order** and **employee** tables are related by a foreign key (**sales_rep** in the **sales_order** table is a foreign key to the **emp_id** column in the **employee** table). Although this could lead to referential integrity problems in the remote database, they are easily avoided by using the database extraction utility.

☞ For a discussion of this and other publication design issues, see "Designing to avoid referential integrity errors" on page 149.

- ◆ A slightly different publication design (including an article that contains enough of the **employee** table to satisfy the foreign key relationship) would make the publication more robust; for this section we are publishing whole tables only.

Publishing some of the rows from a table

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

When to use WHERE and subscription expressions

You can combine a WHERE clause and a subscription expression in an article.

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.

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

Publishing a subset of rows using a WHERE clause

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

```
CREATE PUBLICATION pub_orders_samuel_singer (  
    TABLE sales_order WHERE sales_rep = 856  
)
```

In Sybase Central, the Publication Wizard guides you through creating a WHERE clause for an article.

❖ To create a publication using a WHERE clause:

- ◆ Enter a CREATE PUBLICATION statement that includes the rows you wish to include in a WHERE clause. The syntax is as follows:

```
CREATE PUBLICATION publication_name (  
    TABLE table_name ( column_name, ... )  
    WHERE search-condition  
)
```

Example

- ◆ The following statement creates a publication that publishes the **id**, **company_name**, **city**, and **state** columns of the **customer** table, for the customers marked as active in the **status** column.

```
CREATE PUBLICATION pub_customer (  
    TABLE customer (  
        id,  
        company_name,  
        city,  
        state )  
    WHERE status = 'active'  
)
```

In this case, the **status** column is not included in the publication. It must therefore have a default value so that inserts at remote databases will not fail at the consolidated database.

Publishing a subset of rows using a subscription expression

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

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

```
CREATE PUBLICATION pub_orders_samuel_singer (
  TABLE sales_order
  WHERE sales_rep = 856
)
```

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

Benefits of subscription expressions

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

❖ To create an article using a subscription expression:

- ◆ Enter a CREATE PUBLICATION statement that includes the expression you wish to use as a match in the subscription expression. The syntax is as follows:

```
CREATE PUBLICATION publication_name (
  TABLE table_name ( column_name, ... )
  SUBSCRIBE BY expression
)
```

Example

- ◆ The following statement creates a publication that publishes the **id**, **company_name**, **city**, and **state** columns of the **customer** table, and which matches the rows with subscribers according to the value of the **state** column:

```
CREATE PUBLICATION pub_customer (
  TABLE customer (
    id,
    company_name,
    city,
    state )
  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 pub_customer ('GA')
FOR Ann_Taylor ;
```

```
CREATE SUBSCRIPTION
TO pub_customer ('MA')
FOR Sam_Singer
```

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

For more information

- ◆ For information on how to use subqueries in a publication, see "Partitioning tables that do not contain the subscription expression" on page 125.
- ◆ For more information on creating subscriptions, see "Creating subscriptions" on page 157.
- ◆ In Sybase Central, the Publication Wizard guides you through creating a subscription expression for an article.

Dropping publications

Publications can be dropped using the DROP PUBLICATION statement. The following statement drops the publication named **pub_orders**.

```
DROP PUBLICATION pub_orders
```

Dropping a publication has the side effect of dropping all subscriptions to that publication.

Notes on publications

- ◆ The different publication types described above can be combined. A single publication can publish a subset of columns from a set of tables, and use both a WHERE clause to select a set of rows to be replicated and a subscription expression to partition rows by subscription.
- ◆ DBA authority is required to create publications.
- ◆ Publications can be altered and dropped only by the DBA.

- ◆ Altering publications in a running SQL Remote setup is likely to cause replication errors, and could lead to loss of data in the replication system unless carried out with care.
- ◆ Views cannot be included in publications.
- ◆ Stored procedures cannot be included in publications. For a discussion of how SQL Remote replicates procedures and triggers, see "Replication of procedures" on page 102 .
- ◆ For other considerations of referential integrity, see the section "Designing to avoid referential integrity errors" on page 149.

Publication design for Adaptive Server Anywhere

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

Similar material for Adaptive Server Enterprise

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 section and the corresponding section for Adaptive Server Enterprise users, "Publication design for Adaptive Server Enterprise" on page 166.

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

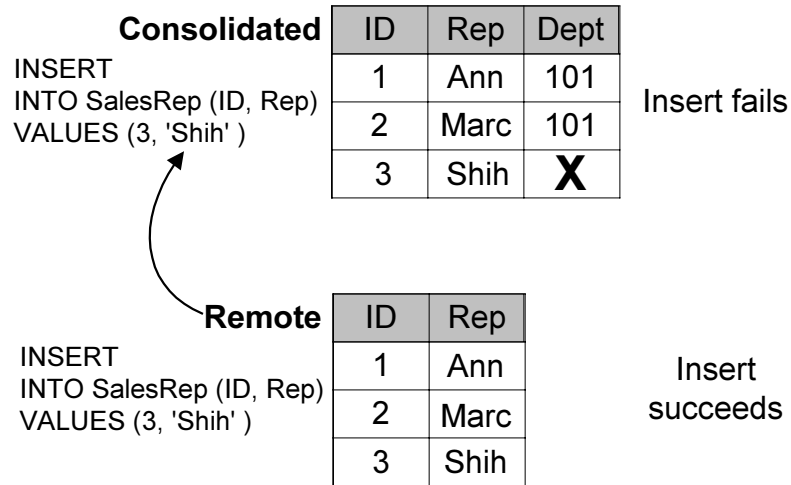
Supporting
INSERTS at
remote databases

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

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

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

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



Using BEFORE triggers as an alternative

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

Design tips for performance

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

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

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

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

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

Partitioning tables that do not contain the subscription expression

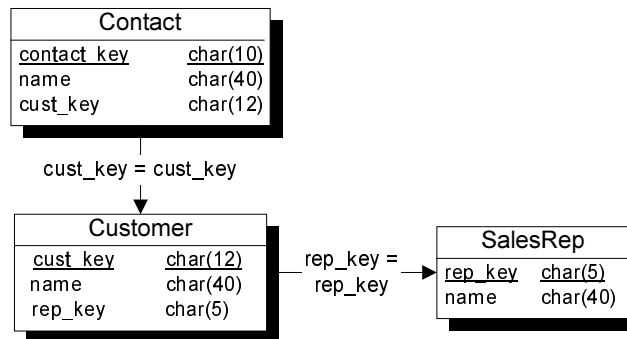
In many cases, the rows of a table need to be partitioned even when the subscription expression does not exist in the table.

The Contact example

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

Example

Here is a simple database that illustrates the problem.



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

The tables in the database

The three tables are described in more detail as follows:

Table	Description
SalesRep	<p>All sales representatives that work for the company. The SalesRep table has the following columns:</p> <ul style="list-style-type: none">◆ rep_key An identifier for each sales representative. This is the primary key.◆ name The name of each sales representative. <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE SalesRep (rep_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, PRIMARY KEY (rep_key))</pre>
Customer	<p>All customers that do business with the company. The Customer table includes the following columns:</p> <ul style="list-style-type: none">◆ cust_key An identifier for each customer. This is the primary key.◆ name The name of each customer.◆ rep_key An identifier for the sales representative in a sales relationship. This is a foreign key to the SalesRep table. <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE Customer (cust_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, rep_key CHAR(12) NOT NULL, FOREIGN KEY REFERENCES SalesRep, PRIMARY KEY (cust_key))</pre>

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

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.

Partitioning the Customer table in the Contact example

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

```
CREATE PUBLICATION SalesRepData (
    TABLE SalesRep
    TABLE Customer SUBSCRIBE BY rep_key
)
```

Partitioning the Contact table in the Contact example

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

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

```
CREATE PUBLICATION SalesRepData (
  TABLE SalesRep
  TABLE Customer
  SUBSCRIBE BY rep_key
  TABLE Contact
  SUBSCRIBE BY (SELECT rep_key
    FROM Customer
    WHERE Contact.cust_key = Customer.cust_key )
)
```

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

↪ For an Adaptive Server Enterprise consolidated database, the solution is different. For more information, see "Partitioning tables that do not contain the subscription column" on page 168.

Territory realignment in the Contact example

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

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

↪ For information on the way in which Adaptive Server Anywhere and SQL Remote work together to handle this situation, see "Who gets what?" on page 107.

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 will 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**.

Use triggers to maintain Contacts

The solution is to use a trigger containing a special form of UPDATE statement, which does not make any change to the database tables, but which does make an entry in the transaction log. This log entry contains the before and after values of the subscription expression, and so is of the proper form for the Message Agent to replicate the rows properly.

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

Trigger definition

The trigger definition is as follows:

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

A special UPDATE statement for publications

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

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

Here is what the UPDATE statement clauses mean:

Notes on the trigger

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

- ◆ If the trigger uses the following syntax:

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

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

- ◆ If the trigger uses the alternate syntax:

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

The trigger can be a BEFORE or AFTER trigger.

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

In this case, the only subscriber to the row is the new sales representative. In "Sharing rows among several subscriptions" on page 133, we see cases where there are existing as well as new subscribers.

Information in the transaction log

Here we describe the information placed in the transaction log. Understanding this helps in designing efficient publications.

- ◆ Assume the following data:
 - ◆ SalesRep table

rep_key	name
rep1	Ann
rep2	Marc

◆ Customer table

cust_key	name	rep_key
cust1	Sybase	rep1
cust2	ASA	rep2

◆ Contact table

contact_key	name	cust_key
contact1	David	cust1
contact2	Stefanie	cust2

◆ Now apply the following territory realignment Update statement

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

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

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

SalesRepData - Publication Name
rep1 - BEFORE list
rep2 - AFTER list
UPDATE Customer
SET rep_key = 'rep2'
WHERE cust_key = 'cust1'
```

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

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

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

Sharing rows among several subscriptions

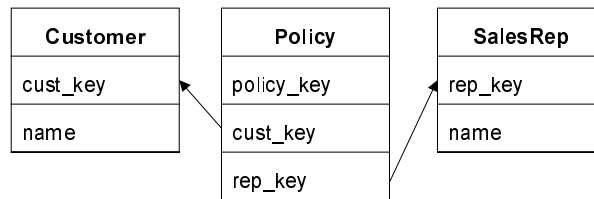
There are cases where a row may need to be included in several subscriptions. For example, we may have a many-to-many relationship. In this section, we use a case study to illustrate how to handle this situation.

The Policy example

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

Example database

Here is a simple database that illustrates the problem.



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

The tables in the database

The three tables are described in more detail as follows:

Table	Description
SalesRep	<p>All sales representatives that work for the company. The SalesRep table has the following columns:</p> <ul style="list-style-type: none">◆ rep_key An identifier for each sales representative. This is the primary key.◆ name The name of each sales representative. <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE SalesRep (rep_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, PRIMARY KEY (rep_key));</pre>
Customer	<p>All customers that do business with the company. The Customer table includes the following columns:</p> <ul style="list-style-type: none">◆ cust_key A primary key column containing an identifier for each customer◆ name A column containing the name of each customer <p>The SQL statement creating this table is as follows:</p> <pre>CREATE TABLE Customer (cust_key CHAR(12) NOT NULL, name CHAR(40) NOT NULL, PRIMARY KEY (cust_key));</pre>

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

- Replication goals**
- The goals of the replication design are to provide each sales representative with the following information:
- ◆ The entire **SalesRep** table.
 - ◆ Those rows from the **Policy** table that include sales relationships involving the sales rep subscribed to the data.
 - ◆ Those rows from the **Customer** table listing customers that deal with the sales rep subscribed to the data.
- New problems**
- The many-to-many relationship between customers and sales representatives introduces new challenges in maintaining a proper sharing of information:
- ◆ We have a table (in this case the **Customer** table) that has no reference to the sales representative value that is used in the subscriptions to partition the data.
- Again, this problem is addressed by using a subquery in the publication.
- ◆ Each row in the **Customer** table may be related to many rows in the **SalesRep** table, and shared with many sales representatives databases.

Put another way, the rows of the **Contact** table in "Partitioning tables that do not contain the subscription expression" on page 125 were partitioned into disjoint sets by the publication. In the present example there are overlapping subscriptions.

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

The publication

A single publication provides the basis for the data sharing:

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

The subscription statements are exactly as in the previous example.

How the publication works

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

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

```
...  
    TABLE SalesRep,  
...
```

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

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

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

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

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


```

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

```

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

Multiple-valued subqueries in publications

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

Territory realignment with a many-to-many relationship

The problem of territory realignment (reassigning rows among subscribers) requires special attention, just as in the section "Territory realignment in the Contact example" on page 128.

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

How customers are transferred

In this example, we require that a customer transfer be achieved by deleting and inserting rows in the **Policy** table.

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

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

Using Triggers to solve the problem

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

**A BEFORE
INSERT trigger**

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

```
CREATE TRIGGER InsPolicy
BEFORE INSERT ON Policy
REFERENCING NEW AS NewRow
FOR EACH ROW
BEGIN
    UPDATE Customer
    PUBLICATION SalesRepData
    SUBSCRIBE BY (
        SELECT rep_key
        FROM Policy
        WHERE cust_key = NewRow.cust_key
        UNION ALL
        SELECT NewRow.rep_key
    )
    WHERE cust_key = NewRow.cust_key;
END;
```

**A BEFORE
DELETE trigger**

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

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

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

**Multiple-valued
subqueries**

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

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

...

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

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

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

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

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

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

Notes

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

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

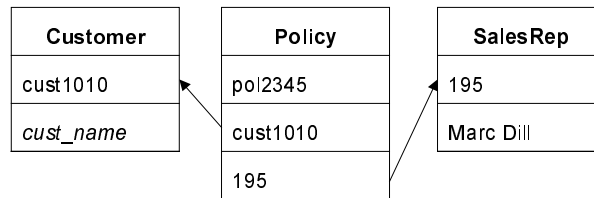
When the `Subscribe_by_remote` option is ON, operations from remote databases on rows with a subscribe by value of NULL or an empty string will assume the remote user is subscribed to the row. By default, the `Subscribe_by_remote` option is set to ON. In most cases, this setting is the desired setting.

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

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

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

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.



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

Later, when the Message Agent scans the log, it builds a list of subscribers from the subscription expression, and Marc Dill is not on the list, as the row in the Policy table assigning the customer to him has not yet been applied. If `Subscribe_by_remote` were set to OFF, the result would be that the new 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 the row belongs to the Sales Rep that inserted it, the `INSERT` is not replicated back to Marc Dill, and the replication system is intact.

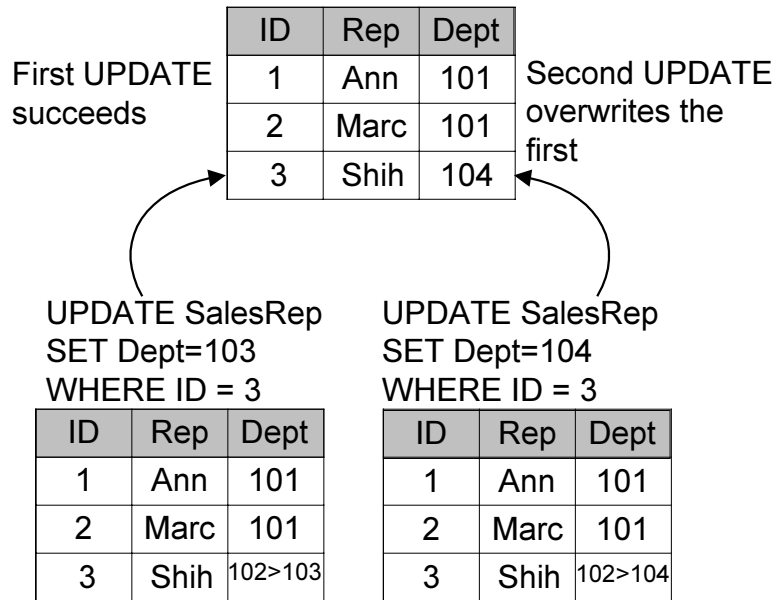
If `Subscribe_by_remote` is set to `OFF`, you must ensure that the Policy row is inserted before the Customer row, with the referential integrity violation avoided by postponing checking to the end of the transaction.

Managing conflicts

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

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

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



Default conflict resolution

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

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

Conflict resolution does not apply to primary key updates

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

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

How SQL Remote handles conflicts

When a conflict is detected

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

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


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

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

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

Conflict resolution can take several forms. For example:

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

 The method of conflict resolution is different at an Adaptive Server Enterprise consolidated database. For more information, see "How SQL Remote handles conflicts" on page 183.

Implementing conflict resolution

This section describes what you need to do to implement custom conflict resolution in SQL Remote for Adaptive Server Anywhere. The concepts are the same in SQL Remote for Adaptive Server Enterprise, but the implementation is different.

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

- 1 Any conflict resolution triggers defined for the operation are fired.
- 2 The UPDATE takes place.
- 3 Any actions of the trigger, as well as the UPDATE, are replicated to all remote databases, including the sender of the message that triggered the conflict.

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

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

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

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

Using conflict resolution triggers

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

UPDATE statements with a VERIFY clause

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

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

The VERIFY clause compares the values of specified columns to a set of expected values, which are the values that were present in the publisher database when the UPDATE statement was applied there.

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

Conflict resolution trigger syntax

The syntax for a RESOLVE UPDATE trigger is as follows:

```
CREATE TRIGGER trigger-name
RESOLVE UPDATE
OF column-name ON table-name
[ REFERENCING [ OLD AS old_val ]
  [ NEW AS new_val ]
  [ REMOTE AS remote_val ] ]
FOR EACH ROW
BEGIN
...
END
```

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

Using the VERIFY_ALL_COLUMNS option

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

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

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

Using the
CURRENT
REMOTE USER
special constant

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

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

Conflict resolution examples

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

Resolving date conflicts

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

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

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

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

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

Implementing the
solution

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

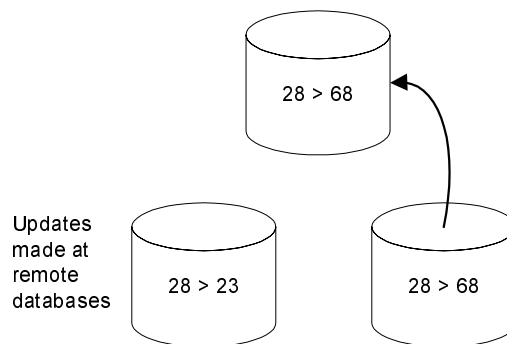
```
CREATE TRIGGER contact_date RESOLVE UPDATE
ON contact
REFERENCING OLD AS old_name
NEW AS new_name
FOR EACH ROW
BEGIN
    IF new_name.contact_date <
        old_name.contact_date THEN
        SET new_name.contact_date
            = old_name.contact_date
    END IF
END
```

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

Resolving inventory conflicts

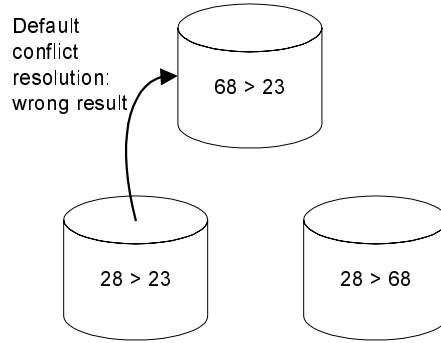
Consider a warehouse system for a manufacturer of sporting goods. There is a table of product information, with a **quantity** column holding the number of each product left in stock. An update to this column will typically deplete the quantity in stock or, if a new shipment is brought in, add to it.

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

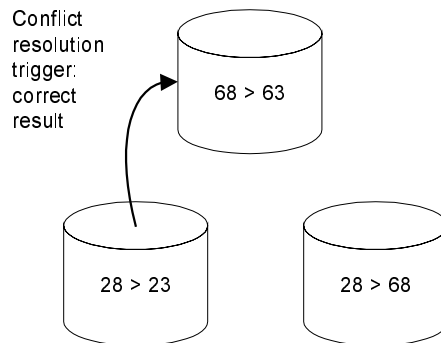


The warehouse entry gets added to the database: the **quantity** column now shows there are 68 small tank-top tee shirts in stock. When the update from the sales representative arrives, it causes a conflict—Adaptive Server Anywhere detects that the update is from 28 to 23, but that the current value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

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



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



Implementing the solution

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

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

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

Consistency is maintained at the remote database as follows:

- 1 The original remote UPDATE changed the value from 28 to 23.
- 2 The warehouse's entry is replicated to the remote database, but fails as the old value is not what was expected.
- 3 The changes made by the RESOLVE UPDATE trigger are replicated to the remote database.

Reporting conflicts

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

Designing to avoid referential integrity errors

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

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

The **sales** publication described in "Publishing a set of tables" on page 117 includes the **sales_order** table:

```
CREATE PUBLICATION pub_sales (  
    TABLE customer,  
    TABLE sales_order,  
    TABLE sales_order_items,  
    TABLE product  
)
```

The **sales_order** table has a foreign key to the **employee** table. The id of the sales rep is a foreign key in the **sales_order** table referencing the primary key of the **employee** table. However, the **employee** table is not included in the publication.

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

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

Designing triggers to avoid errors

Actions performed by triggers are not replicated: triggers that exist at one database in a SQL Remote setup are assumed by the replication procedure to exist at other databases in the setup. When an action that fires a trigger at the consolidated database is replicated at the replicate site, the trigger is automatically fired. By default, the database extraction utility extracts the trigger definitions, so that they are in place at the remote database also.

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

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

- ◆ Have actions of the trigger be conditional on a SELECT statement which determines if rows exist.

The RESOLVE UPDATE trigger is a special trigger type for the resolution of UPDATE conflicts, and is discussed in the section "Conflict resolution examples" on page 146. The actions of RESOLVE UPDATE triggers are replicated to remote databases, including the database that caused the conflict.

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 CHAR(12) NOT NULL,  
    PRIMARY KEY (table_name, value),  
);
```

The columns of this table have the following meanings:

Column	Description
table_name	Holds the names of tables for which primary key pools must be maintained. In our simple example, if new sales representatives were to be added only at the consolidated database, only the 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	An identifier for the recipient. 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);
```

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.

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

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

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

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

...

The subscription argument is the location identifier.

In some circumstances it makes sense to add the KeyPool table to an existing publication and use the same argument to subscribe to each publication. 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()
BEGIN
    FOR EachTable AS TableCursor
    CURSOR FOR
        SELECT table_name
        AS CurrTable, max(value) as MaxValue
        FROM KeyPool
        GROUP BY table_name
    DO
        FOR EachRep AS RepCursor
        CURSOR FOR
            SELECT location
            AS CurrRep, count(*) as NumValues
            FROM KeyPool
            WHERE table_name = CurrTable
            GROUP BY location
        DO
            // make sure there are 100 values.
            // Fit the top-up value to your
            // requirements
            WHILE NumValues < 100 LOOP
                SET MaxValue = MaxValue + 1;
                SET NumValues = NumValues + 1;
                INSERT INTO KeyPool
                (table_name, location, value)
                VALUES
                (CurrTable, CurrRep, MaxValue);
            END LOOP;
        END FOR;
    END FOR;
END;
```

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

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

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

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

Cannot use a trigger to replenish the key pool

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

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

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

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

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

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

- ◆ **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(
    IN customer_name CHAR( 40 ) )
BEGIN
    DECLARE new_cust_key INTEGER ;
    CALL NewKey( 'Customer', new_cust_key );
    INSERT
    INTO Customer (
        cust_key,
        name,
        location
    )
    VALUES (
        'Customer ' ||
        CONVERT (CHAR(3), new_cust_key),
        customer_name,
        CURRENT PUBLISHER
    );
END
```

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

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

Subscriptions with no subscription expression	<p>To subscribe to a publication, each subscriber must be granted REMOTE permissions and a subscription must also be created for that user. The details of the subscription are different depending on whether or not the publication uses a subscription expression.</p>
	<p>To subscribe a user to a publication, if that publication has no subscription expression, you need the following information:</p> <ul style="list-style-type: none"> ◆ 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.
	<p>The following statement creates a subscription for a user ID SamS to the pub_orders_samuel_singer publication, which was created using a WHERE clause:</p>
	<pre>CREATE SUBSCRIPTION TO pub_orders_samuel_singer FOR SamS</pre>
Subscriptions with a subscription expression	<p>To subscribe a user to a publication, if that publication does have a subscription expression, you need the following information:</p> <ul style="list-style-type: none"> ◆ User ID The user who is being subscribed to the publication. This user must have been granted remote permissions. ◆ Publication name The name of the publication to which the user is being subscribed. ◆ Subscription value The value that is to be tested against the subscription expression of the publication. For example, if a publication has the name of a column containing an employee ID as a subscription expression, the value of the employee ID of the subscribing user must be provided in the subscription. The subscription value is always a string.
	<p>The following statement creates a subscription for Samuel Singer (user ID SamS, employee ID 856) to the pub_orders publication, defined with a subscription expression sales_rep, requesting the rows for Samuel Singer's own sales:</p>
	<pre>CREATE SUBSCRIPTION TO pub_orders ('856') FOR SamS</pre>
Starting a subscription	<p>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.</p>

