solid.

www.solidDB.com

solidDB
SmartFlow Data Replication Guide

Version 6.0 | June 2007

solidDB 6.0

solidDB SmartFlow Data Replication Guide

Document number: SSG60
Product version: 06.00.1011
Date: 12.06.2007

restrictions: 1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required. 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software. 3. This notice may not be removed or altered from any source distribution.

This product is assigned the U.S. Export Control Classification Number ECCN=5D992a.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Welcome

solidDB SmartFlow Replication is a capability to replicate the data across disparate computer nodes in a network to meet the needs of your applications. With SmartFlow, the data can be as close to the application as implied by the latency, performance, or availability requirements. The core of SmartFlow is asynchronous replication based on the publish/subscribe model. In the solidDB context, the replication type used in SmartFlow is called "synchronization",

1.1 About This Guide

solidDB SmartFlow Data Replication Guide introduces you to synchronization concepts and architecture and describes how to set up, use, and administer a solidDB using SmartFlow.

1.1.1 Organization

This guide contains the following chapters:

- Chapter 2, *Introduction to Data Synchronization Using SmartFlow*, gives you an overview of Solid Data Synchronization.
- Chapter 3, *solidDB Data Synchronization Architecture*, familiarizes you with the SmartFlow architecture.
- Chapter 4, *Getting Started With Data Synchronization*, provides a quick tour (using sample scripts) for setting up synchronization with basic SmartFlow commands — proprietary extensions to Solid SQL.
- Chapter 5, *Planning And Designing For SmartFlow Applications*, describes the issues and considerations for designing a distributed system using SmartFlow.
- Chapter 6, *Implementing a solidDB SmartFlow Application*, describes the basic tasks required to implement synchronization.
- Chapter 7, *Updating and Maintaining the Schema of a Distributed System*, describes how to change table and publication definitions without necessarily requiring each replica to get a full refresh the next time that it refreshes.
- Chapter 8, *Administering solidDB with SmartFlow*, describes how to maintain your solidDB with SmartFlow technology. The administration tasks covered in this chapter include managing synchronization errors, and tips on backing up masters and replicas.
- Chapter 9, *Performance Monitoring and Tuning*, discusses techniques that you can use to improve the performance of the SmartFlow feature of solidDB.

- Appendix A, *Bulletin Board Parameters*, explains the "standard" bulletin board parameters that are used in Intelligent Transactions. These parameters help the replica and master communicate about what to do if the data cannot be processed as the replica requested.

- Appendix B, *Synchronization Events*, lists synchronization-related events that solidDB automatically generates, and that you can use to monitor the progress of messages that are exchanged between servers as they synchronize their data.

Glossary

Glossary provides definitions of Solid Data Synchronization terminology.

1.1.2 Audience

This guide assumes the reader has general DBMS knowledge and a familiarity with SQL. It also assumes that you are already knowledgeable about solidDB. You may want to read solidDB Getting Started Guide and solidDB Administration Guide before reading this manual.

1.2 Conventions

1.2.1 About solidDB

solidDB from Solid Information Technology (Solid) represents a family of advanced database solutions for mission-critical applications.

This documentation assumes that all options of solidDB are licensed for use. In some cases, however, a customer may choose not to license certain options. These include in-memory engine, disk-based engine, CarrierGrade Option (also known as "HotStandby" in previous releases), and SmartFlow Option. Please refer to your organization's contract with Solid, or contact your Solid account representative.

1.2.2 Typographic Conventions

This manual uses the following typographic conventions:

Table 1.1. Typographic Conventions

Format	Used for
Database table	This font is used for all ordinary text.
NOT NULL	Uppercase letters on this font indicate SQL keywords and macro names.

Format	Used for
`solid.ini`	These fonts indicate file names and path expressions.
`SET SYNC MASTER YES;` `COMMIT WORK;`	This font is used for program code and program output. Example SQL statements also use this font.
run.sh	This font is used for sample command lines.
`TRIG_COUNT()`	This font is used for function names.
`java.sql.Connection`	This font is used for interface names.
`LockHashSize`	This font is used for parameter names, function arguments, and Windows registry entries.
`argument`	Words emphasised like this indicate information that the user or the application must provide.
solidDB Administration Guide	This style is used for references to other documents, or chapters in the same document. New terms and emphasised issues are also written like this.
File path presentation	File paths are presented in the Unix format. The slash (/) character represents the installation root directory.
Operating systems	If documentation contains differences between operating systems, the Unix format is mentioned first. The Microsoft Windows format is mentioned in parentheses after the Unix format. Other operating systems are separately mentioned.

1.2.3 Syntax Notation

This manual uses the following syntax notation conventions:

Table 1.2. Syntax Notation Conventions

Format	Used for
`INSERT INTO` *`table_name`*	Syntax descriptions are on this font. Replaceable sections are on *`this`* font.
`solid.ini`	This font indicates file names and path expressions.

Format	Used for
[ ]	Square brackets indicate optional items; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{ }	Curly brackets delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of three dots indicates continuation of previous lines of code.

1.3 Solid Documentation

Below is a complete list of documents available for solidDB. Solid documentation is distributed in an electronic format, usually PDF files and web pages.

Electronic Documentation

- *Release Notes*. This file contains installation instructions and the most up-to-date information about the specific product version. This file (`releasenotes.txt`) is copied onto your system when you install the software.
- *solidDB Getting Started Guide*. This manual gives you an introduction to the solidDB.
- *solidDB SQL Guide*. This manual describes the SQL commands that solidDB supports. This manual also describes some of the system tables, system views, system stored procedures, etc. that the engine makes available to you. This manual contains some basic tutorial material on SQL for those readers who are not already familiar with SQL. Note that some specialized material is covered in other manuals. For example, the Solid "administrative commands" related to the High Availability (HotStandby) Option are described in the *solidDB High Availability User Guide*, not the solidDB SQL Guide.
- *solidDB Administration Guide*. This guide describes administrative procedures for solidDB servers. This manual includes configuration information. Note that some administrative commands use an SQL-like syntax and are documented in the *solidDB SQL Guide*.

- *solidDB Programmer Guide*. This guide explains in detail how to use features such as Solid Stored Procedure Language, triggers, events, and sequences. It also describes the interfaces (APIs and drivers) available for accessing solidDB and how to use them with a solidDB database.

- *solidDB In-Memory Database User Guide*. This manual describes how to use the in-memory database of solidDB In-memory Engine.

- *solidDB SmartFlow Data Replication Guide*. This guide describes how to use the Solid SmartFlow technology to synchronize data across multiple database servers.

- *solidDB AcceleratorLib User Guide*. Linking the client application directly to the server improves performance by eliminating network communication overhead. This guide describes how to use the AcceleratorLib library, a database engine library that can be linked directly to the client application.

  This manual also explains how to use two proprietary Application Programming Interfaces (APIs). The first API is the Solid SA interface, a low-level C-language interface that allows you to perform simple single-table operations (such as inserting a row in a table) quickly. The second API is SSC API, which allows your C-language program can control the behavior of the embedded (linked) database server

  This manual also explains how to set up a solidDB to run without a disk drive.

- *solidDB High Availability User Guide*. Solid CarrierGrade Option (formerly called the HotStandby Option) allows your system to maintain an identical copy of the database in a backup server or "secondary server". This secondary database server can continue working if the primary database server fails.

Chapter 2. Introduction to Data Synchronization Using SmartFlow

This chapter introduces you to solidDB SmartFlow data synchronization, which allows you to store, manage, and synchronize data across databases. This is useful in many different industries, such as media delivery networks and various telecommunications devices and applications.

SmartFlow Data Synchronization functionality has been implemented inside solidDB. It utilizes all the capabilities that solidDB provides, such as transactions and SQL to provide a rich set of data distribution features.

As a simple example, suppose that you manage an enterprise that has multiple branch offices. You might want to give every branch office or user a local copy of your database so that each user gets the fastest possible response time. But if each office or user has her own copy of the database, then there's a high risk of the databases becoming inconsistent over time, as various data is updated in numerous databases of the system. solidDB SmartFlow provides the flexibility and fast access of providing local copies of the database, while maintaining the data consistency of the data of the distributed system.

solidDB SmartFlow allows you to have a "master" database and many local copies (called "replicas"). SmartFlow allows you to synchronize data between master and replica(s) frequently. You may even disconnect local copies of the database (in your laptop or PDA, for example) from the network and then reconnect them later; each time you reconnect, you can do bi-directional synchronization, sending the latest version of your local data to the master server, and downloading the latest data from the master to your local replica server.

Another advantage of Solid's master/replica synchronization technology is that each user or office can be given only the "slice" of information that is relevant to their work. This reduces the bandwidth required to synchronize databases across a network. This can also be used to increase data security by limiting access without requiring all data to be stored in a single secure location.

The SmartFlow synchronization is of periodical nature. Therefore, there is a possibility of conflicting updates. For example, a technician in the field might update a customer's address on a local replica, while a customer service person at the main office might update the customer's address in the master database. When the replica is reconnected to the network, whose data takes precedence if the data doesn't match? You can write application-specific logic to resolve such conflicts if they occur; solidDB SmartFlow's features support writing such conflict-resolution logic.

2.1 About solidDB SmartFlow

solidDB SmartFlow has a new approach to data synchronization. It applies concepts that are fundamentally different from the data replication solutions of previous generation.

The fundamental difference is an architecture that has been completely built into the core database engine. It uses extensions to the relational data management functions to enable building data synchronization functionality that meets the requirements of the applications. This new model recognizes what traditional replication solutions do not; that data synchronization contains application-specific aspects. For proper functionality, synchronization functionality must be tailorable to meet the requirements of the business application. Applications, for example, need to accurately detect conflicts over time based on their own unique business rules and logic and resolve these conflicts in a multi-database system.

With the solidDB SmartFlow solution, application developers now have a set of data management functions that allow them to include robust data synchronization functionality into the applications with minimal effort.

A "master" database contains the master copy of the data. One or more replica databases contain full or partial copies of the master's data. A replica database, like any other database, may contain multiple tables. Some of those tables may contain only replicated data (copied from the master), some may contain local-only data (not copied from the master), and some may contain a mix of replicated data and local-only data.

Replicas may submit updates to the master server, which then verifies the updates according to rules set by the application programmers. The verified data is then "published" and made available to all replicas.

2.1.1 solidDB SmartFlow Features

solidDB SmartFlow provides the following features:

- System-wide information sharing

  With solidDB SmartFlow, each server in the system may have its own local copy of the data that it needs. There is no need to provide users with on-line access to central data management resources. In addition, each replica database of a SmartFlow system can serve a specific purpose. For example, one replica could be dedicated to a decision support or reporting application, while another could be dedicated to an on-line transaction processing application.

- Data integrity

  In a multi-database system, where updates can occur in multiple databases, maintaining data integrity poses challenges. solidDB SmartFlow's own transaction management architecture addresses data integrity issues by allowing the transactions's application developer to build transaction validation capabilities into the transaction itself. This ensures that information is accurate based on the application's own rules and logic.

- High performance and flexibility

  SmartFlow architecture lets users tailor the synchronization process to maximize performance. For example, large amounts of data can be transferred over the network when the available bandwidth is optimal. Sim-

ilarly, the propagation of only high-priority transactions, which reflect urgent data, can be specified during rush hours.

2.2 Purpose

The solidDB SmartFlow option allows users to synchronize data across multiple computers. Each computer will have a database managed by a solidDB with SmartFlow (tm) Option. A SmartFlow-enabled solidDB can send data to another SmartFlow-enabled solidDB, and the receiving solidDB can store the data in its database.

There are several different data distribution models that you may use. For example, you can use a geographic or conceptual model to control how data is distributed.

- You might want every computer to have an exact copy of all of the data. For example, you might want every repair person out in the field to have a complete, up-to-date copy of the repair parts lists and prices.

- Alternatively, you might want to "slice" the data up, distributing different pieces of the whole to different computers. This distribution might be based on geographic responsibility — for example, the computer at headquarters might have a copy of all of the data (e.g. all the customer accounts), while each local branch might keep only a copy of the data that applies to it. Or you might want to have a rack of telecommunications line cards, each of which is responsible for handling particular connections or addresses. A single "master" computer keeps a complete set of connections and assigns them to individual line cards.

You may also control how data is distributed over time.

- For example, you might choose to update data at regular intervals, such as once a week, once a day, once a minute, or once every 5 seconds.

- Alternatively, you might choose to propagate updated data as soon as the change occurs, rather than according to a clock or calendar.

- You might want to use a mix — propagating certain types of data (updated email addresses, for example) as soon as they are updated, while propagating other types of data (such as billing summaries) once a month.

SmartFlow synchronization technology allows you to decide how to divide up the data, and when that data should be sent to another SmartFlow-enabled solidDB.

Master/Replica Model

Solid's synchronization technology relies on a "master/replica" model. A single computer keeps a "master" or "official" copy of the data. All other computers ("replicas") in the system may also have a copy of some or all of the master data. When the replica changes data, that data is "unofficial" until it has been sent to, and accepted by, the master. Once the data has been accepted on the master, the replica that sent that data (and

other replicas, as well) may request a copy of the new official data. In this way, replicas may temporarily become out of sync with the master, but such differences can be corrected quickly.

Solid's solution allows a virtually unlimited number of replicas to have access to data. Each replica may read and write the data, as long as users know that local writes must be accepted at the master before they are official. Replicas do not need to be connected to the network 100% of the time. A replica may be stored on a PDA or other computer that is only connected to the network part of the time. This allows replicas to operate for long periods independently of the master. This provides flexibility. It also means that your entire system isn't disabled just because the master database server was shut down for maintenance or to correct a problem. Distributed systems based on Solid's synchronization technology are inherently robust with respect to isolated failures.

A replica database may hold not only copies of some or all master data, but also "local" data that is not shared with the master. A single computer may hold a mix of local and shared data.

2.2.1 Typical Applications

Below are examples of how Solid's synchronization technology may be used:

A bank might keep all of its account information in a central database. Branch offices might get subset of the data for customers who use that particular branch.

Mobile sales people or repair people might keep copies of a subset of customer information or product information on their mobile computing devices, while a central office keeps a complete set of the data.

A rack of telecommunications equipment may hold many line cards, each with its own memory. One card may act as the "master" assigning connections to specific line cards. Other line cards may act as replicas, each handling a subset of the connections. Each card might be responsible for a particular range of network addresses, for example.

A content provider might keep a complete set of files (movies, music, etc.), which could be downloaded to individual consumers for their use. Note that this type of application is not limited to "1-to-many" relationships. Customers could have relationships with multiple content providers, so that there is a many-to-many relationship between content providers and customers.

2.3 How Data Is Exchanged

solidDB allows bi-directional data flow. Masters may send data to replicas, and replicas may send data to masters.

Sending data from masters to replicas is done using a model that is called "publish and subscribe" because it is based loosely on the idea that the master "publishes" data that replicas may "subscribe" to.

When we send data from a replica to a master, we call that "propagating" the data.

2.3.1 Sending Data from a Master to a Replica

A master database may share as much or as little data as it wants. A user on the master creates "publications", which are sets of data that replicas may request. A publication is similar to a view — it is a set of data defined by stating the tables where the data comes from, and the portions of the data that should be included.

Once these publications are created, replicas may "register" for those publications.

Once a replica has registered for a publication, the replica may get refreshes from that publication. The replica may use parameters to request only a subset of the publication. For example, the master might decide to publish billing summaries for all customers. Replicas (at branch offices) might want only the billing summaries of the customers that branch is responsible for. To specify which billing summaries the replica wants, the replica may provide input parameters that act as search criteria for the data of the publication.

Thus the outline of steps is:

1. Master creates a publication with specified data (e.g. billing summaries of all customers).

2. Replica registers for that publication.

3. Replica requests a refresh from that publication to get all or part of the data.

Note that the model of "publish and subscribe" is similar to, but not identical to, the way magazines are published and subscribed to. A large publishing company might sell many different monthly magazines — perhaps one on SQL, one on C, and one on Java. The publisher decides what information to put into each issue. You do not control the content. You as a subscriber register to receive one or more of those publications. For example, you might send in your money to subscribe to an SQL magazine.

A key difference between the world of magazine publishing and the world of SmartFlow synchronization is that in solidDB synchronization, the recipient, not the publisher, decides when to get new data. In magazine publishing, the magazine company decides when to send the magazine. If you subscribe to a monthly computer magazine, the publisher sends you a magazine each month, whether or not you have time to read it. In the world of SmartFlow synchronization, however, it is you the subscriber who decide when to request new data. In this way, solidDB synchronization is similar to publishing on the web rather than publishing on paper. With web publishing, you pay the publisher, the publisher puts data on the web site whenever the publisher wants, and you visit that web site whenever you want.

We use the word "push" to describe the situation where the publisher decides when to send data to the recipient, and we use the word "pull" to describe the situation where the recipient decides when to get data from the publisher. In the world of web publishing (and solidDB synchronization), the reader "pulls" the data.

In some applications, it is important for the master (the "publisher") to send data as soon as it is updated. Solid's synchronization technology provides a Sync Pull Notify feature that enables a master to notify replicas when there is new data that might be worth subscribing to. solidDB's Sync Pull Notify functionality provides the equivalent of "push" technology, but with more flexibility.

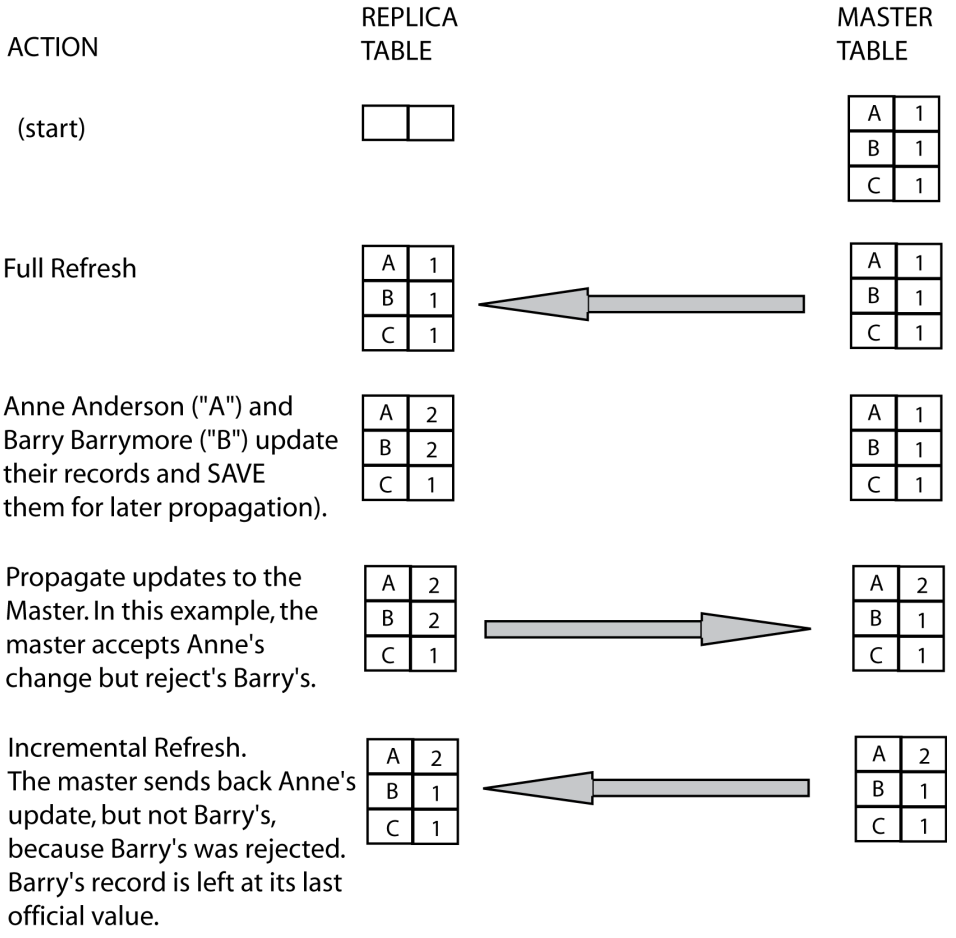
solidDB synchronization, like magazine publishing, allows the customer (the "replica") to subscribe to more than one master database (more than one magazine). If you want to subscribe to three different magazines, or three different master databases, you may do so. Note, however, that data from each master must be stored in a separate catalog (logical database) in the replica. (One database catalog may contain data from only one master database. However, a single physical database may contain multiple database catalogs. Hence, a replica database server may contain data from multiple master databases. One server may also contain both master and replica database catalogs. These topics are discussed in more detail later in this manual.)

solidDB synchronization is like magazine publishing and subscribing in another way. With solidDB synchronization, a subscriber may request a "full" or "incremental" download (or refresh) of information. An incremental refresh contains just the changes since the most recent prior refresh — analogous to receiving the most recent issue of a magazine, or seeing only the new articles on a web site. Alternatively, a solidDB subscriber may request a full refresh — which is analogous to ordering the complete set of back issues for a magazine, or looking at the entire current web site. Not surprisingly, replicas in a solidDB system always start with a full download; after that, they may request only incremental refreshes if they wish. (Note: although the term "refresh" implies that it is an update of previous data, we use the term "refresh" loosely to refer to the initial download as well as subsequent downloads.)

Data sent from the replica to the master is not automatically accepted at the master. solidDB's "Intelligent Transaction" capability allows the Master to reject or modify data to make sure that only valid data is stored in the master. This is explained in more detail in Section 3.2.3, “Intelligent Transaction”.

The illustration below shows an overview of the propagate and refresh process between a master and a single replica. This illustration includes an example of a record that was accepted by the master and another record that was rejected by the master (e.g. because the update was invalid according to business rules enforced on the master).

Figure 2.1. Propagate And Refresh



Later in this manual, we explain in more detail what happens during the Incremental Refresh. Specifically, we explain the process by which Barry's pre-updated record is restored to the last "official" value.

A solidDB system can be configured to automatically keep track of which data is "new" for each replica; users themselves do not need to keep track of anything to request only an incremental update. Furthermore, each replica's need for incremental data is tracked independently. A replica that hasn't refreshed for a week

will receive the most recent week's worth of changes. A replica that refreshed an hour ago will be sent only the most recent hour's worth of changes.

2.3.2 Sending Data from a Replica to Its Master

When data is sent from a replica to its master, we say that the data is "propagated" to the master. As with REFRESH operations, it is the replica that decides when and how much data should be sent.

When data is propagated from a replica to a master, the propagation is done by composing a "message" that is sent from the replica to the master. This message contains SQL statements rather than rows of data. A single message may contain multiple statements, and in fact may contain multiple transactions. (There is no way to send a fragment of a transaction to the master; the message must commit the SQL statements that it sends. You may divide a message into multiple transactions, but you cannot divide a transaction into multiple messages.)

Although there is almost no restriction on the SQL statements that you may use, as a practical matter the statements that you send to the master are typically the same as the statements that you executed on the replica. In other words, instead of sending raw data to the master, you send a series of statements that perform the same operations on the master as you performed on the replica. For example, if you performed the following steps on the replica:

```
INSERT INTO employees (id, name) VALUES (12, 'Michelle Uhuru');
UPDATE employees SET department = 'Telecommunications' WHERE id = 12;
```

then you would send the same commands to the master so that you would repeat the same steps on the master.

Caution

Be careful when executing statements that may affect more records on the master than they did on the replica. For example, a command that uses a unique employee ID in the WHERE clause is likely to have the same effect on the master as on the replica, but a command that has a "broad" WHERE clause may affect more records on the master than on the replica, if the replica has only a subset of the data on the master.

To compose a message, you use

SAVE

statements. In pseudo-code, this looks similar to the following:

```
INSERT INTO employees (id, name) VALUES (12, 'Michelle Uhuru');
```

```
UPDATE employees SET department = 'Telecommunications' WHERE id = 12;
-- Save the statements for later propagation to the master
-- database.
SAVE INSERT INTO employees (id, name) VALUES (12, 'Michelle Uhuru');
SAVE UPDATE employees SET department = 'Telecommunications' WHERE id = 12;
COMMIT WORK;
```

(For more information, see Section 6.6.2, “Saving the Transaction for Later Propagation” and see the Syntax appendix of *solidDB SQL Guide*.)

Note that the propagated SQL statements may include stored procedure calls.

2.3.3 Accepting Propagated Data on the Master

When the replica propagates data to the master, the master is not required to accept that data. Within a master/replica system, only the master has the authority to declare data "official". If the master receives data that violates database rules (such as referential integrity constraints), or business rules (such as prohibiting customers from exceeding a certain credit limit), the master can reject or alter the data.

The master also has other options, such as modifying the data to bring it into conformance. For example, if a customer orders more widgets than are in stock, the master might modify the customer's order so that the customer is sent all the remaining available widgets. This way, the customer's order is not thrown out completely.

2.4 Summary

Solid's SmartFlow technology uses a master/replica model. The master has the "official" copy of the data, and replicas may subscribe to that data. Replicas may also change the local copy of the data and propagate the transaction to the master, but the master has the authority to alter or reject the data of the transaction to maintain the consistency of the master data.

Data is sent from the master to the replica when the replica requests a refresh of data from a publication. The replica composes a message to the master that contains the "REFRESH" command.

Data is sent from the replica to the master when the replica composes and sends a message where it propagates transactions to the master.

Bi-directional synchronization occurs when a message contains both propagated transactions and REFRESH requests.

The next few chapters will show the actual syntax used to perform the operations that send data back and forth between masters and replicas. Chapter 4, *Getting Started With Data Synchronization* includes a working example of synchronization, so that you can see each step involved.

Chapter 3. solidDB Data Synchronization Architecture

3.1 solidDB SmartFlow Architecture Concepts

The solidDB SmartFlow architecture is based on the multi-tier data redundancy model. The data is known as redundant when the same data exists in multiple databases in the same system. This means that multiple, possibly temporarily different versions of the same data item can co-exist.

3.1.1 Multi-tier Redundancy Model

The multi-tier data redundancy model has one top-level master database and multiple replica databases below it. The replicas are updateable but the replica data is always *tentative* until it has been committed to the master database. A replica can act as a master to some other replica below it in the hierarchy.

This model allows implementation of a bi-directional asynchronous data synchronization mechanism between databases in a way that fully addresses the database consistency and scalability issues of a multi-database system.

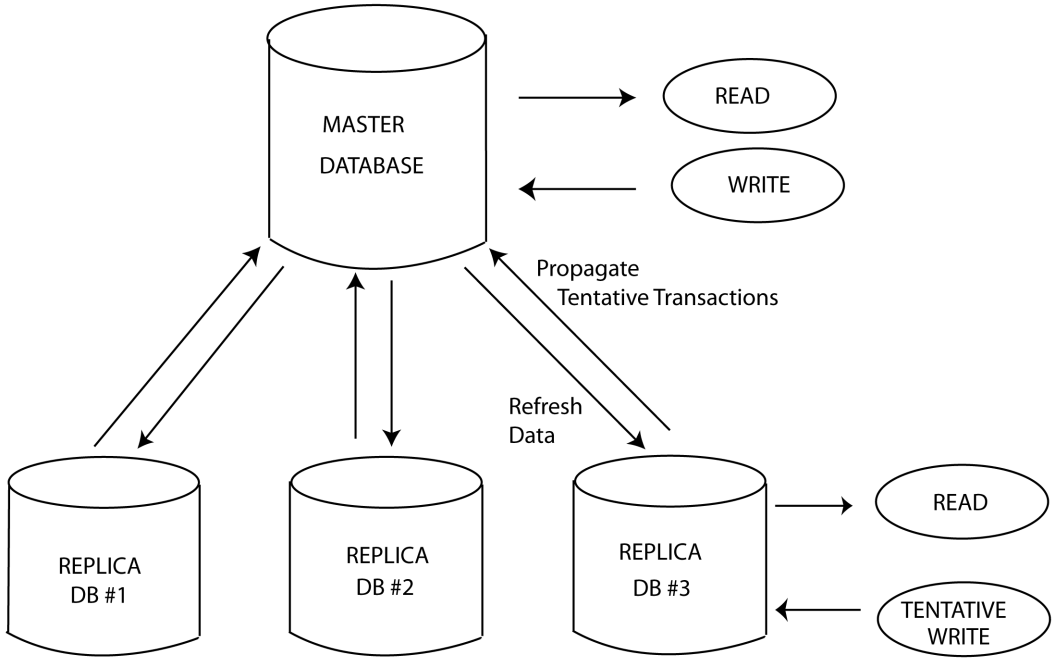
The multi-tier data redundancy model is based on the following principles:

- For each data item, there is one *master version* considered the official version of the data. Other copies of the item are *tentative versions*, that is, *replicas*.
- A replica can act as a master to other replicas below it. A replica that is also a master contains a subset (or a full set) of the data in the *master version* above it.
- Data in each database of the system is updateable.
- Modifications made directly to a master database are official.
- Transactions that are committed in a replica database are tentative until they have been successfully propagated to the master database and committed there.
- Replica databases are refreshed by sending changed data from the master database to replicas.

Basic Two-tier Architecture

The simplest implementation of this model is a two-tier synchronization architecture as shown in Figure 3.1, “Two-tier Data Redundancy Architecture”. Note that transactions are always sent to the master databases where they are committed. The changed master data is then sent to the replicas.

Figure 3.1. Two-tier Data Redundancy Architecture



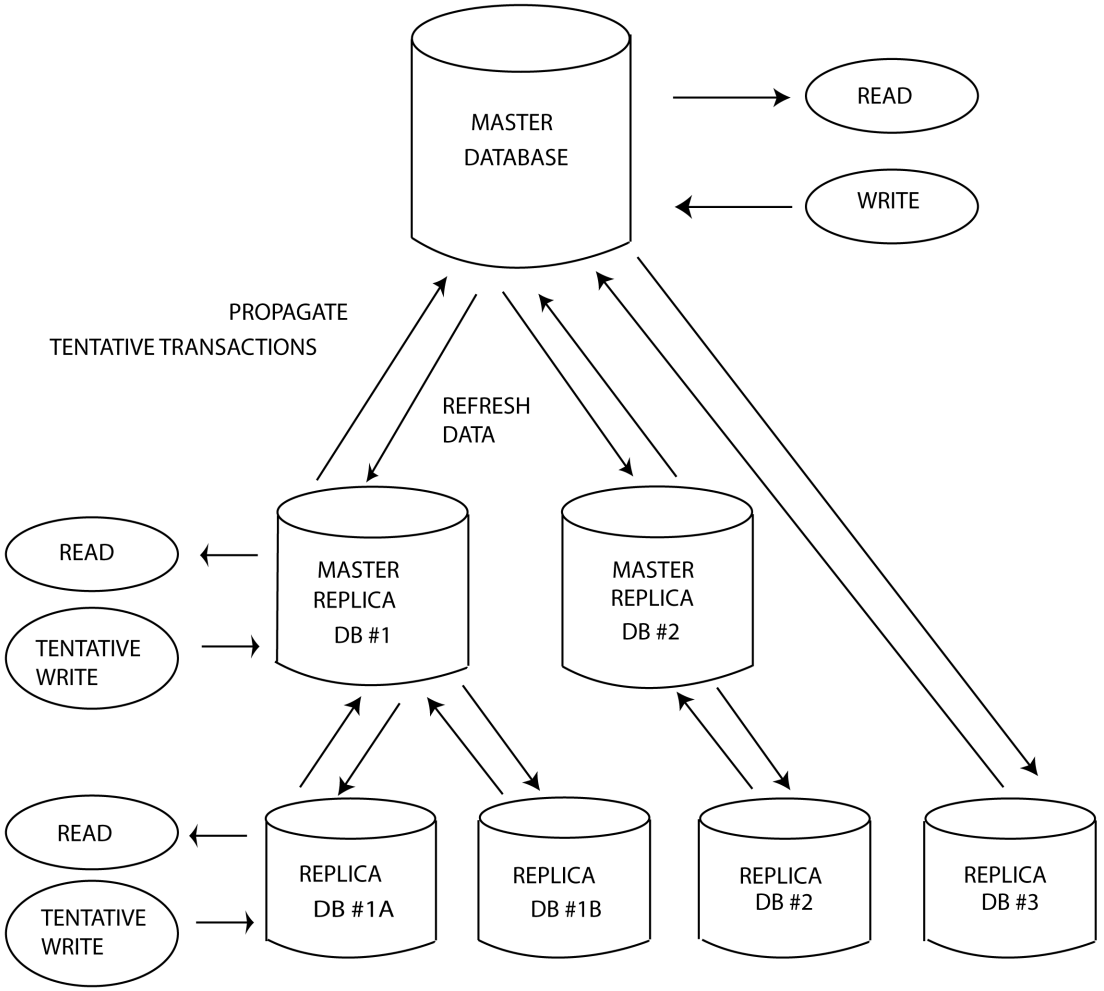
Advanced Multi-Tier Architecture

A more advanced implementation of this model enables information flow in more than two tiers through replicas that are also masters to some other replicas below them in the hierarchy. This architecture is often used in a scenario that requires the flow of system information to local areas and information from various local areas to specific end nodes.

Note

We will use the term "many-tiered" to describe systems that have three or more tiers, that is, systems that have nodes that are both master and replica. We will use the term "multi-tier" to describe systems that have two or more tiers.

Figure 3.2. Multi-tier Data Redundancy Architecture



solidDB SmartFlow Transaction Model

In both two-tiered and multi-tiered architectures, transactions that modify re-validated replica data are always tentative. A tentative transaction becomes official when it has been accepted in the master database. This means that the overall life cycle of a transaction is extended from the moment of replica commit to the moment of master commit. During this phase, activities can occur within an application that can invalidate a transaction that has already been committed by a replica. Thus all transactions that are to be propagated to the master database need to have embedded in them sufficient validation logic to ensure the integrity of the master database.

To address the data integrity issues of a synchronized multi-database system, solidDB SmartFlow introduces a new transaction model, the Solid Intelligent Transaction. This model provides a way for developers to implement transactions that always leave the master database in a consistent state. For more details, read Section 3.2.3, “Intelligent Transaction”.

Note that transactions committed in a replica database are propagated only to its master database. These transactions are not propagated to other replicas directly. Instead, the other replicas can request changed data from a master database by requesting refreshes from one or more publications. (A "publication" is a set of data on the master; clients may refresh from a publication to get updated data from the master.)

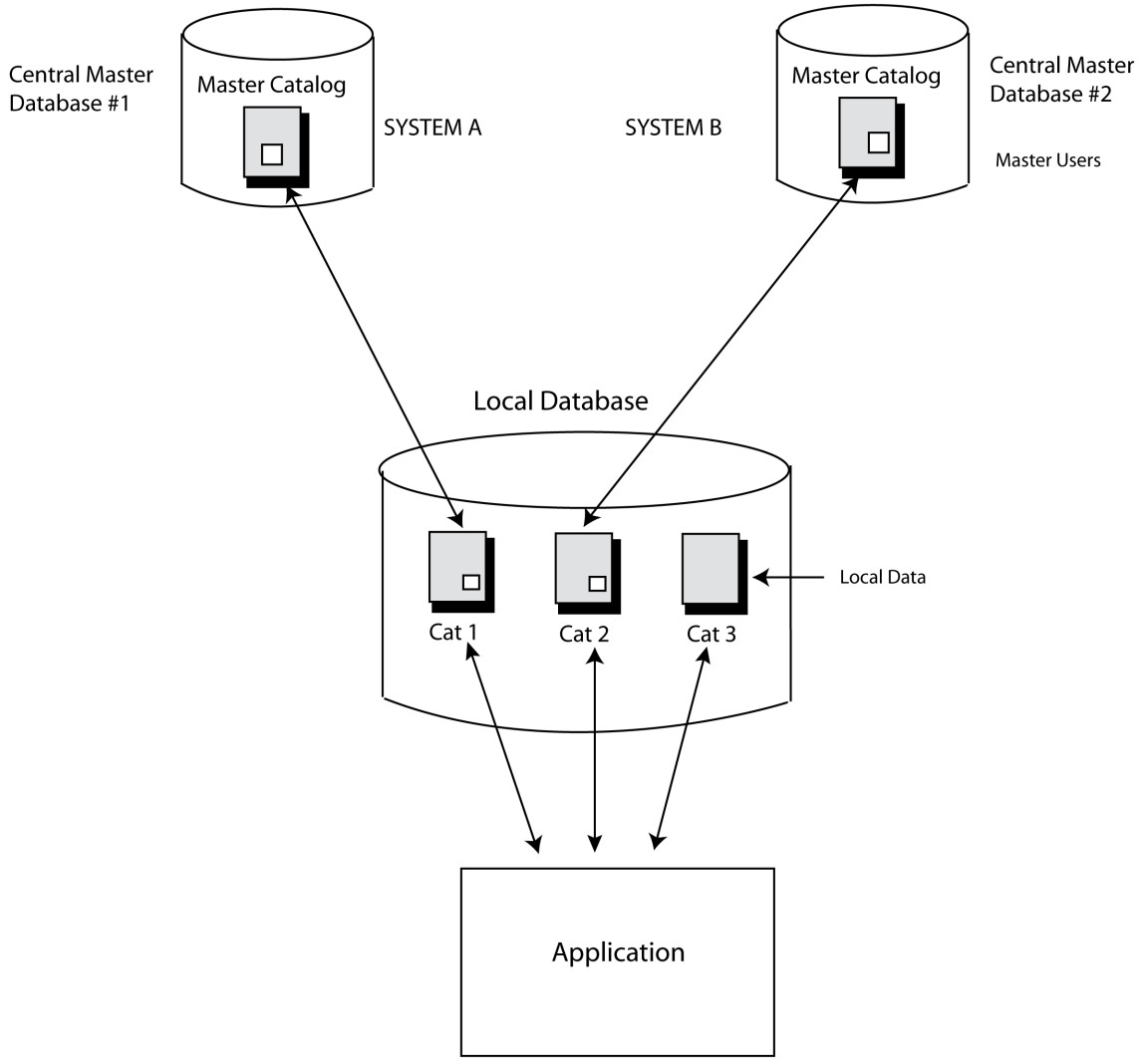
3.1.2 Multi-Master Synchronization Model

In today's distributed, networked environments, a system can consist of multiple applications. These applications may have their own databases. solidDB SmartFlow allows a physical database to contain data from multiple master databases. For instance, a local database can contain a replica from a billing host system and network configuration host system. Both two-tier and many-tier architectures are scalable to accommodate multiple master databases. The two-tier architecture shown in Figure 3.3, “Multi-Master Model” makes use of multi-master synchronization.

Figure 3.3, “Multi-Master Model” further illustrates the concept of multi-master synchronization on a table level. In this figure, note the following:

- A database server can contain replica databases from multiple masters.
- Systems A and B are separate and independent of each other.
- For each replica database, a database catalog is created in the database server.
- Replica A synchronizes with master A and replica B synchronizes with master B.
- A database server can also have one or multiple master databases or local-only databases in one or multiple catalogs, respectively.

Figure 3.3. Multi-Master Model



Multi-Master Features

solidDB SmartFlow's multi-master model:

- Allows replicas (through registration) to synchronize data with multiple solidDB SmartFlow masters.

- Keeps replica data from different masters separate using catalogs.
- Keeps local data separate from shared data.

Each of these features is described in the following sections.

Managing Replica Data in a Multi-Master Environment

A solidDB database may be divided into multiple, independent partitions or *catalogs*, and each catalog may be divided into multiple independent *schemas*. The ability to divide a database into catalogs is useful if your database contains multiple topics or is used by more than one application. Typically, each application would have its data stored in a separate database catalog.

In a multi-master environment, the ability to have multiple catalogs allows you to specify multiple databases (master or replica) for synchronization within one database server. For example, a solidDB server of an access router can have two catalogs, one for a replica of a configuration management database and the other catalog for a replica of a subscriber provisioning system.

The following section provides the concepts and background necessary to partition your solidDB database for synchronization.

solidDB Database, Catalogs, and Schemas

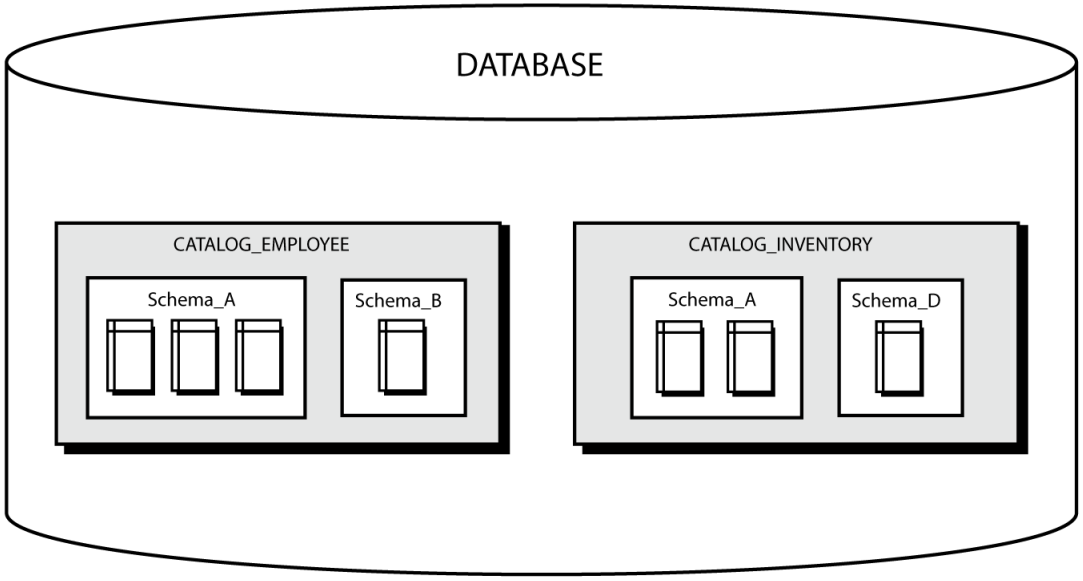
solidDB stores data in a file (or a set of files). These files are known as the *physical database*. The locations of these files are specified in the `solid.ini` configuration file. These files may be stored in one user-specified directory or in multiple user-specified directories.

Since each `solid.ini` configuration file specifies the location of one physical database, a single solidDB program can theoretically operate on different physical databases at different times, simply by starting the server with different `solid.ini` files. Although you may have more than one physical database from creating multiple physical files (and `solid.ini` files), one solidDB instance "sees" and "works with" only one physical database at a time. Using a single solidDB program with multiple databases at different times is uncommon.

A physical database file may contain more than one *logical database*. Each logical database is a complete, independent group of database objects, such as tables, indexes, procedures, triggers, etc. Each logical database is called a *catalog*. Note, however, that a solidDB catalog contains a wide variety of data objects, not just indexes (as in the traditional sense of a library card catalog, which serves to locate an item without containing the full contents of the item). For more details on the solidDB, read *solidDB Administration Guide*.

As a general rule, we will use the term "catalog" to refer to a "logical database", and we will use the term "database" to refer to a "physical database".

Figure 3.4. Database, Catalogs, and Schemas



Catalogs and the objects within them are arranged in a hierarchy. As shown in Figure 3.4, “Database, Catalogs, and Schemas”, a solidDB catalog can contain a database object known as a *schema*. Inside each catalog there may be multiple schemas. Each schema, in turn, may have multiple database objects, such as tables, views, sequences, etc. Catalogs are at the top of the hierarchy, and objects such as tables, etc. are at the bottom of the hierarchy.

Within a schema, the name of each object must be unique. For example, you cannot have two tables named "table1" inside the same schema. Similarly, within a catalog, the name of each schema must be unique. For example, you cannot have two schemas named "smith_schema" inside the same catalog.

Although object names within a schema must be unique, different objects in different schemas may have the same name. For example, it is legal to have both of the following:

```
smith_schema.table1
jones_schema.table1
```

Similarly, although schema names within a catalog must be unique, different catalogs may contains schemas with the same name. For example, the following is valid:

```
employee_catalog.smith_schema.table1
inventory_catalog.smith_schema.table1
```

When an object name has is preceded by the schema name and the catalog name, as shown above, the object name is said to be "fully qualified", that is, unambiguous. Database object names are qualified in DML statements as:

```
catalog_name.schema_name.database_object
```

or

```
catalog_name.user_id.database_object
```

Typically, each user in a catalog is allowed to have his or her own schema(s). For example, a user may own

```
smith_schema
```

in

```
employee_catalog
```

A single user may have schemas in more than one catalog. For example, as shown above, user "smith" might have schemas named "smith_schema" in multiple catalogs. As long as database objects within each catalog are specified by fully qualified names (i.e. names that include the object name, the schema name, and the catalog name), there is no confusion as to which database object is the one required. By organizing your data appropriately inside catalogs and schemas, you may "restrict" the "context" so that users or applications see only those database objects that are relevant to their task. You apply these concepts when creating catalogs and schemas for synchronization.

Catalogs and Synchronization

Catalogs allow you to implement multiple logical databases for synchronization. If your local physical database has replica data from multiple masters, then the replica needs one catalog for each master database of which

the replica has a copy (or a partial copy). Note that a catalog on a replica may contain not only synchronized data, but also local data, which belongs to the local database only and never gets replicated from and to other databases.

In this way, tables are defined in both replica and master databases to distinguish local from shared data. Shared data is synchronized with the master database, but data belonging to only the replica or only the master is never changed during synchronization.

In a multi-master environment, catalogs keep data from different masters separated. Object name conflicts between catalogs do not occur. Even when the same table names and other object names are used in different masters, the distinct catalog names for each master qualify table and object names, as well as provide a way to specify which objects belong to which master. In addition, solidDB SmartFlow enforces that a single catalog contain no objects from different masters. However, using catalogs does not require that all objects in a schema be a synchronized object, so catalogs can contain local tables.

Schema names of a master and replica must be identical. This is consistent with the basic two-tiered architecture, which is no different from a single replica that is registered to a single master. When a database is created, a default schema name is created which is the user id of the database owner. Separate schemas within databases are created with the CREATE SCHEMA statement. For details on managing database objects with schemas, read *solidDB Administration Guide*.

solidDB SmartFlow's use of catalogs and schemas for synchronization offers a flexible and scalable architecture. One replica is registered to only one master; that is, one replica catalog is mapped to one master catalog. However, a single physical database can have multiple catalogs. Additional master databases are included by creating more master catalogs in the same local database which map to new replica catalogs in the same or different database servers. Furthermore, both master catalogs and replica catalogs can exist in the same physical database.

As shown in Figure 3.2, “Multi-tier Data Redundancy Architecture”, a master/replica hierarchy may have more than two layers, and some catalogs within that hierarchy may serve as both replica and master catalogs. For details on defining catalogs with the dual role of master and replica, read Section 5.2.1, “Defining Master and Replica Databases”.

For details on implementing catalogs, read the section called “Guidelines for Multi-Master Topology”.

Transactions in a Multi-Master Environment

In a multi-master environment, a transaction cannot span two different masters. For example, the following is invalid:

```
SAVE UPDATE table A in CATALOG A
```

```
SAVE UPDATE table B in CATALOG B
COMMIT WORK
```

A transaction is propagated to a specific master database. This master cannot be changed in the middle of a transaction, which means that all saved statements in the transaction are propagated to one master only. solidDB can detect cases where one transaction is updating data from two different masters. In such cases, the operation fails with an error message.

solidDB SmartFlow allows local data modifications in the same transaction within a catalog. The SET CATALOG command explicitly defines the master used for all SmartFlow-related operations. The SET CATALOG command is executed before any synchronization command and is required when more than one catalog is defined in the database.

3.2 solidDB SmartFlow Architecture Components

solidDB SmartFlow gives application programmers a rich set of data distribution and management functions. These functions support a reliable, flexible, and robust data distribution system that meets the specific needs of the application.

The solidDB SmartFlow architecture consists of the following functional components:

- *Master and replica* databases for storing official and tentative versions of data.
- *Publications and subscriptions* for transferring new and changed data from the master database to the replica database.
- *Intelligent Transactions* for propagating changes from a replica database to the master database.
- *Asynchronous store and forward messaging* for implementing safe and reliable communication between the master and a replica.

The components are described in greater detail in the following sections along with the role they play in synchronizing data.

3.2.1 Master and Replica Databases

In a distributed system that uses solidDB SmartFlow, the master database stores the official version of the data. This data includes the data of the business applications as well as the synchronization definitions. The synchronization definition data includes catalog, database schema, publication definitions, registration, and their subscriptions, user access definitions of the replica databases, etc.

The replica database stores local data, as well as transactions that are to be propagated to the master. All replica data, or a suitable part of it, can be refreshed from the master database whenever needed by sending a

```
REFRESH
```

command to one or more publications. The local data includes data of the business applications, typically a subset of the master database, as well as system tables that contain information specific to the particular database.

In a multi-tier synchronization environment, synchronized databases may be configured to serve a dual role, as both a master and a replica. These roles are established by creating a catalog and defining it to be both a replica and a master. Read Section 3.1.2, “Multi-Master Synchronization Model”.

3.2.2 Publications, Subscriptions, and Refreshes

The synchronization architecture of a multi-database system requires a way for applications to download data from the master database to the replica database, and to refresh this replica data on an as-needed basis.

A *publication* is a definition of a set of master data that can be downloaded to replicas. Replica databases use *subscriptions* to register their interest in a particular publication from the master. A publication is registered in a replica. Users can *refresh* data from only those publications that are registered. In this way publication parameters are validated, preventing users from accidentally refreshing from unwanted or non-existing publications or making ad hoc refresh commands.

The initial download ("refresh") always returns data of a full publication; all data of the publication that matches the search criteria (given as publication parameters) is sent to the replica database. For more details on publications and subscriptions, read Section 6.5, “Creating Publications”.

After the initial download, subsequent refreshes to the same publication (using the same parameter values) receive only the data that has been changed since the prior refresh. This is known as an *incremental refresh*. Typically, only publication updates with the latest modifications need to be sent to a replica. Creating publications and specifying that they be incremental are done through solidDB SmartFlow commands, which are extensions to Solid SQL. See Chapter 6, *Implementing a solidDB SmartFlow Application* for details on the

```
CREATE PUBLICATION
```

command.

3.2.3 Intelligent Transaction

The main concept in understanding SmartFlow is that the data of all replica databases is unofficial and therefore any modification done to data on a replica is tentative. The modification becomes official only when it is successfully validated and committed in the master database.

This "create now in replica, commit later in master" requirement extends the life cycle of a transaction from a fraction of a second to an undefined duration. In a multi-database system, transactions are propagated from

replicas to the master database over a time period that can vary from seconds to even weeks. The challenge of this kind of transaction is to ensure that whenever it is validated and committed to the master database, it changes the master database from one consistent state to another consistent state.

Ensuring Database Consistency

A database is consistent if the transactions that modify the contents of the database meet the following criteria at the commit moment of the transaction:

- DBMS specific rules, such as referential integrity rules, are not violated.
- Business rules that apply to the business transactions and their respective database transactions are not violated.

When the propagated replica transaction is eventually committed in the master database it is possible that the state of the master database is different than the state of the replica database (where the transaction was originally created). The state of the master database may have been changed because of propagated transactions from other replicas or updates done directly to the master database after the replica's latest refresh. For this reason, the replica transaction may not be used in the master database with its original content.

To address the consistency requirement in the two-tier replication model, each transaction that can become invalid during its life cycle must contain built-in business logic for ensuring that the master database remains consistent when the transaction is committed there. If the database becomes inconsistent with the original behavior of the transaction, the transaction must detect this and change the behavior so that the consistency of the database is maintained.

solidDB SmartFlow's Intelligent Transaction model provides a framework for implementing transactions with long life spans. Transaction propagation in SmartFlow architecture is based on Solid Intelligent Transaction technology. See Figure 2.1, “Propagate And Refresh” for a simple illustration of propagating and refreshing data when some records are accepted by the master and some are not.

Intelligent Transaction Scenario

To illustrate Intelligent Transaction implementation, assume an order entry application has a business rule that customers must not exceed their credit limit. If the limit has been exceeded, new orders are prohibited.

In a multi-database system it is possible that the customer credit limit in a replica database is OK, whereas the same data in the master database indicates a limit overrun. In this situation, a customer can still enter an order to the replica database, because the information about the limit overrun has not reached that database yet. However, when the "add a new order" transaction is propagated from the replica to the master database, it must not be committed in its original form, because that would mean a violation of the "credit limit" business rule. Instead the transaction needs to change its behavior to be valid. For instance, the "status" column of the

order must be given the value "invalid" in the master database to keep the order separate from the valid orders. The invalid order can be refreshed back to the replica to notify the replica users that the transaction has failed.

Multi-database Systems versus Centralized Systems

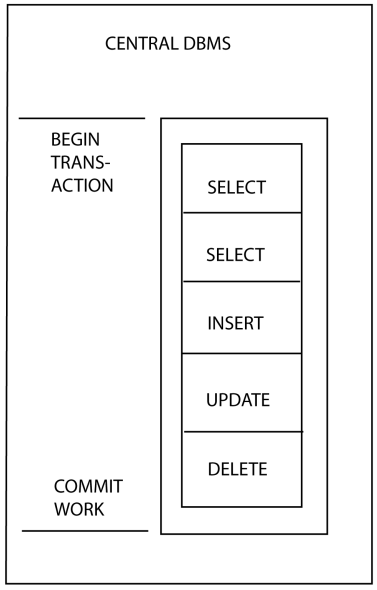
In a traditional client/server system that uses a central database, the validation logic of each transaction is typically in the client application or in the application server's services. For instance, in an Order Entry application, the application logic must check prior to committing the transaction that the credit limit of a customer is not exceeded by the new order.

When propagating a transaction to the master database, similar validation is needed to ensure the database integrity. The only difference is that the transaction validation logic of the application is not available to the synchronization mechanism. Therefore the logic must be bundled into the transaction itself. The following kind of validation logic is required in each transaction:

- update conflict detection
- validation using business rules
- DBMS error handling

The transactions of a centralized system are very different from the transactions of a multi-database system. In a centralized system, the lifetime of a transaction is typically a fraction of a second and with the DBMS locking mechanism, update conflicts are not possible.

Figure 3.5. Transaction in a Central Database

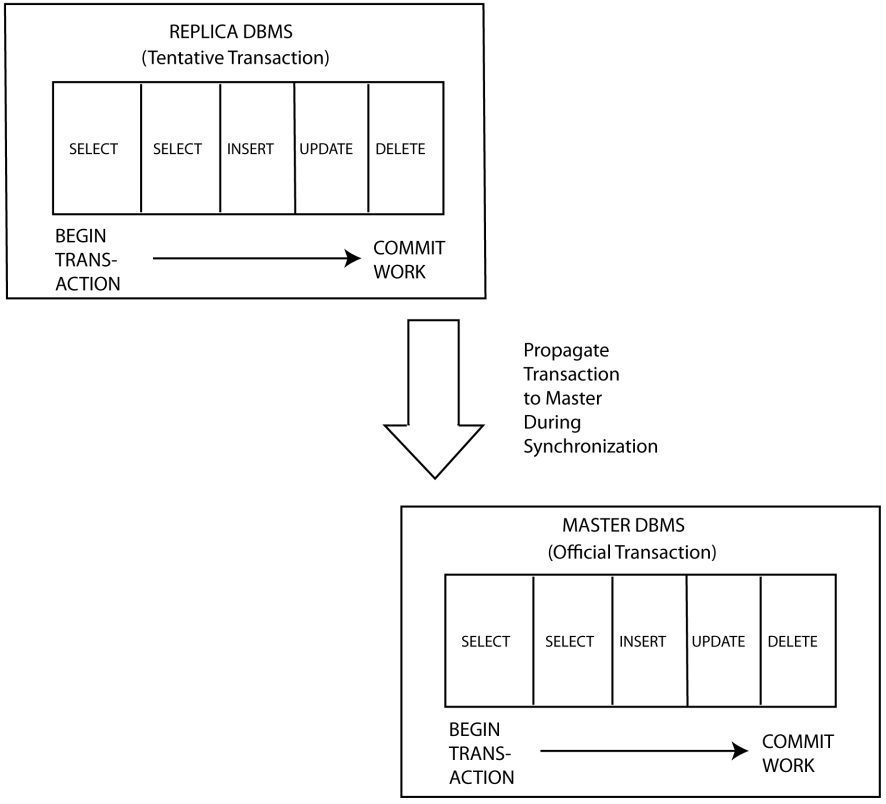


Unless local users have master access, they are unable to perform any synchronization operations.

The figure above illustrates a typical transaction. Within the transaction, some queries are made prior to write operations to validate the contents of the transaction. For example, an order entry system can check that a customer credit limit is OK prior to creating a new order to the customer. During the transaction, the concurrency control mechanism of the server takes care of the update conflicts and other issues caused by concurrent usage of the data.

In a multi-database system, a transaction is initially created and saved in the replica database but finally committed in the master database later when the transaction is propagated there as part of the database synchronization process. The tentatively committed transaction can exist in the system for an unlimited period of time. In other words, the life cycle of the transaction is entirely different.

Figure 3.6. Transaction in a Synchronized Database

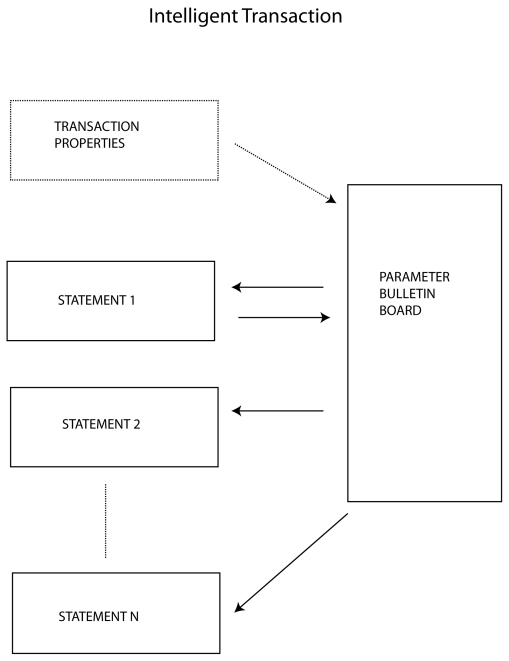


Intelligent Transaction in the Multi-Database System

In a multi-database system, a transaction has "two lives". The transaction is created as tentative in the replica database where it is validated and committed by the business application. The transaction is saved (i.e. put to the transaction queue) in the replica database for later propagation to the master database. The transaction has its "second life" when it is propagated to the master database. There it must perform the same validation routines, that is, the queries that were performed in the replica database. For instance, if a customer credit limit was checked in the replica database to ensure the transaction validity, the same operation must usually be done also in the master database prior to committing the transaction. Otherwise the validity of the transaction cannot be guaranteed in the master database.

To support the extended life cycle of the two-tier data redundancy model, an extension to the database transactions has been developed. Solid Intelligent Transaction allows a transaction to validate itself in the master database and adjust its behavior to ensure the validity of the transaction.

Figure 3.7. The Structure of Intelligent Transaction



How Intelligent Transaction Functionality Works

With Solid Intelligent Transaction functionality, a transaction not only has the capability of validating itself in its current database, but is also capable of changing its behavior (that is, the database operations) if the original behavior was invalid.

Statements of a transaction can be any SQL statements, but most often are calls to stored procedures. Statements should contain logic that is required to ensure the validity of the statement in different environments and situations.

When executing the transaction in the master database, the statements of the transaction can communicate with each other by putting *transaction parameters* on the Parameter Bulletin Board for the following statements of the same transaction to read. This communication ability of the statements makes it possible to create transactions that can validate themselves and adjust their behavior according to the current environment.

For example:

This example applies to the Intelligent Transaction scenario presented at the beginning of this section. In that scenario, a transaction which is propagated to the master database attempts to add a new order to a customer whose credit limit has been exceeded. This example now presents the SQL statement logic behind the transaction.

The transaction has the following operations:

- Insert a row to the CUST_ORDER table
- Update the CREDIT column of the CUSTOMER table

This is the processing that occurs:

1. Prior to inserting a new row to the CUST_ORDER table, the INSERT_ORDER procedure checks that the customer credit is OK. In this example, we assume it is not.
2. The intelligent transaction, therefore, inserts the new row to the CUST_ORDER table with a different STATUS value (for example, STATUS = 'Not approved').
3. Because the order is not a valid one, the update operation of the CREDIT column must not be done. Therefore the INSERT_ORDER procedure puts a parameter with name "ORDER_FAILED" and value "YES" to the bulletin board.
4. The UPDATE_CUST_CREDIT procedure checks the bulletin board and detects that it contains information that applies to this procedure.
5. Now the UPDATE_CUST_CREDIT procedure knows that it must not update the credit amount.
6. Later, when the replica refreshes to get up-to-date data, the replica's own information about the customer order will be updated, and the data on the replica will correctly indicate that the order was not processed Thus we have "closed the loop", ensuring that both the master and replica have correct data, even when the replica made a request/update that could not be fulfilled.

3.2.4 Asynchronous Store and Forward Messaging

Communication between the solidDB SmartFlow master and replicas is based on asynchronous store and forward messaging. Each message is assembled dynamically and can contain numerous synchronization tasks. For example, it is possible to propagate multiple transactions from a replica to the master and request refreshes from numerous publications from the master database in one message.

The messages from the replica are sent asynchronously to the master database. Message queuing capabilities built into solidDB SmartFlow architecture guarantee that no message is deleted from the sending node before the entire message has arrived at the receiving node.

Chapter 4. Getting Started With Data Synchronization

solidDB features Solid's patented data distribution technology, which is called "SmartFlow", and we will use the term SmartFlow frequently in the next few chapters.

This chapter gives detailed instructions on executing sample scripts that demonstrate basic SmartFlow functionality. This chapter assumes familiarity of solidDB basic administration functions described in *solidDB Administration Guide*. Knowledge in the following areas is assumed:

- Basic administration of solidDB (installing, starting, shutting down, configuring network names, etc.). Read relevant chapters of *solidDB Administration Guide* for details.
- Working with solidDB (connecting to the database, running SQL statements)
- Solid SQL stored procedure programming
- Fundamentals of solidDB SmartFlow architecture. Be sure to read Chapter 3, *solidDB Data Synchronization Architecture* before you use this chapter.

4.1 Before You Begin

In SmartFlow, synchronization is implemented with SQL statements. The data synchronization SQL statements described in this chapter can be used through SQL Editor (teletype), as well as through the ODBC, JDBC, and Light Client APIs. For details on passing SQL to solidDB using an API, please refer to the documentation for that particular API. (See *solidDB Programmer Guide* for documentation on the APIs.)

Before you begin to use the sample scripts, you need to prepare your solidDB environment for executing them. Instructions for setting up a solidDB evaluation environment are provided in a separate document named *solidDB Getting Started Guide*.

4.2 SmartFlow Implementation

This chapter contains a brief step-by-step example on setting up, configuring and using solidDB to use SmartFlow technology.

Table 4.1. Chapter Summary

Step	Action	Description
1	Getting started	Contains instructions on installing and starting two solidDB database servers.
2	Configuring the servers and creating the publications.	Configures the servers as master and replica. Defines a table (in both the master and replica databases) that meets the SmartFlow design requirements. Defines a sample publication in the master database.
3	Providing transactions	Contains two transactions implemented as Solid stored procedures in both the master and replica databases.
4	Using synchronization functionality	Registers the sample publication, updates the replica database, and saves the transactions for propagation to the master database.
5	Implementing synchronization messages	Contains solidDB SmartFlow SQL commands to be run from the console in order to synchronize the databases.

4.2.1 About the Scripts

This chapter contains instructions on running basic solidDB SmartFlow operations between two databases. This example is not a full application. The database schema contains only one table and the transactions contained in this chapter are basic ones. Their purpose is to illustrate for you the basic functionality of SmartFlow architecture.

All the SQL scripts included in this chapter are also available in the following directory of the Solid package: `samples/smartflow/eval_setup`.

4.3 Getting Started

Before you get started running the sample scripts, be sure you have:

- Installed the solidDB Development Kit successfully. For details, read the release notes file contained in the solidDB Development Kit.
- Prepared the environment setup described in

  `solid/samples/smartflow/eval_setup/readme.txt`.

4.3.1 Setting Up the Environment

To set up the solidDB SmartFlow environment, start solidDB for the master and replica databases. Instructions for starting the master and replica databases are found in the `readme.txt`.

4.3.2 Designing a Database Table

The typical requirements for a database table used in a synchronized application are:

- It must have unique primary key to ensure global uniqueness of rows.
- It may need to have a row status column to provide a means for handling update conflicts.
- It should include a version number or the time of the last update to allow detection of update conflicts.

Read Section 5.2.3, “Defining a Database Table” for a detailed discussion about these requirements. The SQL statement used later creates a table that meets these criteria.

Following are descriptions of the columns:

- REPLICAID contains a unique identification for the database. In this sample script, note that value 1 is reserved for the replica and value 2 for the master.
- ID is a row identifier that is unique inside the database where the row is created.
- Status has the following values: -1 for updates invalidated by master, 1 for tentative replica data and 2 for official master data.
- INTDATA and TEXTDATA demonstrate the "business data" of the table.

Setting the SYNCHISTORY property on for the table enables incremental publication, which means that only the modified rows are transferred from the master to the replica when synchronizing the table data. Setting the SYNCHISTORY property creates a "shadow table" for the SYNCDEMO main table. The name of the "shadow table" is the name of the main table with "_SYNCHIST_" prefix. Old versions of updated and deleted rows are moved to this shadow table. SYNCHISTORY must be active in *both* the master and the replica databases.

As with any table to be synchronized, we will define the table in both the master and replica databases.

4.4 Configuring the Servers and Creating the Publications

In this section, we will do the following:

1. Configure one server as a master.
2. Configure the other server as a replica.
3. Create the table on each server.
4. On the master, create a publication.
5. On the replica, register to the publication.

We will execute two scripts to carry out these steps. One script is run on the master, and the other on the replica.

4.4.1 On the Master

Connect to the master and run the `master1.sql` SQL script. One way to do this is shown below:

Move to the Solid installation root directory and enter the following command (all on one line):

./bin/solsql -O eval.out "tcp 1315" dba dba_password ./samples/smartflow/eval_setup/master1.sql

where:

- *-O eval.out* is an optional parameter that defines the output file for results.
- *"tcp 1315"* is the network protocol and address of the master server. You may have to customize this part of the command.
- *dba* and *dba_password* are username and password respectively.
- *master1.sql* is the executed SQL script.

You can view the results in `eval.out` with any text editor.

Note

When you execute SmartFlow scripts, you must set the autocommit mode OFF. If you are using SolidConsole, the autocommit mode is set ON by default, so be sure to set it OFF. If you are using the SQL Editor (teletype), the autocommit mode is set OFF by default.

```
--*****************************************
-- master1.sql
-- Execute this script in the MASTER database.
-- Initializes the master with node name MASTER.
-- Also creates the table and the publication.
--*****************************************

-- Create the catalog named "sync_demo_catalog".
-- Give this node the name "master_node".
-- Register this node as a master, and not as a replica.

CALL SYNC_SETUP_CATALOG (
   'sync_demo_catalog',
   'master_node',
   1,
   0);
COMMIT WORK;

-- Set the catalog to be the current catalog.
SET CATALOG sync_demo_catalog;
COMMIT WORK;

-- Create the table that we will synchronize.
CREATE TABLE SYNCDEMO
(
   REPLICAID INTEGER NOT NULL,
   ID INTEGER NOT NULL,
   STATUS INTEGER NOT NULL,
   INTDATA INTEGER,
   TEXTDATA CHAR(30),
   UPDATETIME TIMESTAMP,
   PRIMARY KEY (REPLICAID, ID, STATUS)
);
ALTER TABLE SYNCDEMO SET SYNCHISTORY;
```

```
COMMIT WORK;

-- Create a publication that publishes all data of the SyncDemo table.
-- Note that CREATE PUBLICATION commands must be inside double quotes.
"CREATE PUBLICATION PUB_DEMO
   BEGIN
   RESULT SET FOR SYNCDEMO
   BEGIN
      SELECT * FROM SYNCDEMO ;
   END
END";
COMMIT WORK;
```

This publication will contain all the rows in the `syncdemo` table. Refreshes from this publication will be incremental because the SYNCHISTORY property is set for this table in both the master and replica databases. See *solidDB SQL Guide* for more information about the SYNC_SETUP_CATALOG stored procedure.

4.4.2 On the Replica

Move to the Solid installation root directory and enter the following command (all on one line):

./bin/solsql -O eval.out "tcp 1316" dba dba_password ./samples/smartflow/eval_setup/replica1.sql

where:

- *-O eval.out* is an optional parameter that defines the output file for results.
- *"tcp 1316"* is the network protocol and address of the master server. You may have to customize this part of the command.
- *dba* and *dba_password* are username and password respectively.
- `master1.sql` is the executed SQL script.

You can browse the results in `eval.out` with any text editor.

If you have followed instructions in the `readme.txt` document, no changes are required to `replica1.sql`. Otherwise, you may need to make changes to the following areas in the script:

- In the call to `SYNC_REGISTER_REPLICA()` you may have to specify a different user id and password for connecting to the master database for the first time.

- In the call to `SYNC_REGISTER_REPLICA()` you may have to set the master database's connection string to a different value from 'tcp localhost 1315'.

Note

When you execute solidDB SmartFlow scripts, you must set the autocommit mode OFF. If you are using SolidConsole, the autocommit mode is set ON by default, so be sure to set it OFF. If you are using the SQL Editor (teletype), the autocommit mode is set OFF by default.

`replica1.sql`

```
--*******************************************************
-- replica1.sql
-- Initialize a replica database with node name
-- "replica_node_01".
-- (Each replica must have a unique node name.)
-- Execute this script in the REPLICA database.
-- NOTE: AUTOCOMMIT must be set off for MESSAGE handling!
--*******************************************************
-- Create a replica catalog named 'sync_demo_catalog' and
-- register it with the master database server called
-- 'master_node'. Also, specify the network address and name of the
-- master node, and specify the user ID and password to use.

CALL SYNC_REGISTER_REPLICA (
   'replica_node_01',
   'sync_demo_catalog',
   'tcp localhost 1315', -- Master's net address. CUSTOMIZE!
   'master_node', -- Node name of master.
   'dba',
   'dba_password');
COMMIT WORK;

SET CATALOG sync_demo_catalog;
COMMIT WORK;

-- Create the table.
CREATE TABLE SYNCDEMO
(
   REPLICAID INTEGER NOT NULL,
   ID INTEGER NOT NULL,
```

```
   STATUS INTEGER NOT NULL,
   INTDATA INTEGER,
   TEXTDATA CHAR(30),
   UPDATETIME TIMESTAMP,
   PRIMARY KEY (REPLICAID, ID, STATUS)
);
-- Enable the use of incremental publications for this
table.
ALTER TABLE SYNCDEMO SET SYNCHISTORY;
COMMIT WORK;

-- Register a publication that is already defined in the master.
CALL SYNC_REGISTER_PUBLICATION (
   'sync_demo_catalog',
   'pub_demo');
COMMIT WORK;
```

For more information about the stored procedures SYNC_REGISTER_REPLICA and SYNC_REGISTER_PUBLICATION see the *solidDB SQL Guide*.

4.5 Providing Transactions

Typically, write operations in synchronized architecture are implemented using stored procedures. This allows you to implement business logic in the transactions to handle possible conflicts without violating the application's data integrity and business rules. For details on stored procedures, see *solidDB Programmer Guide*.

The procedure logic in the following scripts is designed to be as simple as possible. The only logic for handling conflicts is implemented in the update procedure. If the update fails, then the row is inserted with status -1. Other values for status field are 1 for a tentative write at the replica and 2 for the official data accepted by the master.

Run the `proced1.sql` and `proced2.sql` scripts at both the replica and master databases.

The procedure below inserts a row into table syncdemo. For simplicity, there is no processing for errors. A real application should have logic for handling unique key constraint violations and other possible error situations.

PROCED1.SQL

```
--*******************************************************
-- proced1.sql
```

```
-- Creates a procedure for inserting data into the sample
-- table SYNCDEMO.
--
-- Execute in both the MASTER and REPLICA DBs.
--
-- NOTE: THIS HAS NO VALIDATION RULES. DUPLICATES ARE HANDLED
-- BY IGNORING THE DUPLICATE INSERT!
--*******************************************************

   SET CATALOG sync_demo_catalog ;

"CREATE PROCEDURE SYNCDEMO_INSERT
   (MACHINEID INTEGER,
   ID INTEGER,
   INTDATA INTEGER,
   TEXTDATA CHAR(20),
   UPDATETIME TIMESTAMP,
   TARGETDB CHAR(1))
RETURNS
   (SUCCESS INTEGER, ROWS_AFFECTED INTEGER)

BEGIN

   DECLARE STATUS INTEGER ;
   IF TARGETDB = 'R' THEN
      STATUS := 1 ;
         ELSE IF TARGETDB = 'M' THEN
         STATUS := 2;
      ELSE
         STATUS := -1;
      END IF;
   END IF;

   EXEC SQL PREPARE SYNCDEMO_INS
   INSERT INTO SYNCDEMO
   (REPLICAID,ID,STATUS,INTDATA,TEXTDATA,UPDATETIME)
   VALUES(?,?,?,?,?,?);

   EXEC SQL EXECUTE SYNCDEMO_INS USING
   (MACHINEID,ID,STATUS,INTDATA,TEXTDATA,UPDATETIME);

   SUCCESS := SQLSUCCESS;
```

```
   ROWS_AFFECTED := SQLROWCOUNT;

   EXEC SQL CLOSE SYNCDEMO_INS;
   EXEC SQL DROP SYNCDEMO_INS;

END";
COMMIT WORK;
```

The following procedure updates a row in the `syncdemo` table. It contains a simple implementation of conflict resolution logic. If the procedure finds the row does not exist, it inserts the row with status *-1* instead. Please note that the logic of a real application should be able to handle an almost unlimited number of conflicts as well as other transaction validation errors.

PROCED2.SQL

```
--*******************************************************
-- proced2.sql
-- Creates a procedure for updating data of the sample
-- table SYNCDEMO.
--
-- Execute in both the MASTER and REPLICA DBs.
--
-- VALIDATION RULE: IF THE TIMESTAMP HAS CHANGED, THEN THE UPDATE
-- IS CHANGED TO INSERT WITH PARAMETER 'TARGETDB' AS 'F'.
-- THIS WILL RESULT IN AN INSERT WITH THE STATUS OF -1 IN
-- THE CALLED PROCEDURE SYNCDEMO_INSERT.
--*******************************************************
SET CATALOG sync_demo_catalog ;

"CREATE PROCEDURE SYNCDEMO_UPDATE
(
MACHINEID INTEGER,
ID INTEGER,
INTDATA INTEGER,
TEXTDATA CHAR(20),
UPDATETIME TIMESTAMP,
TARGETDB CHAR(1)
)
RETURNS
(SUCCESS INTEGER, ROWS_AFFECTED INTEGER)
BEGIN
```

```
DECLARE TMPSTR VARCHAR;
DECLARE TNOW TIMESTAMP;
DECLARE DSTAT INTEGER;
DECLARE STATUS INTEGER;
IF TARGETDB = 'R' THEN
STATUS := 1;
ELSE
STATUS := 2;
END IF;

TNOW := NOW();
TMPSTR := 'R';
DSTAT := -1;

EXEC SQL PREPARE SYNCDEMO_UPD
UPDATE SYNCDEMO SET
   STATUS = ?,
   INTDATA = ?,
   TEXTDATA = ?,
   UPDATETIME = ?
WHERE
   REPLICAID = ? AND
   ID = ? AND
   UPDATETIME = ? ;

EXEC SQL EXECUTE SYNCDEMO_UPD USING
   (STATUS, INTDATA, TEXTDATA, TNOW, MACHINEID, ID, UPDATETIME);

SUCCESS := SQLSUCCESS ;
ROWS_AFFECTED := SQLROWCOUNT;

IF (SUCCESS = 1) AND (ROWS_AFFECTED = 0) THEN
   TMPSTR := 'F' ;

   EXEC SQL PREPARE SYNC_UPD1 CALL
      SYNCDEMO_INSERT(?,?,?,?,?,?);
   EXEC SQL EXECUTE SYNC_UPD1 USING
      (MACHINEID, ID, INTDATA, TEXTDATA, TNOW, TARGETDB);
   EXEC SQL FETCH SYNC_UPD1 ;

   SUCCESS := SQLSUCCESS;
   ROWS_AFFECTED := SQLROWCOUNT;
```

```
END IF ;

END";
COMMIT WORK;
```

Note

To keep this walk-through as simple as possible, the procedure above takes a parameter value for the column ID instead of using a sequence value. Also a parameter value is provided for the database ID. For a real application, the column ID should contain a sequence value. For more information about using sequences, refer to *solidDB Programmer Guide*.

4.6 Using solidDB SmartFlow Functionality

You are ready to run the following scripts, which demonstrate basic solidDB SmartFlow functionality. This includes:

- Registering a publication to the replica so that once it requests a refresh to the publication, the replica can receive it. Registering publications allows publication parameters to be validated. This prevents users from accidentally requesting refreshes they do not want, or requesting ad-hoc refreshes.
- Updating the replica with a transaction, which is saved at the replica for later propagation to the master database.

To proceed with synchronization:

1. Run the following SQL statements in both the replica and master databases:

   `select.sql`

   ```
   --*******************************************************
   -- select.sql
   -- Sql statement to check the status of the sample table
   --*******************************************************
   SET CATALOG sync_demo_catalog ;
   COMMIT WORK ;
   SELECT * FROM SYNCDEMO ;
   ```

   Since you have note yet inserted any rows in the table, you can see 0 rows returned.

2. Register publication and insert two rows in the replica by running the following statements at the replica:

REPLICA2.SQL

```
--*******************************************************
-- replica2.sql
-- This script registers to publication PUB_DEMO, inserts two
-- rows to the REPLICA database and
-- saves the transaction to be propagated to the MASTER
--
-- Execute in the REPLICA database
--*******************************************************
-- register to publication
SET CATALOG sync_demo_catalog ;
MESSAGE REG_PUBL BEGIN;
MESSAGE REG_PUBL APPEND REGISTER PUBLICATION
PUB_DEMO;
MESSAGE REG_PUBL END;
COMMIT WORK;
MESSAGE REG_PUBL FORWARD TIMEOUT FOREVER;
COMMIT WORK;

CALL SYNCDEMO_INSERT (1,1,100,'First row','1998-05-15 12:00:00','R');
SAVE CALL SYNCDEMO_INSERT (1,1,100,'First row','1998-05-15 12:00:00','M');
CALL SYNCDEMO_INSERT (1,2,101,'Second row','1998-05-15 12:00:01','R');
SAVE CALL SYNCDEMO_INSERT (1,2,101,'Second row','1998-05-15 12:00:01','M');

COMMIT WORK;
```

3. Use `select.sql` to verify that the replica contains two rows and the master none.

   At this point, the replica database also has two saved statements in one transaction waiting to be propagated to the master database.

4.7 Implementing Synchronization Messages

The synchronization messages are programmed using SQL statements that are executed in a replica database.

1. Synchronize master and replica databases by running `replica3.sql` in the replica.

   `replica3.sql` creates a message `my_msg`, which does two things:

a. propagate all local transactions to the master, and

b. refresh from the `pub_demo` publication to get any updated data.

After the script creates the message, the script sends the message to the master database and waits for a reply.

REPLICA3.SQL

```
--*******************************************************
-- replica3.sql
-- creates a new message with name 'my_msg'
-- append tasks to message my_msg:
--             propagate transactions
--             refresh publications
--
-- Execute in the REPLICA database.
--
-- NOTE: AUTOCOMMIT must be off!
--*******************************************************
SET CATALOG replica_catalog ;
MESSAGE my_msg BEGIN ;
MESSAGE my_msg APPEND PROPAGATE TRANSACTIONS ;
MESSAGE my_msg APPEND REFRESH PUB_DEMO ;
MESSAGE my_msg END ;
COMMIT WORK ;

-- Send the message to the master, don't
wait for reply.
MESSAGE my_msg FORWARD;
COMMIT WORK;
-- Request reply to the message separately from
master.
MESSAGE my_msg GET REPLY TIMEOUT DEFAULT ;
COMMIT WORK;
```

Use `select.sql` to verify that both the replica and master contain two rows.

2. Delete a row from the master by running the following statements at the master:

MASTER2.SQL

```
--************************************************
-- master2.sql
-- Deletes a row from the sample table
-- Execute in the MASTER database
--************************************************
SET CATALOG sync_demo_catalog ;
DELETE FROM SYNCDEMO WHERE ID = 2;
   COMMIT WORK ;
```

Use `select.sql` to verify that the replica contains two rows and the master contains one.

3. Synchronize by running all statements in `replica3.sql` in the replica.

   Use `select.sql` to verify that both replica and master contain one row.

4. Insert two more rows in the replica by running all statements in `replica4.sql` in the replica:

   REPLICA4.SQL

```
--******************************************************
-- replica4.sql
-- This script inserts two rows to the REPLICA database
-- and saves the transaction to be propagated to the MASTER
--
-- Execute in the REPLICA database
--
******************************************************
SET CATALOG sync_demo_catalog; CALL SYNCDEMO_INSERT
(1,3,102,'Third row','1998-05-15 12:10:00','R');
SAVE CALL SYNCDEMO_INSERT (1,3,102,'Third
row','1998-05-15 12:10:00','M');
CALL SYNCDEMO_INSERT (1,4,103,'Fourth
row','1998-05-15 12:10:01','R');
SAVE CALL SYNCDEMO_INSERT (1,4,103,'Fourth
row','1998-05-15 12:10:01','M');
COMMIT WORK;
```

5. Synchronize by running all statements in `replica3.sql` in the replica.

   Use `select.sql` to verify that both the replica and master contain three rows.

6. Update one row at the replica by running `replica5.sql` in the replica:

   REPLICA5.SQL

```
--*******************************************************
-- replica5.sql
-- This script updates one row in the REPLICA database
-- and saves the row to be propagated to the MASTER
--
-- Execute in the REPLICA database
--*******************************************************
SET CATALOG sync_demo_catalog ;

CALL SYNCDEMO_UPDATE (1,1,201,'Row 1
changed','1998-05-15 12:00:00','R');
SAVE CALL SYNCDEMO_UPDATE (1,1,201,'Row 1
changed','1998-05-15 12:00:00','M');
COMMIT WORK;
```

7. Synchronize by running all statements in `replica3.sql` in the replica.

   Use `select.sql` to verify that both the replica and master databases contain three rows and that the update has been propagated to the master database.

8. Update one row at the master by running `master3.sql` in the master:

   MASTER3.SQL

```
--*******************************************************
-- master3.sql
-- This script updates one row in the MASTER database
--
-- Execute in the MASTER database
--*******************************************************
SET CATALOG sync_demo_catalog ;

CALL SYNCDEMO_UPDATE (1,3,203,'Row 3
masterchange','1998-05-15
```

```
12:10:00','M');
COMMIT WORK ;
```

Do not synchronize the data to replica.

Use `select.sql` to verify that both the replica and master contain three rows and that the last update has occurred only at the master database.

9. Update the same row at the replica by running `replica6.sql` in the replica:

REPLICA6.SQL

```
--*******************************************************
-- replica6.sql
-- This script updates one row in the REPLICA database
-- and saves the row to be propagated to the MASTER
--
-- Execute in the REPLICA database
--*******************************************************
SET CATALOG sync_demo_catalog ;
CALL SYNCDEMO_UPDATE (1,3,203,'Row 3
replicachange','1998-05-15 12:10:00','R');
SAVE CALL SYNCDEMO_UPDATE (1,3,203,'Row 3
replicachange','1998-05-15 12:10:00','M');
COMMIT WORK;
```

Use `select.sql` to verify that the updates in master and replica are now different.

10. Synchronize by running all statements in `replica3.sql` in the replica.

Use `select.sql` to verify that the conflict caused by updating the same row got processed properly.

Both master and replica should contain four rows. One of the rows should have invalid (*-1*) status since the last update operation at the replica will cause a conflict at master database.

4.8 Sync Pull Notify

So far, our discussion of data synchronization has focused on the "pull" model, in which the replica decides when to "pull" a copy of updated data from the master. However, in many situations, users may prefer a "push" model, where the master determines when to send data to the replicas. For example, in a "push" model, the master can send data as soon as that data is updated, rather than waiting for replicas to request an updated copy.

The difference between "pull" and "push" models is similar to the difference between "polling" and responding to "signals" or "interrupts". In a "pull" model, the replica never knows when new data has arrived at the master, and thus the replica typically "polls" the master at regular intervals, or when reconnecting to the network, or when manually told to do so by the replica's synchronization administrator. In a "push" model, on the other hand, the master knows when new or updated data has arrived, and sends the data to the replicas.

The advantage of the "pull" model is that it tends to semi-randomly distribute requests to the master; this keeps the master from being overloaded with a large number of update requests at once. The "pull" model also works well when replicas are connected to the network only irregularly (e.g. if you keep your data on a PDA (Personal Digital Assistant) and only synchronize your PDA occasionally). The advantages of the "push" model are, of course, that out-of-date data is updated rapidly, and replicas don't have to waste time "polling" for new data when none has arrived.

Although solidDB does not provide a true "push" capability, solidDB provides Sync Pull Notify capability, which is similar to, but more flexible than, push synchronization. With the Sync Pull Notify approach, the master notifies the replica that there is updated data, and the replica may then choose whether to download the updated data.

To implement a system based on Sync Pull Notify, the system must meet the following requirements:

- Updated data must not be "pushed" to replicas until it is committed. Basically, the data should be pushed at the end of the transaction, not during the transaction.

- The system must be able to detect when data has changed in the master and thus may need to be pushed to the replica(s).

In solidDB SmartFlow, Sync Pull Notify is not implemented as a feature; instead the Sync Pull Notify capability is implemented by using a combination of the features "START AFTER COMMIT", "Remote Stored Procedures", and "Replica Property Names". These features are described briefly below. More complete descriptions are elsewhere (Replica Property Names are described later in this manual, and Remote Stored Procedures and START AFTER COMMIT are documented in *solidDB SQL Guide*).

The Remote Stored Procedure feature is just what it sounds like — a way to call a stored procedure in another database. Specifically, a replica may create a stored procedure, and a master may call that stored procedure. For example, the replica might create a stored procedure that contains a REFRESH command, and the master could then call that procedure, thus causing the replica to request the refresh.

The "START AFTER COMMIT" feature allows a user to specify an action that will be taken when a transaction is committed. For example, if a specific transaction updates information on the master, then the START AFTER COMMIT could tell the master to do a remote procedure call to the replica after the update has been successfully committed.

The combination of START AFTER COMMIT and Remote Stored Procedures allows you to implement Sync Pull Notify. When the master updates some data, the master can notify the subscribers by creating a START AFTER COMMIT that calls a Remote Stored Procedure on each replica; the stored procedure then refreshes to get the updated data.

The next issue is to decide which replicas are notified — in other words, on which replicas do we call the stored procedure when data is updated. To control this, we use the feature "Replica Property Names", described below. (It is also possible to refer to a single replica using its nodename.)

Note

The START AFTER COMMIT feature and the Remote Stored Procedures feature are documented in more detail in *solidDB SQL Guide*. You may want to read more about those features before you continue reading about Sync Pull Notify.

4.8.1 Replica Property Names

Property Names allow a replica to be labeled. Replicas that are labeled may be grouped, which becomes important when using the START AFTER COMMIT feature (see START AFTER COMMIT ...). For example, you might have some replicas that are related to the bicycle industry and others that are related to the surfboard industry, and you may want to update each of those groups of replicas separately. You can use Property Names to group these replicas. All members of a group have the same property and have the same value for that property.

Properties and values may be almost any arbitrary names and literals. For example:

Table 4.2. Replica Properties and Values

Property	Some Possible Values
Region	north, south, east, west
Color	red, yellow, green, blue, rainbow, transparent
Mood	upbeat, blue
Philosophy	realist, nihilist, purple

Because the properties and values have no meaning to the server and are simply arbitrary labels, the values do not need to "make sense". If you decide that "purple" is a useful value in the "philosophy" category, then go ahead and use it. Furthermore, different properties may use overlapping values. If you decide that "blue" is both a useful color and a useful mood, then you may use it as a value for both properties. (A server categorized as having a blue mood will NOT fall into the same category as servers having a blue color; properties are completely independent.)

A server may have more than one property, and thus may belong to more than one group. For example, my replica might have

Region = west

Color = green

A server does not have to have a setting for every possible property; my replica might not have any Philosophy or Mood property at all.

For examples of using properties, see the section on the START AFTER COMMIT command.

Replica property names may be added at any time, and new replicas may also be added at any time. By using property names to group your replicas, you avoid having to re-write the logic on the master to call remote stored procedures on replicas. For example, suppose that you want to notify all offices in the "north" region when a particular piece of data is updated. If you add a new office (replica), you can simply give that new office the property "Region=north", and that new office will automatically be notified when relevant data is updated. Nothing on the master needs to be changed, except that the master must be notified that the new replica has the property "Region=north". Also, of course, the new replica must have a copy of the stored procedure that the master expects to call.

Every replica has an implicit property called "name", which is the node name assigned to the replica when it was registered.

A replica's properties are useful in Sync Pull Notify only if the master knows the replica's properties. Either the master or the replica may set a property for the replica. If the replica sets its own property, then it must notify the master of that property and value. The replica can send its properties to the master with the messages using SAVE. The master can set properties for a named replica with the SET command.

Syntax in master:

```
SET SYNC PROPERTY <propertyname> = { 'value' | NONE } FOR REPLICA
<replicaname>
```

Syntax in replica:

```
SAVE SET SYNC PROPERTY <propertyname> = { 'value' | NONE; }
```

Examples

Master:

```
SET SYNC PROPERTY color = 'red' FOR REPLICA replica_node_01;
SET SYNC PROPERTY color = NONE FOR REPLICA replica_node_01;
```

Replica:

```
SAVE SET SYNC PROPERTY color = 'red';
SAVE SET SYNC PROPERTY color = NONE;
```

4.8.2 Introduction to Sync Pull Notify

Before we discuss Sync Pull Notify in detail, we should contrast it with other methods of replicating data.

"Sync Push" vs. "Sync Pull" vs. "Sync Pull Notify"

When data is replicated, the servers can use

- a "pull" approach, or
- a "push" approach, or
- a hybrid, such as "sync pull notify".

solidDB SmartFlow supports the "pull" and "pull notify" approaches.

The "pull" approach is that the replica requests the data. This corresponds to the "refresh" operation in solidDB SmartFlow. When the replica refreshes, it requests that the master server send all of the data in a particular publication (or the data that has changed since the last refresh operation).

The "push" approach is that the master sends the data to the replica at a time chosen by the master. (This is usually, but not necessarily, immediately after the data was updated on the master). When the master pushes the data, the replica must accept the data. solidDB servers do not use a true "push" method.

solidDB uses a hybrid approach called "sync pull notify". In this approach, the replica is notified that new data is available. The replica may then "pull" the new data by executing a REFRESH command. The REFRESH is optional; after being notified that there is new data, the replica can choose to refresh immediately, or to refresh after delaying, or simply to ignore the notification and not refresh at all. See the next section for details.

Implementing Sync Pull Notify

The solidDB server implements Sync Pull Notify as a two-step process. First, the master server notifies the replica server; second, the replica server does a REFRESH.

Generally, the master server notifies the replica by calling a remote stored procedure. The replica must already have created this stored procedure. Normally, this procedure will itself contain the appropriate REFRESH command(s). (It is possible to use more indirect methods, but this approach is the shortest and simplest.)

What induces the master to call the remote stored procedure? One way to do this is to create a trigger. For example, suppose that you know that replica_node_01 would like to refresh data in the table named "employees" every time the table is changed on the master. You could create INSERT, UPDATE, and DELETE triggers on the master's copy of the employees table. Each trigger would call the remote stored procedure on replica_node_01 that requests a refresh of the data.

```
trigger
 |
 |
 v

stored procedure
 |
 |
 v

REFRESH statement
```

Although the process of notifying a replica is straightforward, there is no automated way to decide which replica(s) should be notified. The person who creates or updates the triggers on the master must know the name of the stored procedure to call on each replica that wants to be notified.

Similarly, the replica must have REFRESH statements that specify the appropriate publications. There is no automated way of having the master server figure out which replicas subscribe to which publications and then automatically creating triggers to call procedures on those replicas. (This wouldn't be sufficient anyway because the master server would also have to compose the appropriate stored procedure and force the replica to create that stored procedure.)

Whether you use a trigger or a START AFTER COMMIT or some other method to notify replicas that data in the master has been updated, it is important to remember that only the specified nodes are notified.

Note

If you decide to use a trigger to initiate Sync Pull Notify or use some other mechanism to run synchronization very frequently, always make sure that you are aware of the possible performance consequences. Because synchronization messaging is based on a store & forward architecture that writes

data to disk before sending it over the network, there is always some overhead related to synchronization messaging. The relative overhead of the messaging is the bigger, the smaller amount of data is to be synchronized. For example, a solidDB server is capable of performing hundreds or thousands of update operations per second but it may be able to handle only some dozens of synchronization messages per second. If you write an update trigger into a frequently updated table to initiate synchronization for each occurred update, the update performance is limited by the synchronization messaging performance. Instead of achieving performance of 1000 updates per second, you may be able to run only 10 updates per second because every update causes a synchronization message that contains only one row of changed data. The remedy to this issue is to make the synchronization occur only e.g. once every few seconds. This way, each synchronization message contains more than one row in it and the overhead of the synchronization message is only a fraction of the worst case scenario.

DEFAULT Keyword in Remote Stored Procedures

To understand the DEFAULT keyword in remote stored procedures, you must understand both remote stored procedures and the START AFTER COMMIT command.

A START AFTER COMMIT command may contain the optional clause:

```
FOR EACH REPLICA [WHERE ... ];
```

For example

```
START AFTER COMMIT FOR EACH REPLICA WHERE region = 'west'
UNIQUE CALL my_proc;
```

or, if you'd like the command to apply to all replicas:

```
START AFTER COMMIT FOR EACH REPLICA UNIQUE CALL my_proc;
```

The server that is executing this statement effectively creates a list of all the replicas that match the WHERE clause and then calls the specified procedure (`my_proc`) for each replica on the list. While that stored procedure is running, the DEFAULT clause identifies the replica that is currently being processed. For example, suppose that we have three replicas that match the WHERE clause, i.e. three replicas for which region = 'west':

California

Oregon

Washington

The `my_proc` procedure is then called three times. During the first call DEFAULT is equal to "California"; during the second call DEFAULT is equal to "Oregon"; and during the third call the DEFAULT is equal to "Washington". If the stored procedure named my_proc wants to call a remote stored procedure on each of the three replicas, it can do so by using the syntax:

```
CALL remote_proc_name AT DEFAULT;
```

Each time that `my_proc` is called, DEFAULT will be set to the name of one of the three replicas, so the effect will be to call each of the following:

```
CALL remote_proc_name AT California;
CALL remote_proc_name AT Oregon;
CALL remote_proc_name AT Washington;
```

If the "remote_proc_name" contains a command to refresh from the master, then the effect will be that all three replicas are notified that there is updated data and they should refresh. Thus by using a combination of

START AFTER COMMIT

and

remote procedure calls

we have been able to notify every replica that needs to refresh the data.

One way to make such maintenance easier is to use the "properties" feature described earlier. For example, suppose that you have three replica servers named California, Oregon, and Washington, all of which refresh from a particular publication. Now suppose that you want to add a new replica named Arizona and it should also refresh from the same publication and thus should be notified under the same circumstances as the servers California, Oregon, and Washington. You can simply set the property for that new server to match the property that the other three servers have (and which is used in the WHERE clause of the START AFTER COMMIT). For example,

```
SET SYNC PROPERTY region = 'west'; -- on the replica
SET SYNC PROPERTY region = 'west' FOR REPLICA arizona; -- on the master
```

When to Use Sync Pull Notify

The Sync Pull Notify feature decreases the delay between updating information on the master and updating information on the replica(s). However, in some situations it may increase network traffic.

Sync Pull Notify may increase the load on your network. If your current synchronization approach is to synchronize each replica once per hour, and if you typically have multiple updates per hour, then you only use 1 set of network messages per replica per hour to synchronize. If you switch to using the Sync Pull Notify feature, however, then you will have as many sets of network messages per replica per hour as you have updates.

Of course, the converse is also true. If you currently refresh frequently but only have updates infrequently, then Sync Pull Notify will actually reduce your network traffic because you will only refresh when you actually need data; you won't need to frequently "poll" to see if any data has changed. If you work in a situation where updates are infrequent, but it's necessary for you to know about those updates immediately when they occur, then Sync Pull Notify is a very good solution.

Note that "pull" and "push" (Sync Pull Notify) approaches are not mutually exclusive. You may use a combination of these. As an example, you might choose to design your system so that every day at a specific time the master will notify the replicas that it is time to refresh. However, a repair person going out into the field and taking her replica database with her could also issue a "REFRESH" command just before leaving the office, thus making sure that she has the most up-to-date data.

When designing your system for SyncPull Notify, you may find it helpful to know that there are only three ways that data in the master can be changed, and thus there are only three situations in which you might need to "push" updated data to the affected replicas:

1. Data may be changed directly on the master, that is, a client may insert, update, or delete a record in a table on the master.

2. The master may receive data from a replica.

3. If the master server is both a master and a replica (e.g. it is "in the middle" in a hierarchy that has three or more levels), then the server may request a refresh from its master and get data from that.

4.8.3 Scheduling REFRESH or Sync Pull Notify

In some cases, you may want to perform an operation such as a REFRESH or a Sync Pull Notify at regular intervals, for example, every 30 seconds. You may use the `SLEEP()` command to do this within a stored procedure. Below are 2 examples of performing tasks at regular intervals by using the `SLEEP()` command. Note that these examples would be implemented as stored procedures, and the stored procedures would probably be run in the background by getting called from the body of a START AFTER COMMIT command. Note that the parameter passed to the `SLEEP()` function is the desired duration (in milliseconds).

Here is a very simple example of using sleep:

```
CREATE PROCEDURE SIMPLE_CLOCK
RETURNS (T TIMESTAMP)
```

```
BEGIN
   -- Loop "forever".
   WHILE 1 LOOP
      T := NOW();
      RETURN ROW;
      EXEC SQL COMMIT WORK;
      SLEEP(1000);
   END LOOP
END
```

Here is an example of scheduling REFRESH commands by using `sleep()` in procedure code:

```
CREATE PROCEDURE REFRESH_SCHEDULER
BEGIN
   DECLARE I INTEGER;
   I := 0;
   WHILE I = 0 LOOP
      EXEC SQL COMMIT WORK;
      SLEEP(10000); -- here the procedure sleeps 10 seconds (10000ms)
      EXEC SQL EXECDIRECT call refresh_now;
   END LOOP
END
```

You can extend this approach to apply it to Sync Pull Notify.

The SLEEP function can be called from stored procedures. The duration is measured in milliseconds. Note that the duration is approximate. The resolution of the clocks and timers on your platform may not support millisecond accuracy. Also, the exact timing depends in part upon how busy the computer is. Furthermore, any SQL statement or procedure call that is executed in a START AFTER COMMIT statement is run asynchronously in the background and doesn't have very precise execution timing. Finally, the duration of the non-sleep activities also affects the timing. For example, if your loop contains a `SLEEP()` that lasts 10 seconds and an SQL statement that takes 2 seconds to run, then of course your loop will actually run approximately every 12 seconds, not every 10 seconds.

Chapter 5. Planning And Designing For SmartFlow Applications

This chapter describes the design and planning issues you need to consider before installing and implementing an application that uses solidDB SmartFlow data synchronization technology. Chapter 4, *Getting Started With Data Synchronization* provides you with a quick overview of SmartFlow functionality. Now you can begin planning for solidDB SmartFlow application development and customizing it to meet your own unique business needs. This chapter shows you how to plan and design your multi-database system to do this. It pinpoints the various areas of the application and database where design issues apply.

5.1 Planning for SmartFlow Installation

Before installing a distributed database system using solidDB SmartFlow, you need to determine, analyze, and evaluate the synchronization needs of your application. These needs affect the resource and application requirements of the system. In addition, performance considerations can affect how you decide to distribute data, initiate data propagation, schedule synchronization, create your infrastructure, and allocate computer and network resources.

5.1.1 Distributing Data

The amount and nature of local data needed at the replica affects the resource requirements of the synchronization process. For more scalability, plan to partition the data into different replica databases so that replica database contains only a subset of the master data. Typically the better the data is partitioned, the more scalability you achieve in your overall system. Be sure to consider performance needs when designing the logical and physical data model of the system.

5.1.2 Tailoring the Synchronization Process

A distributed SmartFlow system can utilize the off-peak hours of the system. The solidDB SmartFlow architecture allows the synchronization process to be fully tailored. Be sure to consider the capacity of the available infrastructure. For example, you can tailor large amounts of data transfer over the network when the available bandwidth is optimal. Whereas during rush hours, you can allow the transfer of only the most urgent synchronization tasks such as propagation of high-priority transactions.

Because there is some overhead related to synchronizing data, a compromise between the overall performance of the database and the data timeliness is often needed. The higher the timeliness requirement of the data (i.e. the smaller the synchronization messages are), the more overhead the synchronization causes and hence, the less scalability you have in the overall system.

5.1.3 Evaluating Performance and Scalability

When planning for performance and scalability, the infrastructure should provide enough capacity for I/O handling, fault tolerance, and synchronization message transfer. Each of the components that you need to consider for capacity planning that affect performance and scalability are described in this section.

Master Database

The master database is a critical component of the system. All synchronized transactions created in the system are eventually committed in the master database. Similarly, publication data is refreshed from the master database. From the system infrastructure point of view, this means two things:

- The capacity of the master server must be sufficient to manage the CPU and disk-I/O load caused by the replica transactions and refreshes. Some additional disk I/O is caused by the store and forward messaging of the synchronization architecture.

- The fault tolerance of the master server must be at a sufficient level. Since the replica databases communicate with each other only through the master, the master server is the single point-of-failure. If the master server goes down, synchronization between replicas stop. (You may want to consider using the Solid Carrier Grade option to mirror the master database server.)

Optimizing the Load of the Master Database

In a typical system, most of the database load is read I/O load caused by the read-intensive on-line usage of the database. In a multi-database system, this load can be distributed to a large number of databases. The capacity of the master database is then left for processing the transactions that have been propagated from the replica databases of the system.

Because all "shared" or synchronized transactions of the system are committed in the master database, it is very important that the resources of the master database are used as efficiently as possible. The following actions can help optimize the resource usage of the master database:

- If possible, dedicate the master database for synchronization use only. On-line access to this database may have unpredictable response times if heavy synchronization processes are being run simultaneously.

- Optimize the indexing of the database for synchronization use only. For example, if the database has no on-line usage, provide only those indices that are used by the search criteria and joins of the publications, as well as those needed by the transactions.

- If a database for centralized on-line use is needed, it is often preferable to have a full replica of the master database available for that purpose. This database can have indexing that is optimized for the on-line usage.

- Keep the publications simple. Complex publications with lots of joins between tables mean more complex queries that require more server resources.

  Since incremental refreshes usually use fewer resources than full refreshes, you can enable incremental refreshes by setting the synchistory property for the tables of the publication. This allows the master server to send only the master data that has changed in the publication, rather than a full publication. Read Section 6.5, “Creating Publications” for details.

- Do not synchronize more frequently than necessary.

- Utilize the off-peak hours in the synchronization processes. Synchronize the large masses of "less urgent" data when the on-line usage of the system is at minimum.

Replica Database

The usage pattern of replica servers of the system is usually fairly "traditional." The servers are accessed by applications that perform queries and write operations to the database. The capacity of the replica databases should be sufficient to serve the normal on-line usage of the server. Reserve some additional capacity to cover the overhead caused by database synchronization.

If possible, deny user access to the physical database file to ensure the maximum level of data security in the system.

Network

Be sure to place the master server on a machine that has the best possible throughput. Carefully estimate and test the maximum amount of data transferred during synchronization to ensure that the bandwidth of the network is sufficient for database synchronization.

Be sure to test the network for transmission of the synchronization messages. These messages contain:

- Header data (insignificant)
- Transactions which include:
  - procedure calls as strings
  - parameters as binary data
- Refreshes to publications which include:
  - all inserted and updated rows from the master database
  - primary keys of rows that are deleted from the master database

5.2 Designing and Preparing Databases for Synchronization

After you install solidDB on each machine as instructed in the release notes file in the solidDB Development Kit, you are ready to prepare and design your databases for synchronization. This requires the following tasks:

- Define master and replica databases.
- Create your database schema according to SmartFlow guidelines.
- Create catalogs if you have a multi-master environment or you are using different schema names in your master and replica database.
- Define concurrency conflict handling in synchronized tables, i.e. determine if the tables should use optimistic or pessimistic concurrency control.
- Provide user access required for synchronization.
- Set up backups of the master database and large replicas.
- Design application(s) for data synchronization.

Each of these topics is described in the following sections.

5.2.1 Defining Master and Replica Databases

Before you create your database schema, you need to set your database catalogs as a "master" or "replica" or both using the SET SYNC command. You can use SolidConsole or Solid SQL Editor (teletype) to enter the SolidConsole commands required for set up.

To specify a catalog as a dedicated "master" database, enter the following command in the catalog where the database resides:

```
SET SYNC MASTER YES;
COMMIT WORK;
```

To specify a catalog for a dual role (that is, a middle tier database of a multi-tier synchronization hierarchy), enter the following command in the catalog where the database resides:

```
SET SYNC MASTER YES;
```

```
SET SYNC REPLICA YES;
COMMIT WORK;
```

In each catalog where a dedicated replica resides, specify the catalog as a "replica" database:

```
SET SYNC REPLICA YES;
COMMIT WORK;
```

The current database catalog can be defined with the SET CATALOG command. If no catalog is specified, the base catalog is used.

5.2.2 Creating the Database Schema

In a multi-database system, the usage of databases can vary a lot. Therefore you must consider the way databases in your system will be used when physically implementing and tuning them.

Following are the guidelines for using schemas and catalogs. Refer to the section that applies to your SmartFlow architecture.

Guidelines for a Two-Tier Topology

A two-tier data redundancy model has one master database and multiple replica databases. Both master and replica databases can have different schemas using the default schema name which is the user id of the database owner. In this case, no schema is explicitly defined; instead the server automatically assigns one with the user id. It is recommended that you use identical schema names for the master and replica databases. Although you can use different schema names, be aware that different schema names in master and replicas may complicate the application programming.

To use schemas, a schema name must be created before creating the database objects that will be associated with the schema. To create a schema use the CREATE SCHEMA command. See the *solidDB SQL Guide* for details.

Guidelines for Multi-Tier Topology

A multi-tier topology contains three or more tiers in the hierarchy of synchronized databases. The top tier of the topology is the master database for the overall system. The mid-tier databases of the topology have a dual role of both master and replica databases. The lowest tier contains only replicas.

A multi-tier topology is especially useful in systems that have a wide geographic distribution and a potentially large number of replica databases that also have local data (i.e. data that does not require synchronization with the top-tier master). The data in this type of system is typically partitioned to limit data access to specific

replicas. For example, a network management system that contains a database to manage configuration and event information for a large managed network meets the criteria for a multi-tier topology.

Guidelines for Multi-Master Topology

solidDB's physical database file may contain more than one logical database. Each logical database is a complete, independent group of database objects, such as tables, indexes, procedures, triggers, etc. Each logical database is implemented as a database catalog. Each of these catalogs can act as an independent master or replica database. This makes it possible, for example, to create two or more independent replica databases into one physical local database. It is also possible to have one or more catalogs in this same local database that each contain a master database.

Multi-master topologies are useful in environments where a solidDB database is used by multiple applications. For example, a local database may contain a replica of two masters: one for a configuration management application and another one for a usage-monitoring application. Note that you can combine multi-tier and multi-master topologies.

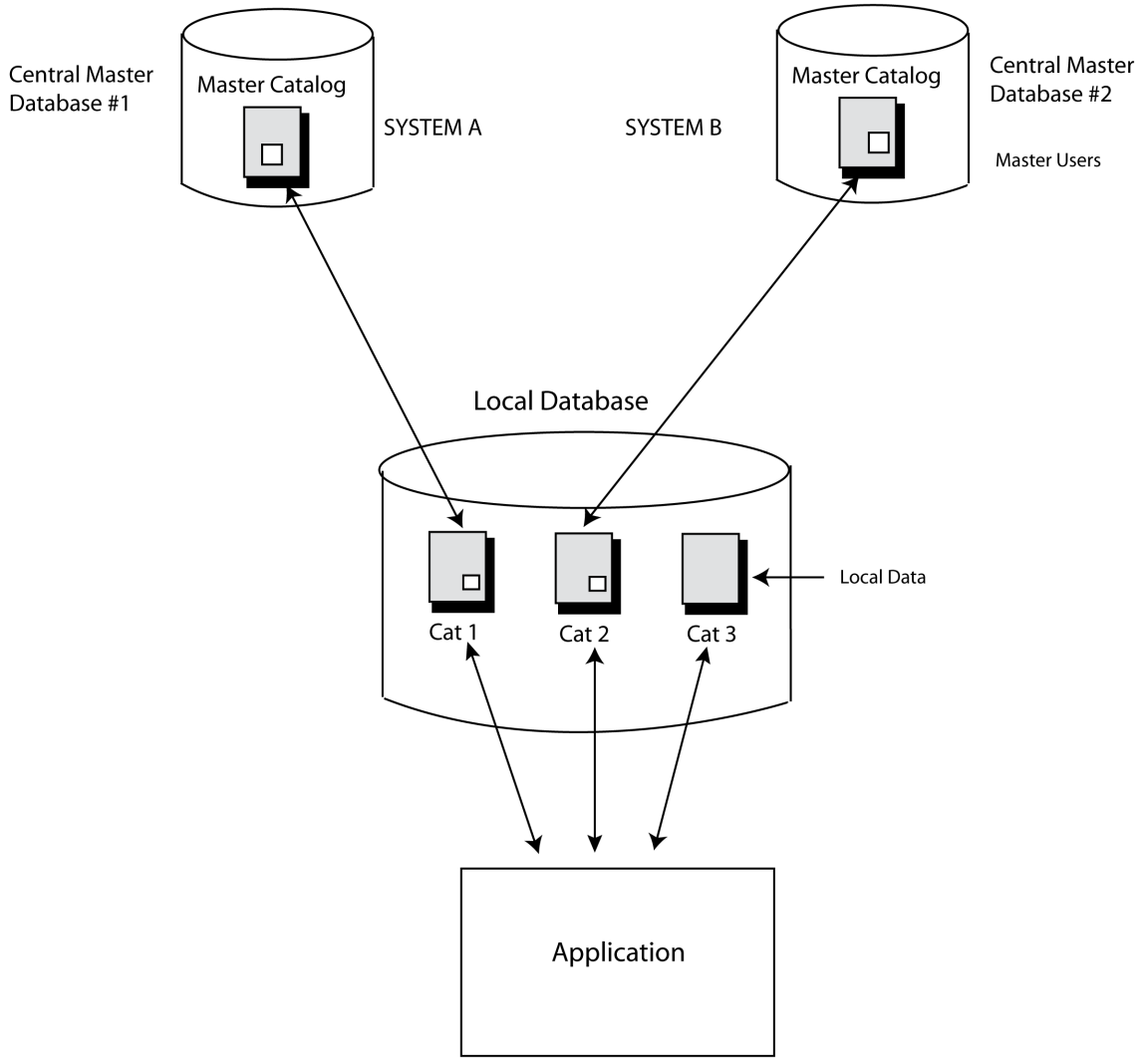
Creating Catalogs

The following are guidelines for designing and implementing multiple catalogs used for synchronization:

- When creating a database, solidDB creates a default catalog for the database.
- Besides the default catalog, a single solidDB database can contain any practical number of catalogs.
- If you do not explicitly CREATE and SET any catalogs, then you will use the default catalog for the database.
- Each catalog of a database can be either a master, replica, or both.
- Each catalog can contain multiple schemas. Transactions can access database objects in any catalog.
- A catalog can contain local tables as well as tables that are synchronized with a master. A single transaction may use a combination of local tables and master tables.
- The physical database has a set of defined local users that have access to the local data management functions. For accessing the data synchronization functions, each catalog has one or more master users that have been downloaded as part of a replica registration.

Figure 5.1, “Multi-Master Model” illustrates these guidelines.

Figure 5.1. Multi-Master Model



To create and set catalogs for masters and replicas, use the CREATE CATALOG and SET CATALOG commands. Refer to the CREATE CATALOG and SET CATALOG commands in the *solidDB SQL Guide* for details on creating and setting catalogs.

On Master:

```
CREATE CATALOG INVENTORY;
SET CATALOG INVENTORY;
COMMIT WORK;
```

On Replica:

```
CREATE CATALOG INVENTORY;
SET CATALOG INVENTORY;
COMMIT WORK;
```

Note

1. A catalog name does not need to be the same in a master and replica. want to use (if you do not specify a fully qualified name).

2. When actually using the catalogs after they are created, you may specify them by using fully-qualified table names (including the catalog name), or you may use the SET CATALOG command to specify which catalog you want to use (if you do not specify a fully qualified name).

Using Schemas within Catalogs

To logically partition a database, you create a catalog(s) first before you create a schema. After you create the catalog and schema, you then create the database objects that will be associated with the schema. If you create the database objects without a specified schema, the schema becomes your user id.

You may use multiple schemas within catalogs (although a single schema may be sufficient). If you have multiple schemas, you may specify them either by including the schema as part of the table name, or you may use the SET SCHEMA command to specify which schema you want to use.

If you do not explicitly CREATE and SET any schema name, then you will use the default schema name, which is your user id.

Note

With catalogs, there is one default for the entire database, but with schemas, there is one default for each user — there is NOT one default for the entire database.

Refer to the CREATE SCHEMA and SET CATALOG commands in the *solidDB SQL Guide* for details on creating and setting schemas.

Set up Data for Synchronization

This section applies to both two-tier, and multi-tier, multi-level architectures. It assumes that you have created your catalogs and schema names (if required).

The following are guidelines for designing and implementing the schema and using the CREATE TABLE command to create the master database and replica database tables.

You define tables that are required for synchronization and will be used in a publication. A publication is a set of data to be downloaded from the master database to a replica database. When creating your schema you need to define:

- Tables of the master database
- Tables of the replica database
- Replica databases can contain all tables of the master database or a subset of them.
- Replica databases can also contain tables that are for local use only.
- Replica tables can contain a subset of columns from the master table.
- The name of the replica table can be different from the master table. When publications are created using the CREATE PUBLICATION command, a master table name can be associated with a replica table that has a different name. The publication definition takes care of the mapping between the master and replica tables.

Keep in mind the following when creating tables:

- All tables in the schema must have a user-defined primary key. The primary keys in master and replica tables must be identical and uniqueness must be guaranteed globally. More columns in a replica's primary key will lead to conflicts in propagating transactions to the master database. More columns in a master's primary key will similarly lead to conflict when refreshing data to a replica.
- Apply the ALTER TABLE SET SYNCHISTORY command to enable incremental publications on the master and replica tables. Otherwise the master sends the replica full publications (which use more resources) rather than incremental publications.

By setting the SYNCHISTORY property for each table of the publication in both the master and the replica databases, you allow the creation of a shadow history table that keeps track of data updates for incremental publication. For details on the ALTER TABLE syntax, read Section 6.5.1, “Creating Incremental Publications”.

Design the Logical Database

The data modeling of a multi-database system is slightly different from that of a centralized system. The tentative nature of replica data as well as the possible co-existence of multiple different versions of the same data item (row) must be handled properly in the logical data model. Therefore, there are some rules of thumb to consider when designing the logical database of a multi-database system.

Unique Surrogate Primary Keys

All write operations that are executed in the master database must be successful. A "unique constraint violation" causes the entire synchronization process to halt. Therefore the primary keys (and unique indices) of the rows must be unique throughout the entire system. It is strongly recommended, that globally unique, surrogate primary key values are used in all tables of the database. This kind of key can for instance be a combination of database ID and a sequence number. For example:

```
CREATE TABLE CUST_ORDER
DB_ID VARCHAR NOT NULL, ID INTEGER NOT NULL,
... other columns of the table,
(PRIMARY KEY (DB_ID, ID));
```

Detecting Update Conflicts

The easiest method of detecting update conflicts is to use an "updatetime" column in each table of the system. Whenever a row is updated, the current (that is, pre-update) value of the updatetime column is appended to the WHERE clause. If the row is not found, it means that the updatetime has changed, that is, someone else has updated the row causing a conflict. This mechanism is known as "optimistic locking."

Reporting Synchronization Errors

It is always possible that an error can occur during synchronization. The error may be an application-level error such as update conflict in the master database. If such error cannot be automatically resolved, it should be logged for manual resolution.

One way to implement this is to create a synchronization error log table that contains an entry about each error. Whenever an application-level error occurs during synchronization, the stored procedures of the transactions should insert a row to the error log table. For a suggestion on how to create an error log, read Section 6.6.5, “Creating a Synchronization Error Log Table for an Application”.

The error may also be server-level error that halts the processing of the synchronization message. In this case, the error recovery is done at messaging level. For details, please refer to Chapter 8, *Administering solidDB with SmartFlow*.

Design the Database Schema

The usage pattern of master and replica databases can be very different. The queries performed in the master database can also be very different from those of replica databases.

Due to these facts, the indexing of the databases should be carefully designed, bearing in mind the different usage patterns. The indexing of replicas should follow the requirements of the applications that are using the database. When designing an index for a master database, note that unique indexes must be globally unique. Also consider the following index guidelines for publications and transactions.

Publications

Be sure to index queries that are derived from publication definitions. solidDB treats nested publications as joins. To make the publication operation efficient, the joining columns of the tables of the publication must be indexed.

In the example that follows, CUSTOMER and SALESMAN tables are joined together using the SALESMAN_ID column of the CUSTOMER table. To allow efficient execution of refreshes from this publication, index the SALESMAN_ID column using a secondary index.

```
CREATE PUBLICATION pub_customers_by_salesperson (sperson_area varchar)
BEGIN
   RESULT SET FOR salesman
   BEGIN
      SELECT * FROM salesman where area = :sperson_area
      DISTINCT RESULT SET FOR customer
      BEGIN
         SELECT * FROM customer WHERE salesman_id = salesman.id
      END
   END
END;
```

In addition to the indices created by the user, solidDB automatically creates two system indices for tables that have the SYNCHISTORY property set on. The same indices are also automatically created for the history tables of the main tables.

Write Load Caused by Transactions

The write load of the master database sets the practical limits to the scalability of a SmartFlow system. In a SmartFlow system, all propagated transactions are eventually committed in the master database. Each index causes additional disk I/O in all write operations (insert, update, delete) to that table. For this reason, minimize the number of secondary indexes in the master database if the write performance is a critical factor.

5.2.3 Defining a Database Table

The following CREATE TABLE SQL command creates a table that is typical in a database set up for synchronization. Note also that the ALTER TABLE SET SYNCHISTORY SmartFlow extension prepares the table for incremental publications. Be sure also to set up the table for incremental publications if you are using large tables, as opposed to tables that are small and heavily updated.

Note

Before you execute the ALTER TABLE SET SYNCHISTORY statement, be sure you have defined your databases to be masters and replicas using the SET SYNC MASTER and SET SYNC REPLICA commands. Failure to define masters and replicas results in an error message when you attempt to use the ALTER TABLE command. Read Section 6.4, “Setting Up Databases for Synchronization” for details.

Note

If the Replica is read only (no changes are done to the replicated parts of the publication), the statement ALTER TABLE ... SET SYNCHISTORY is not needed. In the same time, the following Flow Replica-resident parameter should be set:

```
set sync parameter SYS_SYNC_KEEPLOCALCHANGES 'Yes';
```

Note also that, in this case, ALTER TABLE ... SET HISTORY COLUMNS cannot be used.

```
CREATE TABLE CUST_ORDER (
ID VARCHAR NOT NULL,
SYNC_STATUS CHAR(3) NOT NULL,
CUST_ID VARCHAR NOT NULL,
PRODUCT_ID VARCHAR NOT NULL,
QUANTITY INTEGER NOT NULL,
PRICE DECIMAL(10,2) NOT NULL,
UPDATETIME TIMESTAMP NOT NULL,
PRIMARY KEY (ID, SYNC_STATUS));
ALTER TABLE CUST_ORDER SET SYNCHISTORY;
COMMIT WORK;
```

Some remarks about the above example:

- The ID column is a generated primary key value (surrogate key) of the new row:

```
ID              VARCHAR NOT NULL,
```

The value should preferably be a composite that contains two parts: the unique ID of the database where the row was first created and a sequence number within that database.

The reason for recommending usage of surrogate keys is the requirement of global key uniqueness. Inserting a row in two different replica databases with the same key value must *not* be permitted. If allowed, the next transaction propagation task would produce a unique constraint violation error in the master database. Whenever such an error occurs, the synchronization process halts and cannot continue until the problem has been fixed by deleting the duplicate row from the database.

- The SYNC_STATUS column holds information about the synchronization status of the row:

  ```
  SYNC_STATUS   CHAR(3) NOT NULL,
  ```

  If the row is a valid one, then the value can be, for example, "OK". On the other hand, if an update conflict or other transaction validation error has happened when the row was committed in the master database, then the row should be inserted to the database with for example, value "C01" (first update conflict of this row). The existence of this column makes it possible to store multiple versions of the same row to the database: one official version (with status "OK") and multiple additional versions for conflict resolution and error recovery purposes. The primary key of the table is composed of the ID and SYNC_STATUS columns.

- The UPDATETIME column contains a timestamp that indicates the last date and time when the row was updated:

  ```
  UPDATETIME TIMESTAMP     NOT NULL,
  ```

  The application logic (including the stored procedures that form the transactions) can use this column to detect update conflicts in the system.

5.2.4 Handling UPDATE Triggers

In some situations, UPDATE triggers require special design and coding.

You might expect that if you UPDATE a record on the master, and if that UPDATE operation fires an UPDATE trigger, then when the record is sent to the replica (via a REFRESH), an UPDATE trigger will also be fired on the master. In fact, however, this is not the case. If a record is updated on the master, and the same record is "updated" via a REFRESH operation on the replica, then the replica server will actually fire a DELETE+INSERT pair of triggers, not an UPDATE trigger. The reason for this is that the replica does not directly update the record; instead it deletes the old record and then inserts the new record.

Description of the Possible Causes of Triggers

The server can also do an additional pair of insert-delete operations that is related to synchronization and that causes triggers to be fired. Prior to copying the master data to the replica's tables, the replica "undoes" all local data changes that have been written into the replica since the last synchronization. The reason for this is that the replica's data is always "tentative" until it has been processed and made official by the master. If the replica has tentative data, then each time that it receives official data from the master, it (the replica) simply discards any unofficial data that it has and stores only the official data from the master. The process of "undoing" the tentative changes on the data causes the replica server to do an additional delete and insert. We explain this in more detail below.

If the REFRESH is incremental, then the process is executed in multiple steps. The first step is effectively to throw out all tentative changes made since the last official data was received, then restore the last official data. The second step is to process the new official data. Thus we have all the old official data as of the last refresh, and we have all the official changes since that refresh, so we have exactly what the master has.

Let's step through an example, in which we get a full refresh from the master, make some changes on the replica, and then get an incremental refresh from the master. During the process of getting the refresh(es) from the master, we throw out all the data on the replica, and store all the data sent by the master.

At 10:00 AM we get a full refresh with data records for Anne Anderson, Barry Barrymore, and Carrie Carlson.

At 10:01 AM Anne Anderson updates her record.

At 10:02, Barry updates his record.

At 10:03, we propagate the changes, in this case the change to Anne's and Barry's records. For the purpose of this example, assume that Anne's changes are accepted by the master and Barry's changes are rejected.

At 10:04, we get an incremental refresh. During the processing of the incremental refresh, the replica discards all changes since the last official refresh (in this case the one at 10:00 AM). Both Anne's changes Barry's changes are discarded from the replica.

At this point, our replica looks just as it did after the last refresh operation and thus is ready to get an incremental subscription that contains only those changes approved by the master since the last refresh.

During our incremental refresh, we get an updated record for Anne (whose propagated data was accepted by the master and then returned during the REFRESH). Barry's changes, which were rejected by the master, are gone forever. The net result is that the database has the official values for all 3 records - the new/approved changes for Anne, and the older (most recent official) data for Barry and Carrie.

(Note that the preceding description uses some simplifying assumptions, including that Barry and Carrie's records were not changed on the master since the last official refresh by this replica.)

Now let's give an example of a complete cycle, starting with an update on the replica and ending with the refresh from the master, so that we can see all the triggers that may be executed during this cycle. In our example, a replica makes a local update that is propagated to the master and executed there, and then the master version of the update is downloaded back to replica in the result set of the synchronization. During this cycle, triggers are fired in master and replica databases as shown below:

1. The update on the replica fires regular UPDATE triggers (both the BEFORE and AFTER triggers) in the replica. If the replica contains any old master values (which it probably does unless it is brand new and has never done a REFRESH), those data values from the master are written to the history table in the replica.

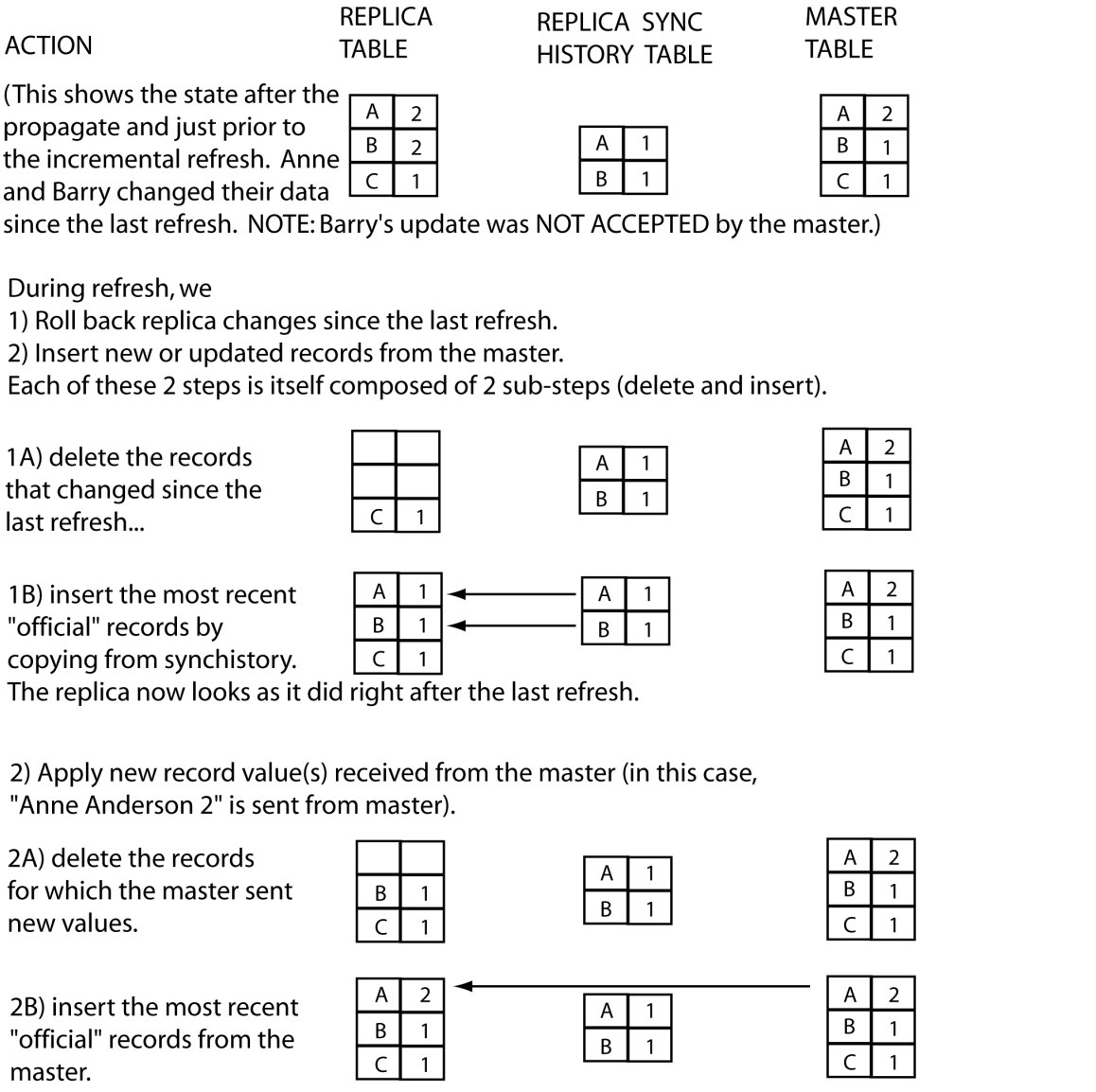
2. The replica then propagates its data to the master. When executing the propagated transaction in the master, UPDATE triggers are fired in the master. The old value of the row to be updated is written to the sync history table of the master database. The new value of the row is of course written to the main table.

3. After the replica requests a refresh, the master server assembles the result set of the subscription. It contains the primary key values of the old versions of updated rows. These rows will be deleted from the replica table. After those rows are deleted, the new values of the same rows will be inserted. (The result set that is sent to the replica will contain first a list of primary keys to be deleted, followed by the rows to be inserted.)

4. When the replica receives the sync message from the master, it deletes all "tentative" data from the main tables. This fires a delete trigger (Delete local tentative row). After this, the replica server inserts the possibly existing old master version of the row from the sync history table. This fires an insert trigger (Insert old official row). After this, the server applies the deletes from the synchronization message. This fires a delete trigger (Delete old official row). Finally, the server applies the inserts from the synchronization message. This fires an insert trigger (Insert new official row).

See Figure 5.2, “Database, Catalogs, and Schemas” for an illustration of step 4 above. The illustration shows a detailed breakdown of what occurs during an incremental refresh. In an earlier diagram in Figure 2.1, “Propagate And Refresh”, we showed the overall process of propagating and refreshing a record. The illustration below shows the details of just the incremental refresh portion of the earlier diagram.) As you can see, for the record updated by "Anne", the replica server actually executes multiple delete and insert operations. The record updated by "Barry" is also deleted and then inserted (rather than simply updated).

Figure 5.2. Database, Catalogs, and Schemas



When synchronizing an update operation that may have taken place in both replica and master databases, we have up to four sync-related triggers that may fire on the replica:

- Delete current tentative
- Insert old official
- Delete old official
- Insert new official

(In addition, some trigger(s) on the master may also have fired, of course.)

Now that you understand all the possible triggers that can occur in a SmartFlow operation when an UPDATE occurs on the master, you will understand all the possible values that the new bulletin board parameters can take. The possible values are fully documented in Appendix A, *Bulletin Board Parameters*, and are summarized below:

The possible values of the *SYS_SYNC_OPERATION_TYPE* parameter in DELETE triggers are:

- CURRENT_TENTATIVE_DELETE (set when deleting the current locally updated value of a row prior to executing the reply message in replica)
- OLD_OFFICIAL_DELETE (set when deleting a row that was deleted in master)
- OLD_OFFICIAL_UNIQUE_DELETE (set when the master sends a row to be added to the replica, but a similar row already exists on the replica. With this parameter value, the old official row is deleted before the new row is added to the replica.)
- OLD_OFFICIAL_UPDATE (set when executing a delete that was created as a result of an update in master)

The possible values of the *SYS_SYNC_OPERATION_TYPE* parameter in INSERT triggers are:

- OLD_OFFICIAL_INSERT (set when restoring the old master value prior to executing the reply message in replica)
- NEW_OFFICIAL_INSERT (set when inserting row that was inserted in master)
- NEW_OFFICIAL_UPDATE (set when executing an insert that was created as a result of an update in master)

If the trigger is fired by a local transaction (i.e. not by synchronization logic), then the value of this parameter is NULL.

The above is valid only for incremental refreshes. If the result set is full, all local data is deleted and a full set of master data is applied. In this case the DELETE operations from the master database are not sent to the

replica. To find out whether the refresh was full or incremental read the value of the bulletin board parameter *SYS_SYNC_RESULTSET_TYPE*. Its possible values are:

- FULL
- INCREMENTAL

Executing the Code Intended for the UPDATE Triggers

Unfortunately, it is not trivial to ensure that an UPDATE operation on the master is executed as an UPDATE on the replica. Instead of directly trying to force an UDPATE trigger on the replica to be executed, the solidDB server (the master) now posts two parameters on the SmartFlow parameter bulletin board. These parameters can be read by the trigger logic on the replica to determine how the record was originally processed on the master. The replica can then execute the logic appropriate for an UDPATE operation, if desired.

The bulletin board parameter(s) tell you how the record was originally processed — e.g. whether it was originally part of an UDPATE operation on the master. Of course, since an INSERT or DELETE trigger has no easy way to stop itself and then force the firing of an UPDATE trigger, merely knowing that the original command was an UPDATE trigger isn't sufficient. You will need to modify the code inside your triggers so that they take different actions depending upon what the original statement was (e.g. whether the original statement on the master was an update or was a DELETE/INSERT pair). If necessary, your DELETE or INSERT trigger will call the same code that your UPDATE trigger would have called. You should do the following on the replica(s):

For each of the six possible triggers on a table

```
BEFORE UPDATE,
AFTER UDPATE,
BEFORE INSERT,
AFTER INSERT,
BEFORE DELETE,
AFTER DELETE,
```

extract all of the trigger's logic into a stored procedure that can be called from inside a trigger.

Thus, for example, if your BEFORE UPDATE trigger looked like:

```
CREATE TRIGGER trigger_name ON table1 BEFORE UPDATE
   BEGIN
   stmt1;
   stmt2;
```

```
   stmt3;
  END;
```

You will now have something like:

```
CREATE PROCEDURE before_update_on_table1
   BEGIN
   stmt1;
   stmt2;
   stmt3;
   END;
CREATE TRIGGER trigger_name ON table1 BEFORE UDPATE...
   BEGIN
   CALL before_update_on_table1;
   END;
```

Once you've done this, you can rewrite your triggers so that they take into account whether the record was originally UPDATED or was originally INSERTED. For example, the logic of your BEFORE INSERT trigger will look something like

```
CREATE TRIGGER trig1 ON table1 BEFORE INSERT...
 BEGIN;
 IF (get_param('SYS_SYNC_OPERATION_TYPE') = 'NEW_OFFICIAL_UPDATE') THEN
  IF (get_param('SYS_SYNC_RESULTSET_TYPE') = 'INCREMENTAL') THEN
   CALL before_update;
  ELSE ...
  END IF;
 ELSE IF (get_param('SYS_SYNC_OPERATION_TYPE') = 'NEW_OFFICIAL_INSERT') THEN
  CALL before_insert; - do what I normally do in my INSERT trigger.
 ELSE ...
 END IF;
 END;
```

Moving the body of your trigger(s) to a stored procedure(s) is relatively easy, since triggers and stored procedure language are essentially the same.

Warning

The "Before" values from the master are not available, even if you use the "BEFORE" keyword in the trigger(s) on the replica.

5.2.5 Handling Concurrency Conflict in Synchronized Tables in Replica

solidDB uses the same concurrency control mechanism to handle such data management functions as online queries and write operations. As a default method of concurrency control, optimistic concurrency control is automatically set for all tables. This means that if two users concurrently attempt to modify the same data, the later attempt fails and an error is returned to the user.

During synchronization, concurrency conflicts can occur through a sequence of events as shown in this example:

1. A replica creates and executes a synchronization script that makes a refresh from a publication.
2. Simultaneously, another user in the replica updates a row that will be refreshed.
3. Before the user commits the transaction, the reply message of the synchronization message arrives at the replica and the engine starts applying the refresh data to the database.
4. The user commits the on-line transaction.
5. The refresh attempts to modify the same row that the on-line user already modified.
6. The execution of the synchronization reply message fails because of a concurrency conflict.

The following table shows you the various ways you can handle a concurrency conflict reflected in this example or a similar situation.

Table 5.1. Handling a Concurrency Conflict

Criteria and/or Method of Recovery	Use this Command for Recovery
If you do not anticipate concurrency conflicts to happen often, then you can recover from this incident by re-executing the failed reply message in a replica.	The command for re-executing the failed reply is: `MESSAGE msgname EXECUTE`
If you anticipate concurrency conflicts to happen often and the re-execution of the message fails because of a concurrency conflict, you can execute the message using pessimistic table-level locking; this ensures the message execution is successful.	The command for executing the message in pessimistic mode is: `MESSAGE msgname EXECUTE PESSIMISTIC`

Criteria and/or Method of Recovery	Use this Command for Recovery
In this mode, all other concurrent access to the table affected is blocked until the synchronization message has completed.	
You can define the reply message to use table-level pessimistic locking when it is initially executed.	The command for requesting the reply message in pessimistic mode from the master is: `MESSAGE msgname GET REPLY PESSIMISTIC`

As part of the MESSAGE FORWARD operation, the reply message can use table-level pessimistic locking when it is initially executed.	The command for requesting the reply message in pessimistic mode is:
	
	`MESSAGE msgname FORWARD`
	`TIMEOUT seconds PESSIMISTIC`
	
	The synopsis has been placed on two lines for layout purposes. The command is entered as one line.

solidDB also allows you to define a table to be pessimistically locked using row-level locking. This approach is useful if lots of conflicting updates are expected on the table.	The command for setting a table to use pessimistic locking is: `ALTER TABLE tablename SET PESSIMISTIC`

5.3 Determining User Access Requirements

After you create your database schema, you need to determine:

- The local users on each replica database that need authorization to use specific tables in a publication, as well as execute rights to procedures they need to execute.
- The master users that need authorization to manage synchronized data in both the master and replica databases. For example, master users who define publications must have access rights to tables referenced by the publications. This also allows them to drop the publication.
- The master user of the system who requires rights to perform synchronization operations through the SYS_SYNC_ADMIN role created specifically for administrative tasks.

- The master or local users who require rights to register replica databases for synchronization through the SYS_SYNC_REGISTER_ROLE.

Read Section 6.3, “Implementing Security through Access Rights and Roles” for details on implementing synchronization access.

5.4 Creating Backups for Fault Tolerance

The critical component of a SmartFlow system is the server that hosts the master database. After you create the master database and its schema, you should use a reliable storage medium, such as tape or CD-ROM, to ensure availability of multiple backup versions. Normal backup procedure tasks you use on standard databases also apply to solidDB databases. For details, see Section 8.3, “Performing Backup and Recovery”.

In addition, you can make the master database fault tolerant by using RAID disks and hardware clustering in your system.

Replica databases can be reconstructed from the master database by refreshing data from the master database to a new replica. Large replicas should be backed up separately using normal Solid backup procedures to ensure quick recovery from disaster situations.

When your SmartFlow system is fully implemented and is sending and receiving transactions and messages, be sure to perform periodic backups to the system. Read Section 8.3.3, “Backing up and Restoring the Master and Replica Databases”.

5.5 Designing the Application for Synchronization

What sets solidDB SmartFlow apart from traditional data replication solutions is its principle of building data synchronization functionality inside the application. The following sections describe some considerations in the functionality of the client application.

5.5.1 Providing a "Tentative Data" Status on the User Interface

The tentative nature of replica data means, in practice, that a transaction that is committed in a replica, may be changed in the master database. Sometimes this has visible implications to the application itself. For instance in an order-entry system, the status of an order can first be "Tentatively OK" which means that it has been accepted by the replica, but not yet by the master. It may be appropriate to also show this tentative status to the user.

5.5.2 Providing a User Interface to Manage Synchronization

In a stand-alone replica database, the data contents of the database may vary. Data can be downloaded to and deleted from the local database based on the user's need. For instance, a salesman may need customer information for the western region today and the same data about the eastern region tomorrow. To be able to dynamically populate the replica database, a user interface may be needed for refreshing new (and existing) data as well as deleting unnecessary local data by dropping the corresponding subscriptions.

Managing the Synchronization Process

The synchronization process of solidDB SmartFlow architecture needs to be implemented at the application level. The synchronization process management may contain the following tasks:

- Define the contents of the synchronization process, that is, define which transactions are propagated to the master and which publications are refreshed to the replica in a single synchronization message.
- Execute the process, that is, send the request message and get the reply message back. Depending on the application need, these steps can be executed as one "synchronous" or two "asynchronous" operations.
- Monitor the status of the process.
- Resolve the system- and application-level errors that may have occurred during synchronization.

You can design these tasks through the user interface or by an automatic process. For example, you can implement a user interface into your application for manual monitoring and execution of these tasks. Alternatively, you may want to fully automate the tasks inside your application so that no user interaction is necessary.

During synchronization, errors can occur. These errors can be either at the application or at the system level.

Application-Level Errors

These are errors that occur when the validation logic of a transaction detects an error and manual actions are necessary to fix the situation. For example, if an "insert order" transaction of an order-entry application detects that the customer credit limit has been exceeded in the master database, manual approval to the order is required. Tracking and resolving the errors typically requires an application-level error log table that can be viewed from the client application.

System-Level Errors

These errors are typically failures in the store-and-forward messaging. For instance, the network may be down when synchronization is being attempted. Due to this, it is important, that proper error handling is implemented in the execution of the synchronization process. The information that is required to monitor and manage the synchronization process is available in the system tables of solidDB.

5.5.3 Providing Intelligent Transaction Based on Application Needs

Solid Intelligent Transaction is an extension to the traditional transaction model. It allows developers to implement transactions that are capable of validating themselves in the current database and adapting their contents (if required) according to the rules of the transaction.

Users create transactions in replica databases. These transactions are tentative since they have yet to be committed to the master database, which contains the "official" version of data. The replica transactions are saved for later propagation to the master database. In this model, transactions are long-lived and there can be multiple instances of a data item in different databases of the system.

When a replica transaction is propagated to the master database, transaction validation errors such as update conflicts may occur. Transactions must respond in such a way that meets the business rules required for the application. This is the best way for ensuring database consistency and reliability.

Developers need to evaluate the business rules required and build transactions based on SmartFlow's easy-to-use model. Read Section 6.6, “Designing and Implementing Intelligent Transactions” for details on creating transactions with Solid Intelligent Transaction.

Chapter 6. Implementing a solidDB SmartFlow Application

This chapter describes the basic tasks required to implement synchronization. These tasks, demonstrated briefly in Chapter 4, *Getting Started With Data Synchronization*, are described in more detail in this chapter. You are also introduced to solidDB SmartFlow synchronization statements, which are SQL extensions that allow you to set up and define your SmartFlow installation for data synchronization.

The topics in this chapter include:

- Using SmartFlow statements
- Implementing synchronization messages
- Implementing synchronization access rights and roles
- Specifying databases for synchronization
- Creating publications and subscriptions
- Implementing Solid Intelligent Transaction

6.1 Using SmartFlow Data Synchronization Statements

SmartFlow synchronization statements are extensions to Solid SQL. They allow you to manage distributed data by letting you specify synchronization operations. You can execute SmartFlow statements using Solid-Console in batch or interactive mode, or Solid SQL Editor (teletype). Read the *solidDB SQL Guide* for more details.

6.1.1 Types of SmartFlow Statements

SmartFlow data synchronization statements let you perform such tasks as registering replica databases, implementing access rights, and creating publications. In addition, you administer the SmartFlow functionality using many of these statements. For example, you use MESSAGE commands to create and manage synchronization messages and DROP commands to remove synchronization objects such as publications and subscriptions.

The SmartFlow statements are grouped into the following categories and their usage is described in this chapter. Refer to the *solidDB SQL Guide* for an alphabetical list of all SQL statements, including those used in SmartFlow operations.

Database Configuration Statements

These statements are used to set up and configure the databases used in a SmartFlow system.

```
DROP MASTER
DROP REPLICA
REGISTER REPLICA
SET SYNC master_or_replica
SET SYNC master_or_replica
SET SYNC CONNECT
SET SYNC NODE
SET SYNC PARAMETER
SET SYNC USER
```

Security Statements

These statements are used to set up security for multi-tier and multi-master environments.

```
ALTER USER SET MASTER
ALTER USER SET PUBLIC | PRIVATE
```

Publication Statements

These statements are used to create and maintain publications, as well as refresh data from them.

```
ALTER TABLE SET { SYNCHISTORY | NOSYNCHISTORY }
CREATE SYNC BOOKMARK
CREATE PUBLICATION
DROP PUBLICATION
DROP PUBLICATION REGISTRATION
DROP SUBSCRIPTION
DROP SYNC BOOKMARK
EXPORT SUBSCRIPTION
GRANT REFRESH ON
IMPORT
MESSAGE APPEND REGISTER | UNREGISTER PUBLICATION
```

```
MESSAGE APPEND REFRESH
REVOKE REFRESH ON
```

Intelligent Transaction Control

The functions and statements below are used to control the execution of intelligent transactions:

- saving transactions in the replica database for propagation to the master,
- defining bulletin board parameters (for example to control which transactions are propagated in a particular message),
- setting parameters on the parameter bulletin board within a transaction, and
- reading parameters from the parameter bulletin board.

```
GET_PARAM( )
PUT_PARAM( )
SAVE
SAVE PROPERTY
```

Message Statements

The MESSAGE statements are used to create and execute the synchronization messages that are sent between a replica and the master database.

```
MESSAGE APPEND PROPAGATE TRANSACTIONS
MESSAGE APPEND REFRESH
MESSAGE APPEND REGISTER | UNREGISTER PUBLICATION
MESSAGE APPEND REGISTER | UNREGISTER REPLICA
MESSAGE APPEND SYNC_CONFIG
MESSAGE BEGIN
MESSAGE DELETE
MESSAGE DELETE CURRENT TRANSACTION
MESSAGE END
MESSAGE EXECUTE
MESSAGE FORWARD
```

```
MESSAGE FROM REPLICA EXECUTE
MESSAGE GET REPLY
```

6.2 Building Messages for Synchronization

The data synchronization between two solidDB servers relies on the solidDB SmartFlow messaging architecture. This is a store-and-forward messaging architecture that is built inside solidDB. It is capable of transferring messages reliably between a replica and the master database.

The synchronization process of SmartFlow architecture consists of two different tasks:

- *propagating transactions* to the master database
- *refreshing of publications* to a replica

A combination of these tasks (containing the propagation command, REFRESHes, or both) is grouped together in a synchronization message. Note that transactions referring to a particular table should always be propagated to the master before refreshing data from that table. Both of these actions are permitted in the same synchronization message as shown in the example below:

Example 6.1. Synchronization Messaging

```
MESSAGE my_msg BEGIN ;
MESSAGE my_msg APPEND PROPAGATE TRANSACTIONS ;
MESSAGE my_msg APPEND REFRESH ORDERS_BY_SALESPERSON ('1') ;
MESSAGE my_msg APPEND REFRESH PARTS_IN_INVENTORY ;
MESSAGE my_msg END ;
COMMIT WORK ;
```

6.2.1 Beginning Messages

You must explicitly begin each synchronization message that is sent from the replica to the master database with the MESSAGE BEGIN statement. The syntax is:

```
MESSAGE unique_message_name BEGIN [TO master_node_name]
```

For the message, provide a name that is unique within the replica database. Be sure to also set autocommit to off. Any open transaction must be committed or rolled back in the connection before executing this statement.

Note

Use the optional TO clause if you want to send a message that contains the REGISTER REPLICA command and are registering a replica with a master that resides in a database catalog other than the default catalog. For details on registration, read Section 6.4.2, “Registering Replicas with the Master Database”.

If you want to create and execute a synchronization message from a stored procedure, here's an example of how to create a synchronization message with a unique message name.

```
DECLARE Autoname VARCHAR;
DECLARE MsgBeginStr VARCHAR;
Autoname := GET_UNIQUE_STRING('MSG_') ;
MsgBeginStr := 'MESSAGE ' + autoname + ' BEGIN';
```

Once you have composed the SQL statement as a string, you can execute it inside a stored procedure in one of two ways — either by using the EXECDIRECT feature, or by preparing and executing the SQL statement.

```
EXEC SQL EXECDIRECT MsgBeginStr;
```

or

```
EXEC SQL PREPARE cursor1 MsgBeginStr;
EXEC SQL EXECUTE cursor1;
EXEC SQL CLOSE cursor1;
EXEC SQL DROP cursor1;
```

6.2.2 Propagating Transactions from Replica to Master

The MESSAGE APPEND PROPAGATE TRANSACTIONS statement lets you propagate transactions from a replica to the master database. Only statements that have been explicitly put to the replica database's transaction queue with the SAVE <sql-statement> statement, can be propagated. The syntax is:

```
MESSAGE unique_message_name APPEND
```

```
[PROPAGATE TRANSACTIONS [WHERE {property_name {=|<|<=|>|>=|<>}
'value_string' | ALL}]]
```

Use the WHERE clause to propagate only those transactions where the *property_name* meets specific criteria. You can set a property to a currently active transaction in the replica database with the SAVE PROPERTY statement. Use the keyword ALL to propagate all statements, including those with no properties.

The keyword ALL overrides any default propagation condition that may have been set earlier with the SAVE DEFAULT PROPAGATE PROPERTY WHERE command. This command is used to make parameters available to other statements in the transaction on the Parameter Bulletin Board.

6.2.3 Refreshing Publication Data from Master to Replica

The MESSAGE APPEND REFRESH statement lets you refresh from a publication that is in the master database. The syntax is:

```
MESSAGE unique_message_name APPEND
   [REFRESH publication_name [(publication_parameters)] [FULL]]
```

Using this statement, you provide parameters to the publication (if they are used in the publication definitions) to narrow the scope of the replica's refresh. For example, you could specify that you only want to refresh the data related to a particular branch office.

6.2.4 Ending Messages

It is good practice to explicitly end each synchronization message with the MESSAGE END statement. This statement, together with the transaction commit operation, makes the message persistent in the replica by saving it to the synchronization system tables of the replica database. The syntax for ending the message is:

```
MESSAGE unique_messsage_name END
```

Note that after ending the message, you must be sure to commit the message by providing the COMMIT WORK statement.

6.2.5 Forwarding Messages to the Master Database

After a message has ended with the MESSAGE END statement and has been committed to make it persistent, you send it to the master database using the MESSAGE FORWARD statement. Each sent message is issued a reply message from the master database and each message returns a result set that should be fetched through the client application. The syntax is:

```
MESSAGE unique_message_name FORWARD
   [TIMEOUT {FOREVER | seconds }]
```

For example:

```
MESSAGE mymsg FORWARD TIMEOUT 60;
```

You can set the TIMEOUT option to define how long the replica database waits for the reply message before it expires and has to be requested using the MESSAGE GET REPLY statement described in the following section.

Note

If a master does not receive a complete message, then the master will not execute the portion of the message that it did receive.

6.2.6 Requesting a Reply Message from the Master Database

If the TIMEOUT in the MESSAGE FORWARD statement is not defined, the message is only forwarded to the master and the replica does not wait for the reply. In this case the reply can be retrieved with a separate MESSAGE GET REPLY call in the replica database. The syntax is:

```
MESSAGE unique_message_name GET REPLY
   [TIMEOUT {FOREVER | seconds }]
```

6.2.7 Configuring SmartFlow Messages

The content of the synchronization process is fully definable by the application designer. This way the application's needs are best considered. Similarly, the synchronization process can be tailored to efficiently utilize the capacity and characteristics of the currently available infrastructure. solidDB SmartFlow architecture itself does not provide any default process but it does not set any limitations on the contents of a custom built process either.

Setting Message Size Maximum

The maximum size of a single synchronization message can be set by database level system parameters. The *SYS_R_MAXBYTES_OUT* parameter sets the maximum length of messages sent from a replica database to the master, while *SYS_R_MAXBYTES_IN* sets the maximum length of messages that can be received to a replica database from the master database.

The default message length for both parameters is 2GB. Valid values for both parameters are between 0 - 2 GB. If 0 is specified, then 2GB is used.

To set these parameters, use the SET SYNC PARAMETER statement in the replica database. The syntax is:

```
SET SYNC PARAMETER parameter_name value_as_string
```

For example:

```
SET SYNC PARAMETER SYS_R_MAXBYTES_OUT '1048576000';
```

Note that for both parameters, an error message is issued if the message is longer than expected.

Setting the Commit Block Size

By default, all data of a publication refresh is written to the replica database in a single transaction. If the reply of a sent message will contain refreshes of large publications, you can adjust the number of rows that are committed in one transaction using the COMMITBLOCK option of the MESSAGE FORWARD or MESSAGE GET REPLY statements. This allows you to divide a single large transaction into multiple smaller transactions. The syntax for using COMMITBLOCK is:

```
MESSAGE unique_message_name FORWARD
[COMMITBLOCK block_size_in_rows]
```

or

```
MESSAGE unique_message_name GET REPLY
[COMMITBLOCK block_size_in_rows]
```

For example:

```
MESSAGE mymsg FORWARD TIMEOUT 300 COMMITBLOCK 1000
MESSAGE mymsg GET REPLY TIMEOUT 300 COMMITBLOCK 1000
```

Setting the maximum size of the commitblock can improve performance of the replica databases. However, data integrity cannot be guaranteed if the data is transmitted in more than one transaction and active users on the replica are changing data at the same time. Therefore, we recommend that you disconnect all online users from the replica database when you use the COMMITBLOCK option.

Note

If a Replica runs in the HSB configuration, the COMMITBLOCK clause is illegal in commands: MESSAGE GET REPLY, DROP SUBSCRIPTION, MESSAGE FORWARD and MESSAGE GET REPLY. If the COMMITBLOCK clause is used, it produces the error: 25083 Commit block can not be used with HotStandby

6.2.8 Executing a Synchronization Process

The synchronization process is usually initiated by, and is largely controlled from, the replica database. If you need master-initiated synchronization, you can use the Sync Pull Notify feature, which allows the master to notify the replica that it is time to start a synchronization operation. For details about Sync Pull Notify, see Section 4.8, “Sync Pull Notify”.

The creation and execution of a synchronization process follows this pattern:

1. Create a message by giving a unique name to it. For example:

   ```
   MSGNAME := GET_UNIQUE_STRING('MSG');
   ```

   Note that the autocommit mode *must* be switched off.

2. Append the synchronization tasks (propagate transactions and refresh from publications) to the message. Any number of tasks can be included in the message.

3. End the message and make it persistent by committing the transaction. From this point on, the store and forward messaging architecture guarantees that data contained by the message will not be lost.

4. Forward (send) the message to the master database.

The reply message can be received as part of the forward command. This is a useful approach if the reply message can be expected within a reasonable amount of time, for example, within a minute. Alternatively, if the reply is expected much later, for example, the next morning, the reply can be requested using a separate GET REPLY command.

Example 6.2. Executing a Typical Synchronization Process

```
-- Create a new message with name 'my_msg'.
-- AUTOCOMMIT must be off! Also, any preceding transaction must be
```

```
-- completed prior to executing the MESSAGE statements.
MESSAGE my_msg BEGIN ;
-- Append tasks to message: propagate transactions.
MESSAGE my_msg APPEND PROPAGATE TRANSACTIONS ;
-- Append tasks to message: refresh from publications.
MESSAGE my_msg APPEND REFRESH ORDERS_BY_SALESPERSON ('1') ;
MESSAGE my_msg APPEND REFRESH PARTS_IN_INVENTORY ;
-- End the message, make it persistent.
MESSAGE my_msg END ;
-- Commit the message creation operation.
COMMIT WORK ;
-- Send the message to master, don't wait for reply.
MESSAGE my_msg FORWARD ;
-- Request reply to the message separately from master.
-- Wait for the reply message for a maximum of 100 seconds.
MESSAGE my_msg GET REPLY TIMEOUT 100 ;
COMMIT WORK ;
```

6.3 Implementing Security through Access Rights and Roles

solidDB SmartFlow enforces security throughout the system by implementing access rights and roles. This section describes basic principles of SmartFlow security and how to use SmartFlow commands to set it up.

6.3.1 How SmartFlow Security Works

The solidDB SmartFlow security model is based on the following principles:

- There are two kinds of users: local users and master users.
- A local user has access rights in the replica database.
- A master user has access rights in the master database.
- For a local user to be able to perform synchronization-related tasks, the local user must be mapped to a corresponding master user in the master database.
- Master user's access rights are used when executing a synchronization message in master database.
- Both local users and master users exist in the replica database of a SmartFlow system.

- Local users can perform local database operations such as execute queries, create tables or call stored procedures based on the access rights that are defined for them. For example, the administrator of the local database can perform any operations on the local database. However, a local user has no access to the synchronization-related statements such as SAVE *`sql_statement`* or MESSAGE statements. A local user must be mapped to a master user in order for that local user to be able to propagate transactions to the master.

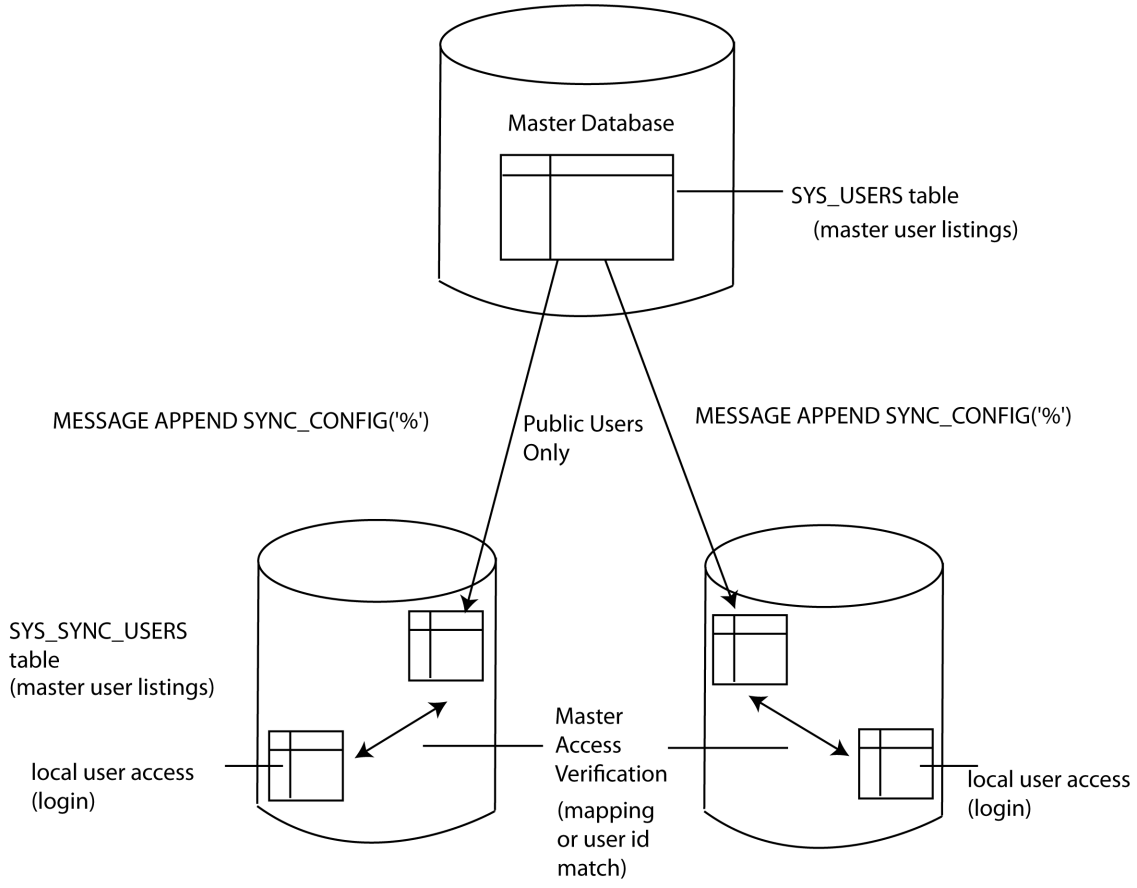
- Master users are users that are defined in the master database and have been downloaded to the replica database as part of the replica registration process. All synchronization operations require that a current master user is defined by mapping a local user ID to a master user ID.

  Master user names and passwords are defined separately in the SYS_SYNC_USERS table (described in the following section) of each replica database, giving master users the rights to save transactions in tables in which they have authorization. User access is also verified in the master database during synchronization.

- Both master and local users can have synchronization-specific roles, such as a role that allows replica registration or administrative role for synchronization functions.

For an overview of access rights requirements for each command, refer to Section 6.3.5, “Access Rights Summary”.

Figure 6.1. SmartFlow User Access Rights



Unless local users have master access, they are unable to perform any synchronization operations.

6.3.2 Changing Replica Access to the Master Database

When a transaction is executed on any server, that transaction must be executed with appropriate privileges (e.g. INSERT, DELETE, UPDATE, etc. privileges on the tables). When a transaction is propagated from a replica server to the master server, the transaction must of course execute with appropriate privileges on the master. (Executing with appropriate privileges on the replica is not sufficient to guarantee that the transaction will execute with appropriate privileges on the master.)

To ensure that propagated transactions can be executed on the master, you must map a replica user to a corresponding master user who has the appropriate privileges. E.g. you might map the user kathy_on_replica1 to the user kathy_on_master, if the user kathy_on_master has the appropriate privileges.

Of course, when the users on the master change, you may need to update the "mapping" information on the replicas; otherwise, your replica users might map to master users that are no longer appropriate or that no longer exist. To download the updated information to applicable replicas, you need to execute the MESSAGE APPEND SYNC_CONFIG command in the replicas. Once the updated master user information has been downloaded, you then need to re-map replica user ids to the appropriate master user ids using the ALTER USER SET MASTER statements (refer to *Mapping Replica User ID with Master User ID* below).

The MESSAGE APPEND SYNC_CONFIG command itself requires appropriate privileges. When you create a new replica, that replica has no privileges and thus cannot connect to the master. You need to create a replica registration user to initially populate the SYS_SYNC_USERS table with the list of master users from the master database; from then on, the list of master users can be downloaded as needed from the master database.

Below, we describe in more detail how to update the mapping of a replica that already has the privileges required to get updated user data from the master.

Updating Master Users for SmartFlow Operations

To update master users in a replica:

From the replica, subscribe user information from the master database in a separate message using the following command:

```
MESSAGE unique_message_name APPEND SYNC_CONFIG
   (sync_config_arg)
```

The *sync_config_arg* defines the search pattern of the user names that are returned from the master database to the replica. If you want all names to be sent to the replica, specify the SQL wildcard % as the input argument.

For example:

```
MESSAGE CFG2 BEGIN;
MESSAGE CFG2 APPEND SYNC_CONFIG('%');
```

```
MESSAGE CFG2 END;
COMMIT WORK;
```

Managing Master Users

Master users control all access to the data synchronization functions of solidDB. To allow a replica database to synchronize its data with the master database, the replica must download master user information from the master database and map one or more local user ids to a master user id.

Mapping Replica User ID with Master User ID

To map a replica user id to a master user id, you use the ALTER USER SET MASTER statement. When you use the ALTER USER SET MASTER statement, you provide the master user names and passwords for the local users that you want to map to master users. The same local user can be mapped to multiple master users.

After you have completed the mapping, when a local user logs into a replica database, solidDB checks to see whether that local user is mapped to any master user id. If no mapping is specified, by default, solidDB looks for the same user id and password in the master and replica. Thus, if mapping is not used, the user id and password in both the master and replica must be the same.

Replica table SYS_SYNC_USERS can be updated with the latest master usernames using the MESSAGE APPEND SYNC_CONFIG command. Refer to Figure 6.1, “SmartFlow User Access Rights”, which illustrates this concept.

Setting Public and Private Users

Database Administrators can alter users in the SYS_USERS table of the master database to designate those users as private or public. If the PRIVATE option is set for a user, this user's id and password are never sent to the replica during a subscription of the publication to the replica.

Even if a PRIVATE user matches a specified subscription request in the MESSAGE APPEND SYNC_CONFIG command, as long as that user is set for PRIVATE, the user's information stays in the master's SYS_USERS table. Only PUBLIC users are downloaded from master to replica to fulfill a SYS_SYNC_USERS table subscription request. By default, a user is set for PUBLIC. For details on setting users to public and private, read the *solidDB SQL Guide*.

To change a user from public to private or vice-versa, use the command:

```
ALTER USER SET { PRIVATE | PUBLIC }
```

Note

There is no way to set a user as private for some replicas and public for others. Users are designated as either public or private throughout a SmartFlow system.

We recommend that you set all users with administrative rights (DBA) to PRIVATE. This provides an extra measure of security by preventing a DBA's password from ever being sent to a replica and becoming public. If the DBA password became exposed and you needed to restore security, all replicas of the system would need to be dropped and recreated after the password was changed in the master database.

Other users may also be set to PRIVATE if those users are not needed in replicas.

Note

If a replica builds messages or executes transactions using a user who is private in a master database, then the operation in the master (when receiving messages or when executing transactions) fails with a security error.

Determining Access Rights of Transactions and Refresh Commands

Synchronization messages are labeled with the master username of the creator of the message. solidDB uses the master username to specify the account under which the message is executed. All subscriptions are executed using this account. Each transaction uses the master user who saved the statements in the replica.

6.3.3 Setting Up Access Rights

The following sections identify the access rights required to implement a SmartFlow system.

Granting Access

Local users must have appropriate access rights (in both the master database and the user's local replica database) to the tables they use for transactions and execute rights to the procedures they execute. Note that if procedures are used to perform synchronization functions in the replica database, the local user who has created the procedure must be mapped to a master user.

The DBA of the master database should grant to master users

- appropriate access rights in the master database to the tables they use for publications and transactions, and

- execute rights to the procedures they execute.

Note

Once access rights are granted, they take effect when the user who is granted the rights logs on to the database. If the user is already logged on to the database when the rights are granted, the rights take effect only if the user:

- accesses for the first time the table or object on which the rights are set, or
- disconnects and then reconnects to the database.

In the applicable replica database, specify which is the currently active master user by mapping the replica user id with the master user id by using the ALTER USER SET MASTER command.

When setting up access rights, you can use the following SmartFlow SQL commands:

```
CREATE USER username IDENTIFIED BY password

GRANT rolename TO username
GRANT [SELECT | UPDATE | INSERT | DELETE] ON tablename TO username

GRANT EXECUTE ON procedure_name TO username
```

Read the *solidDB SQL Guide* for details.

To grant users access to publications, use:

```
GRANT REFRESH ON publication_name TO username
```

Read the section called “Granting REFRESH Access” for details.

To map a replica user id to a master user id:

```
ALTER USER replica_user SET MASTER master_name USER user_specification
```

Read the *solidDB SQL Guide* for details.

Granting REFRESH Access

To grant access rights on a publication to a local user, user role (created with the create role statement), or all users, use the GRANT REFRESH statement in the master database. The syntax is:

```
GRANT REFRESH ON publication_name TO {PUBLIC | user_name,
   [, user_name] ... | role_name [, role_name ] ...
```

For example:

```
GRANT REFRESH ON customers_by_area TO salesman_jones
```

Revoking REFRESH Access

To revoke access rights on a publication to a local user, user role (created with the create role statement), or all users, use the REVOKE REFRESH statement in the master database. The syntax is:

```
REVOKE REFRESH ON publication_name FROM {PUBLIC | user_name,
   [user_name] ... | role_name [, role_name, [, role_name]...}
```

For example:

```
REVOKE REFRESH ON customers_by_area FROM salesman_jones
```

Saving Transactions in Replica

When a local user saves a statement of a transaction in the replica, the transaction in the statement is labeled with the current master user's username. When the transaction is re-executed in the master database, it uses the access rights defined for the master user.

When the master encounters a user access violation during transaction propagation, it terminates the execution of the synchronization message. This ensures that a local replica user is not able to execute any unauthorized statements in the master database.

Creating Access to Applications on Different Masters

In a multi-master environment, you can map a single user id to a different master user in each catalog. When you change the active replica catalog using the SET CATALOG command, the current master user changes

automatically. For example, in a replica database assume there is one local user. This user is mapped to the CALENDARUSER master user of the calendar application and the NEWSUSER master user of the news application. The SET CATALOG command is used to set the current catalog to either CALENDAR or NEWS. If CALENDAR catalog is set, then the current master user is automatically set to the CALENDARUSER master user. Similarly, if NEWS is set, the current master user is set to NEWSUSER.

If mapping is not defined, the first SmartFlow data synchronization operation (for example, SAVE or MESSAGE statement) returns the "no active master user" error.

Creating User rights to Publications and Tables

A user on the master who defines a publication must have read and write access rights to the tables referenced by that publication.

Note that subscriptions are executed in the master database using the master username of the creator of the message that contained the REFRESH *publication* clause.

To use a table that is involved in synchronization, the local user must have rights to the actual subscription tables in the replica database, and the corresponding (mapped) master user must have subscribe access rights to the publication in the master database. When subscription rows are inserted (or deleted) in a replica, solidDB verifies that the subscriber has INSERT and DELETE rights on the tables.

The user who defines a publication also has the right to drop that publication.

Read Section 6.3.3, “Setting Up Access Rights” for details on granting publications access rights.

Creating the Replica Registration User

When a new SmartFlow replica database is created, the SYS_SYNC_USERS table of the new database is empty (contains no data). To register the new replica database with the master database and to initially populate the table with data requires a username that is from the master database and that has registration rights.

You can provide registration rights for a master user in the master database by designating the user with the SYS_SYNC_REGISTER_ROLE or the SYS_SYNC_ADMIN_ROLE using the GRANT *rolename* TO *user* command.

You must provide this registration username and password to the replica site that wants to register with the master. This allows each replica site to explicitly set the registration user at the replica with the following command:

```
SET SYNC USER username IDENTIFIED BY password
```

Since the username resides in the master database, this command allows the registration user to explicitly register the replica. The SYS_SYNC_USERS replica table can then be populated with the public SYS_USERS information from the master database as part of the replica registration process.

When the replica has successfully executed registration, execute the following statement:

```
SET SYNC USER NONE
```

Otherwise, if SET SYNC USER *username* is active and a user saves statements, propagates, refreshes, or registers to a publication, the following error message is returned:

```
User definition not allowed for this operation.
```

Note

The SET SYNC USER command is used for replica registration only. Aside from registration, all other synchronization operations require a valid master user ID that has been downloaded to the replica database using the SYNC_CONFIG task. If you want to designate a different master user for a replica, you must map the replica ID on the replica database to the master ID on the master database. For details, read *Mapping Replica User ID with Master User ID* in this chapter.

For details on how the master users control user access of data synchronization functions, read the section called “Managing Master Users”.

6.3.4 Implementing Special SmartFlow Roles

solidDB's two special roles for performing synchronization operations are:

- SYS_SYNC_ADMIN_ROLE

  This is an administration role for performing SmartFlow data synchronization operations, such as deleting messages. Anyone with this privilege has all synchronization roles granted automatically.

- SYS_SYNC_REGISTER_ROLE

  This is a role only for registering or unregistering a replica database to the master.

SYS_SYNC_ADMIN_ROLE automatically includes the SYS_SYNC_REGISTER_ROLE.

To grant these roles, use the following syntax of the GRANT statement in the master database:

```
GRANT role_name TO user_name
```

Note

Once a user role is granted, it takes effect when the user who is granted the role logs on to the database. If the user is already logged on to the database, the user must disconnect and then reconnect to the database for the role to take effect.

6.3.5 Access Rights Summary

Following is a comprehensive summary of the access rights required to execute each SmartFlow command in a replica database and the master database.

Access Rights in the Replica

The following table lists the access rights requirements for synchronization operations in the replica database.

Table 6.1. Access Rights in the Replica

Command	Task	Access rights requirements
ALTER TABLE SET SYNCHISTORY | NOSYNCHISTORY	Specify whether to set up a table for incremental publication	Same as the SQL ALTER TABLE command (owner of the table, or DBA)
ALTER USER SET MASTER	Map a replica user id to a master user id	SYS_SYNC_ADMIN_ROLE
GET_PARAM( )	Retrieve a parameter that was placed on the bulletin board with PUT_PARAM( )	Any user
PUT_PARAM( )	Place a parameter on the bulletin board	Any user
SAVE	Save a statement of a transaction in the replica database for later propagation to the master	Valid master user
SAVE PROPERTY	Assign properties to the current active transaction	Valid master user
MESSAGE BEGIN	Begin a new synchronization message	Valid master user, SYS_SYNC_ADMIN_ROLE, or SYS_SYNC_REGISTER_ROLE

Command	Task	Access rights requirements
MESSAGE APPEND REFRESH	Refresh from a publication	Valid master user
MESSAGE APPEND PROPAGATE TRANSACTIONS	Propagate transactions	Valid master user
MESSAGE APPEND { REGISTER | UNREGISTER } REPLICA	Register or unregister replicas with the master database	SYS_SYNC_ADMIN_ROLE or SYS_SYNC_REGISTER_ROLE
MESSAGE APPEND{ REGISTER PUBLICATION | UNREGISTER PUBLICATION }	Register or unregister publications in a replica. If the publication is registered, users are allowed to refresh from the publication.	Refresh access to the publication
MESSAGE APPEND SYNC_CONFIG	Download the data of the SYS_SYNC_USERS table to the replicas	SYS_SYNC_ADMIN_ROLE or SYS_SYNC_REGISTER_ROLE
MESSAGE FORWARD	Send saved message to master database	Valid master user or SYS_SYNC_ADMIN_ROLE
MESSAGE GET REPLY	Get reply to the sent message	Valid master user or SYS_SYNC_ADMIN_ROLE
MESSAGE DELETE [FROM REPLICA]	Delete entire message (all transactions) from the replica database to recover from an error	SYS_SYNC_ADMIN_ROLE
MESSAGE DELETE [FROM REPLICA] CURRENT TRANSACTION	Delete current transaction from the synchronization message to recover from an error	SYS_SYNC_ADMIN_ROLE
DROP MASTER	Drop master definition	SYS_SYNC_ADMIN_ROLE
DROP SUBSCRIPTION	Drop subscriptions in a replica database	Valid master user
DROP PUBLICATION REGISTRATION	Drop publication registrations in a replica database	SYS_SYNC_ADMIN_ROLE
IMPORT	Import data from a data file created by the EXPORT SUBSCRIPTION command in a master database.	Valid master user
SET SYNC CONNECT *listen_name* TO MASTER *master_name*	Change the network name associated with a master database	SYS_SYNC_ADMIN_ROLE

Command	Task	Access rights requirements
SET SYNC NODE *node_name* | NONE	Assign a nodename to the database as part of registration; or remove a node name, for example when removing registration and dropping a synchronized database.	SYS_SYNC_ADMIN_ROLE
SET SYNC PARAMETER	Set synchronization-related database parameters in a synchronized database catalog	SYS_SYNC_ADMIN_ROLE
SET SYNC { REPLICA | MASTER } { YES | NO }	Designates the database catalog as a replica and/or master	SYS_SYNC_ADMIN_ROLE
SET SYNC USER NONE	Makes current registration user inactive in the current database connection	Any local user
SET SYNC USER *username* IDENTIFIED BY *password*	Defines the currently active master user name and password used for the registration process.	SYS_SYNC_ADMIN_ROLE

Access Rights in the Master

The following table lists the access rights that are required to execute SmartFlow commands in the master database.

Table 6.2. Access Rights in the Master

Command	Task	Access right requirements
ALTER TABLE SET SYNCHISTORY | NOSYNCHISTORY	Specify whether to set up a table for incremental publication	Same as the SQL ALTER TABLE command (owner of the table, or DBA)
ALTER USER SET { PUBLIC | PRIVATE }	Include or exclude a user id in subscription downloads to a replica SYS_SYNC_USERS table.	DBA or SYS_SYNC_ADMIN_ROLE
GET_PARAM( )	Retrieve a parameter that was placed on the bulletin board with PUT_PARAM( ).	Any user
PUT_PARAM( )	Place a parameter on the bulletin board	Any user

Command	Task	Access right requirements
CREATE PUBLICATION	Define a publication in the master database	Valid master user who has full access to the tables of the publication.
CREATE SYNC BOOKMARK	Create a bookmark in the master database	DBA or SYS_SYNC_ADMIN_ROLE
DROP SYNC BOOKMARK	Drop a bookmark in the master database	DBA or SYS_SYNC_ADMIN_ROLE
GRANT REFRESH ON	Grant access rights on a publication to a user or role defined in the master database.	Creator of the publication, or DBA
REVOKE REFRESH ON	Revoke access rights on a publication to a user or role defined in the master database	Creator of the publication, or DBA
DROP PUBLICATION	Drop a publication in the master database	Creator of the publication, or DBA.
EXPORT SUBSCRIPTION	Export master data to a file	Master user who has subscribe access to the publication
MESSAGE DELETE FROM REPLICA	Delete entire synchronization message (all transactions) to recover from an error	SYS_SYNC_ADMIN_ROLE or DBA
MESSAGE DELETE CURRENT TRANSACTION	Delete current (failed) transaction of a synchronization message to recover from an error	SYS_SYNC_ADMIN_ROLE or DBA
MESSAGE FROM REPLICA EXECUTE	Execute a failed message from the replica in the master database	SYS_SYNC_ADMIN_ROLE or DBA
DROP SUBSCRIPTION REPLICA	Drop a replica's subscription to a publication in the master	SYS_SYNC_ADMIN_ROLE or DBA
DROP REPLICA	Drop a replica database from the master database	SYS_SYNC_ADMIN_ROLE or DBA
SET SYNC {MASTER | REPLICA}{YES | NO}	Designate the database catalog as a master and/or replica	SYS_SYNC_ADMIN_ROLE or DBA
SET SYNC USER NONE	Makes current master user inactive in the current database connection	SYS_SYNC_ADMIN_ROLE or DBA
SET SYNC PARAMETER	Set synchronization-related database parameters in the master database catalog	Valid master user

Command	Task	Access right requirements
SET SYNC NODE { *node_name* | NONE}	Assign a nodename to the master database as part of registration; or remove a node name, for example when removing registration and dropping a synchronized database.	SYS_SYNC_ADMIN_ROLE or DBA

6.4 Setting Up Databases for Synchronization

After a database is defined (as a master, replica, or both), a database schema and catalogs (if necessary) are created, and user access rights are implemented, you are ready to configure the databases for synchronization. This section requires that you assign database node names for each database. You can use Solid SQL Editor (solsql) or SolidConsole to enter the SmartFlow statements required for set up.

6.4.1 Configuring the Master Database(s)

Before you begin, be sure you have defined your master database(s). For details, see Section 5.2.1, “Defining Master and Replica Databases”.

To each master database, provide a node name that is unique within the domain. For example:

```
SET SYNC NODE "MASTER";
COMMIT WORK;
```

6.4.2 Registering Replicas with the Master Database

Before you begin, be sure to set AUTOCOMMIT off so that you can compose multi-statement MESSAGEs. Make sure that you commit or roll back any active transaction. Also, be sure that you have defined a master username and password for registering replicas to the master database, and that you know the name(s) of the catalog(s) in your environment. Also be sure you have defined your replica databases; for details see Section 5.2.1, “Defining Master and Replica Databases”.

In each replica database, perform the following steps:

1. If a local database will synchronize data with multiple master databases (i.e. it will contain more than one replica), then create a catalog for each replica. For example:

   ```
   CREATE CATALOG CAT_FOR_REP1;
   COMMIT WORK;
   ```

2. Give this replica catalog a node name that is unique across the replicas of the master database of this replica. Before you set the node name of the catalog, you must already have set that catalog to be the current catalog. For example:

```
SET CATALOG CAT_FOR_REP1;
COMMIT WORK;
SET SYNC NODE "REPLICA1";
COMMIT WORK;
```

Note

If you have many replicas, then we recommend that you name replicas with logical names that are derived, for example, from the server's logical name or location.

Also, note that a catalog in the master and its corresponding node in the master can have different names.

3. Set the master user for replica registration. For example:

```
SET SYNC USER REG_USER IDENTIFIED BY SECRET;
```

4. Register the replica to master Master1 by sending a registration message. For example:

```
MESSAGE CFG1 BEGIN TO "MASTER";
MESSAGE CFG1 APPEND REGISTER REPLICA;
MESSAGE CFG1 END;
COMMIT WORK;
MESSAGE CFG1 FORWARD TO 'tcp 1315' TIMEOUT FOREVER;
COMMIT WORK;
```

Note

When using the REGISTER REPLICA command and registering a replica with a catalog other than the master server's default catalog, you must provide the applicable master node name of the catalog in the MESSAGE BEGIN command. solidDB can then resolve the correct catalog at the master database for the replica. Following is the syntax:

```
MESSAGE message_name BEGIN TO master_node_name
```

The MESSAGE FORWARD command sends the message to the master database after the message is made persistent with the MESSAGE END command. Note that the network listen name of the recipient of the message is specified in the MESSAGE FORWARD command. This is necessary only when the first message from a new replica to the master database is sent. If a TIMEOUT is not defined, the replica does not fetch the reply. It must be retrieved with a separate MESSAGE GET REPLY call.

5. Subscribe master username information from the master database using the MESSAGE APPEND SYNC_CONFIG command in a separate message. In this example the SQL wildcard '%' is used to request that all usernames be sent from the master database.

```
MESSAGE CFG2 BEGIN;
MESSAGE CFG2 APPEND SYNC_CONFIG('%');
MESSAGE CFG2 END;
COMMIT WORK;
MESSAGE CFG2 FORWARD TIMEOUT FOREVER;
COMMIT WORK;
```

6. When you have subscribed the master username information successfully, reset the sync user to "none" so the registration user is no longer active in the replica database connection. For example:

```
SET SYNC USER NONE;
```

The registration user has no rights other than registration. If the registration user is still active, subsequent commands will typically get the following error message:

```
User definition not allowed for this operation
```

6.5 Creating Publications

A *publication* is a definition of a set of data to be downloaded from the master database to a subscribing replica database. It is completely separated from the transactions that change the data. Note that "traditional" replication methods typically rely on sending transactions (inserts, updates, and deletes) from master to replicas, but solidDB instead sends a snapshot of the updated data to the replicas.

SmartFlow publication definitions may include:

- Data from one or multiple tables — You can define relations between the tables of a publication
- All or subset rows of a table — A publication can contain a normal SELECT statements for selecting data for a publication
    - By limiting a subset row of tables with parameters, you can specify fixed or dynamic search criteria for the publication
- All or defined columns of a table — A publication can contain normal SELECT statement for selecting columns for a publication
- Full or incremental data — A full publication sends all data contained in a publication. An incremental publication sends only data that has changed since the previous refresh.

Note

To save resources and increase performance, we recommend that you use incremental publications. You must set up tables for incremental publication before creating the actual publication. For details, read Section 6.5.1, “Creating Incremental Publications” in the following section.

6.5.1 Creating Incremental Publications

For the server to be able to do incremental refreshes on a table, the server must store some information about the most recent preceding refresh for that table. This refresh information is known as synchronization history data. (The synchronization history data for each table is stored in a synchronization history table. There must be one synchronization history table for each table in the publication.)

In order to make a publication incremental, you must set each table's SYNCHISTORY property, which tells the server to gather that table's synchronization history data. The command to do this is:

```
ALTER TABLE table_name SET SYNCHISTORY
```

You must execute this command for each table in the publication, and you must do so on both the master and replica databases.

The synchistory setting for a table is considered a "property" of that table. By default, this property is set to NOSYNCHISTORY for each table. If this property is set to SYNCHISTORY in both the master and the replica databases, then after the first refresh, subsequent refreshes to a specific publication will send the replica database only new and modified rows when the data in the table is synchronized.

It is recommendable that you set the SYNCHISTORY property for a table before it is referenced by any publication. If you want to alter the SYNCHISTORY property for a table after it has been included as part of

a publication, you need to use the Sync Maintenance Mode. For details, see Section 7.2, “Upgrading the Schema of a Distributed System”.

For example, to set the SYNCHISTORY property of the table named SYNCDEMO, use the command:

```
ALTER TABLE SYNCDEMO SET SYNCHISTORY;
COMMIT WORK;
```

This statement creates a shadow table that stores history data. The shadow table tracks rows that were modified or deleted from the main table. If there are no longer any replica databases that require the data for their refreshes, the unnecessary data of the shadow table is automatically deleted.

For details on optimizing history data management, read Chapter 9, *Performance Monitoring and Tuning*.

6.5.2 Using the Create Publication Statement

When you create a publication, you specify which table (in the master) the data should be read from, and which table (in the replica(s)) the data should be written to.

To create a publication, execute the CREATE PUBLICATION statement in the master database. The syntax is:

```
"CREATE PUBLICATION publication_name
   [( parameter_definition [,parameter_definition ...])]
BEGIN
   main_result_set_definition...
END";

main_result_set_definition ::=
RESULT SET FOR main_replica_table_name
   BEGIN
      SELECT select_list
   FROM master_table_name
   [ WHERE search_condition ] ;
      [[DISTINCT] result_set_definition...
   END

result_set_definition ::=
RESULT SET FOR replica_table_name
   BEGIN
```

```
   SELECT select_list
FROM master_table_name
[ WHERE search_condition ] ;
[[DISTINCT] result_set_definition...]
END
```

The CREATE PUBLICATION statement lets you specify publications for incremental "download" of new and changed data from the master to a replica database.

The data of a publication is always read from the master database in one transaction. This guarantees that the data read from the publication is internally consistent.

Caution

The data read from the publication is internally consistent unless the master is using the READ COMMITTED transaction isolation level. The transaction isolation level for refreshes can be different from the system default. For more information, refer to *solidDB Administration Guide*, Appendix A, Configuration Parameters, parameter *RefreshIsolationLevel*.

By default, the publication data is also written to the replica database in one transaction to maintain that consistency. However, on the replica side you may override the default behavior and split the refresh into multiple transactions by using COMMITBLOCK. If you use COMMITBLOCK, then you lose the guarantee of internal consistency. See the section called “Setting the Commit Block Size” for more details about COMMITBLOCK.

The *search_condition* can reference *parameter_definitions* and/or columns of replica tables defined on previous (higher) levels. Search conditions of a SELECT clause can contain input arguments of the publication as parameters. The parameter name must have a colon as a prefix.

Publications can contain data from multiple tables. These tables may be independent, or there may be relations between them. If there is a relation between tables, you must nest the result sets. The WHERE clause of the SELECT statement of the inner result set of the publication must refer to a column of the table of the outer result set.

Note that even if the publication contains multiple tables and there is a relation between tables, the number of tables that the data is written to (in the replica) is the same as the number of tables that the data was read from (in the master). Records from multiple tables in the master are not summarized or joined into a single record in the replica the way you might join two tables into a single view, or summarize data by using an aggregate function such as SUM().

Here is a typical publication:

```
CREATE PUBLICATION ORDERS_BY_SALESPERSON
```

```
(SALESPERSON_ID VARCHAR)
BEGIN
   RESULT SET FOR CUST_ORDER
   BEGIN
      SELECT * FROM CUST_ORDER
      WHERE SM_ID = :SALESPERSON_ID AND STATUS = 'ACTIVE';
      RESULT SET FOR ORDER_LINE
     BEGIN
        SELECT * FROM ORDER_LINE
        WHERE ORDER_ID = CUST_ORDER.ID;
      END
      DISTINCT RESULT SET FOR CUSTOMER
      BEGIN
         SELECT * FROM CUSTOMER
         WHERE ID = CUST_ORDER.CUSTOMER_ID ;
      END
   END
END;
```

The above sample publication retrieves data from three tables of the master database:

- The main table of the publication is CUST_ORDER. Rows are retrieved from the table using SALESPERSON_ID as a search criterion.
- For each order, rows from ORDER_LINE table are retrieved. The multiplicity between CUST_ORDER and ORDER_LINE is 1-N. The data is linked together using the ID column of CUST_ORDER table and ORDER_ID column of the ORDER_LINE table.
- For each order, also a row from the CUSTOMER table is retrieved. The multiplicity between CUST_ORDER and CUSTOMER tables is N-1; that is, a customer may have multiple orders. The data is linked together using the CUSTOMER_ID column of the CUST_ORDER table and ID column of the CUSTOMER table. The keyword DISTINCT ensures that the same customer information is brought to the replica database only once.

Publication Guidelines

You can make publications that have 1-N and N-1 relationships between the result sets. You can also nest the result sets. For example, a CUST_ORDER can have ORDER LINES (1-N) and each ORDER LINE can have a PRODUCT (N-1).

If the relation between outer and inner result set of the publication is a N-1 relationship, then the keyword DISTINCT must be used in the result set definition.

Each nested result set is internally treated as a join. The more tiers of result sets in the publication, the more complex the queries must be to retrieve the data. Therefore, for better performance, avoid extensive nesting of result sets. In the following examples, better performance is achieved by rewriting an equivalent unnested version of the CREATE PUBLICATION statement:

Example 6.3. Nested Publication Version

```
CREATE PUBLICATION NESTED (IN_ORDER_ID INTEGER)
BEGIN
RESULT SET FOR CUST_ORDER
   BEGIN
      SELECT * FROM CUST_ORDER
      WHERE ID = :IN_ORDER_ID;
      RESULT SET FOR ORDER_LINE
      BEGIN
         SELECT * FROM ORDER_LINE
         WHERE ORDER_ID = CUST_ORDER.ID;
      END
   END
END;
```

Example 6.4. Unnested Publication Version

```
CREATE PUBLICATION UNNESTED (IN_ORDER_ID INTEGER)
BEGIN
RESULT SET FOR CUST_ORDER
   BEGIN
      SELECT * FROM CUST_ORDER
      WHERE ID = :IN_ORDER_ID;
   END
RESULT SET FOR ORDER_LINE
   BEGIN
      SELECT * FROM ORDER_LINE
      WHERE ORDER_ID = :IN_ORDER_ID;
   END
END;
```

Each publication that you create is fully independent from every other publication. This means you cannot define dependencies between publications.

Do not use overlapping publication definitions in replicas. This includes publication definitions with overlapping tables and WHERE conditions. Publication definitions overlap if they can potentially produce overlapping subsets of the same table, i.e. some or all rows can simultaneously be in both subsets.

For example, if the publication "ORDERS_BY_SALESPERSON" retrieves customer information, it is not advisable to have another publication, such as "CUSTOMERS_BY_AREA", that can retrieve the same rows from the master database to the subscribing replica. This will lead to conflict situations e.g. when dropping subscriptions to publications, resulting in the deletion of a subscription's entire replica data, regardless of whether another subscription is referring to those rows.

Also make sure that you are not doing overlapping refreshes of same publication within a replica. For example, if you start a new REFRESH before the reply message of the previous REFRESH operation has been processed, the replica database may contain incorrect data.

After using CREATE PUBLICATION, it is important to commit the transaction before any replica subscribes to the publication. If a transaction that defines a publication is uncommitted at the master, then when the replica tries to subscribe to that publication, the system issues an error message stating that the publication does not exist.

Publication data is requested from the master database using the MESSAGE APPEND REFRESH *`publication_name`* statement. For details, read the next chapter.

6.5.3 Subscribing to Publications

Replica databases use refreshes to request publication data from the master. Refreshes depend on the publication definition in the master database. You must be sure that the replica registers publications so that they can be refreshed from. Users are unable to refresh from publications that are not registered in the replica. Registering publications allows publication parameters to be validated.

Publications are registered in a replica using the MESSAGE APPEND REGISTER PUBLICATION statement. The syntax is:

```
MESSAGE APPEND REGISTER PUBLICATION publication_name
```

For example:

```
MESSAGE MyMsg0001 APPEND REGISTER PUBLICATION publ_customer;
```

For users to access a publication, they must have REFRESH privilege on the publication, and they must have privileges on the tables that are used in the publication. The table owners (or DBA) must grant privileges on the tables that are used in the publication, and the publication creator (or DBA) must GRANT REFRESH to

give the user access rights on the publication. For details, read Section 6.3, “Implementing Security through Access Rights and Roles”.

Note

A replica can only refresh from publications that are defined in the master. A replica cannot use a publication that has been defined in the replica database itself. If a CREATE PUBLICATION command is executed in the replica database, the publication definition is stored in the replica, but it not used unless the replica is also a master to another tier in a hierarchy of three or more tiers.

Publication data is requested from the master database as a publication call with a set of input parameter values (if they are used in the publication). The syntax is:

```
MESSAGE unique_message_name APPEND
      [REFRESH publication_name [( publication_parameters )]
   [FULL]]
```

For example:

```
MESSAGE my_msg APPEND
      REFRESH ORDERS_BY_SALESMAN ('SMITH') ;
```

The initial "refresh" is always a full publication and all data meeting the search criterion of the publication is sent to the replica database. Subsequent refreshes for the same publication contain only the data that has been changed since the prior refresh. This is known as an *incremental publication*. To save resources and increase performance, we recommend that you use incremental publications. Typically, only publication updates with the latest modifications need to be sent to a replica. Read Section 6.5.1, “Creating Incremental Publications” for details on setting tables to track modifications for incremental publication.

When you use the keyword FULL with REFRESH, this forces the fetching of full publication data to the replica. If the publication is a large one, then the initial (non-incremental) download of the data to the replica database will make a large transaction. In such cases, the size of a single transaction of a synchronization message can be limited with the COMMITBLOCK option. See the section called “Setting the Commit Block Size” for more details about COMMITBLOCK.

Combining Subscribed and Local Data

A table on the replica may contain not only subscribed data, but also "local data". To have rows for local use only at a replica, the publications should have 'where' constraints that exclude the local rows. In this situation, the replica will keep the local rows in the table, and will add the subscribed data from the master.

Dropping Subscriptions

Once the subscribed data becomes obsolete in the replica, you can delete the subscribed data by dropping the subscription using the DROP SUBSCRIPTION command in the replica. For details, refer to Section 7.1.7, “Modifying Publications and Tables in Publications”.

Unregistering or Dropping Publication Registrations

Registered publications can be unregistered in the replica using the following command in a synchronization message:

```
MESSAGE APPEND UNREGISTER PUBLICATION publication_name
```

For example:

```
MESSAGE MyMsg0001 APPEND UNREGISTER PUBLICATION publ_customer;
```

This must be part of a message that gets propagated to the master.

Registered publication definitions can also be dropped in the replica without sending a message. The syntax is:

```
DROP PUBLICATION publication_name REGISTRATION
```

For example:

```
DROP PUBLICATION publ_customer REGISTRATION;
```

The DROP PUBLICATION REGISTRATION command is meant only for situations where the replica cannot communicate with the master. If you drop a subscription without notifying the master, then any system information for that subscription will remain on the master and use up space indefinitely. Most importantly, synchronization history data is gathered for this replica even though the replica will never use it. This causes bloating of the "shadow tables" related to the publication. If possible, you should manually release that system info by going to the master database and dropping the subscriptions using the

```
DROP SUBSCRIPTION publication_name(parameters) FROM REPLICA replica_name
```

command.

6.6 Designing and Implementing Intelligent Transactions

Traditionally, a transaction is an atomic set of database operations that changes a database from one valid state to another valid state. A "valid state of a database" is a state in which no integrity rules or consistency rules are violated in the database. These rules can be both database specific (referential integrity) and application specific.

Solid *Intelligent Transaction* is an extension to the traditional transaction model. It allows you to implement transactions that are capable of validating themselves in the current database and adapting their contents (if required) according to the rules of the transaction.

For example, in an order processing system, an application rule might permit the creation of an order only if the customer's credit limit is not exceeded. A "create order" transaction may consist of inserting a row into the CUST_ORDER table and inserting another row into the INVOICE table. If inserting an order fails because a customer credit limit is exceeded, then inserting an invoice about the order should also fail. The INSERT_ORDER procedure should inform the INSERT_INVOICE procedure about the validation error. This allows the INSERT_INVOICE procedure to change its behavior and thus keep the transaction valid.

The existence of application specific consistency rules lead to the following transaction design principles:

A Solid *Intelligent Transaction* is a collection of SQL statements that may contain business logic that is typically implemented as a Solid stored procedure. Transactions that are intelligent have the following behavior and characteristics:

- They contain more than one operation, that is, calls to more than one stored procedure.
- They are long-lived because they are created, tentatively committed, and saved in the replica database, but finally committed in the master database. Thus, all validity checking of each transaction in the master database must be done by the transaction itself.
- They are responsible for the consistency of the master database.

Example 6.5, “Create Order Transaction” creates a "create order" transaction in a simple order entry application. The following sections use this example to illustrate how to implement a Solid *Intelligent Transaction*.

Example 6.5. Create Order Transaction

```
-- Make changes to local database
CALL INSERT_ORDER(...) ;
CALL UPDATE_CUSTOMER_CREDIT(...) ;
```

```
-- Save a property to the transaction
SAVE PROPERTY priority VALUE '1';
-- Save the statements for later propagation to master
SAVE CALL INSERT_ORDER(...) ;
SAVE CALL UPDATE_CUSTOMER_CREDIT(...) ;
-- make the local changes as well as the saved transaction
-- persistent
COMMIT WORK;
```

6.6.1 Updating Local Data

In Example 6.5, “Create Order Transaction”, the first part (local changes) of the transaction is a straightforward execution of the standard SQL clauses:

```
-- Make changes to local database
CALL INSERT_ORDER(...) ;
CALL UPDATE_CUSTOMER_CREDIT(...) ;
```

In the solidDB SmartFlow architecture, local changes remain local unless the statements and parameters of a transaction are explicitly saved for later propagation.

6.6.2 Saving the Transaction for Later Propagation

In this excerpt from Example 6.5, “Create Order Transaction”, the presence of the SAVE statement after the local changes specifies the later propagation of the local updates to the master database.

```
-- save the statements for later propagation to master
SAVE CALL INSERT_ORDER(...) ;
SAVE CALL UPDATE_CUSTOMER_CREDIT(...) ;
```

The syntax for saving a statement for later propagation is:

```
SAVE sql_statement
```

Note that it is also possible to merely save the statements for later propagation and not update the local database at all. In this case, the replica will get the updated information after it propagates the information to the master and then refreshes from updated data from the master.

Important

The SAVEd statement is executed "as is" on the master; the statement does not carry with it any memory of which records on the replica were affected when it was executed on the replica. For example, suppose that you execute a series of statements like:

```
UPDATE employee_table SET salary = salary * 1.10
WHERE state = 'California';
SAVE UPDATE employee_table SET salary = salary * 1.10
WHERE state = 'California';
```

Suppose also that the master database contains 200 employees who work in California, while the replica contains only the 100 employees who worked at the local branch office in San Francisco, California. In that case, the UDPATE command executed on the replica would apply only to the 100 employees contained in the replica's database, but the identical saved statement would apply to all 200 California employees listed in the master's database. When you propagate a command to the master, be careful to use a WHERE clause that ensures that the command applies only to the appropriate records.

6.6.3 Using the SmartFlow Parameter Bulletin Board

solidDB SmartFlow introduces a new parameter passing method, "Parameter Bulletin Board" for transactions to use for various purposes. Parameter Bulletin Board is a memory area to which one can add parameters of various kinds and from which the operations (procedures) of the transaction may read those parameters.

There are three different kinds of parameters that appear on the Parameter Bulletin Board.

- *Volatile Transaction Parameters* that are used within a transaction for transferring information between the different procedures of the transaction.
- *Transaction Properties* (persistent transaction parameters) that are used for giving a transaction some properties (i.e. describing the transaction) when the transaction is created in the replica database.
- *Persistent, catalog-level Sync Parameters* that are used for describing the catalog where the transaction is being executed.

Depending on the type of parameter, there is a different way to specify the parameter and a different mechanism for when and how the parameter is put to the parameter bulletin board. However, the mechanism for reading the parameters from the bulletin board is the same for all different types of parameters.

Passing Parameters between Procedures within a Transaction

Procedures of a transaction may communicate with each other by putting volatile parameters on the parameter bulletin board using the `PUT_PARAM()` function and reading the parameters using the `GET_PARAM()` function. Each parameter is a name-value pair. If a parameter already exists in the bulletin board, the `PUT_PARAM()` function replaces the current value with a new one. If a parameter doesn't exist in the bulletin board, then `GET_PARAM()` returns NULL.

Below are some examples of writing and reading parameters:

```
-- Procedure P1 sets a bulletin board sync parameter.
"CREATE PROCEDURE P1()
BEGIN
PUT_PARAM('CreditLimitExceeded', 'Y');
...
END";

-- Procedure P2 reads the bulletin board sync parameter.
"CREATE PROCEDURE P2()
BEGIN
DECLARE cred_lim_exceeded CHAR(1);
cred_lim_exceeded := GET_PARAM('CreditLimitExceeded');
...
END";
```

Note that you may use `GET_PARAM()` not only to read the sync parameter values set by the `PUT_PARAM()` command, but also to read the transaction properties that were set with the SAVE PROPERTY command.

The parameter bulletin board is visible to all statements of the transaction. This allows different stored procedures, for example, to communicate with each other even if they don't call each other. If one procedure detects an error, it can set a flag that will notify subsequent procedures in that same transaction to skip processing of the erroneous data.

The parameters appear on the parameter bulletin board of the transaction when the transaction is executed in the master database. The parameters are also visible to the replica while the transaction is executing on the replica.

In most cases, the transaction properties are used when the transaction executes on the master, i.e. after the transaction has been propagated from the master to the replica. However, you may use the values on either the replica or the master (or both).

Note

When implementing intelligent transactions for conflict resolution, be sure to set autocommit OFF to prevent losing transaction properties. The life cycle of a transaction parameter is one transaction; that is, it is visible only in the transaction that has set the value. If autocommit is ON, then each statement is a separate transaction and bulletin board values are lost immediately.

Refer to the *solidDB SQL Guide* for details on `PUT_PARAM()` and `GET_PARAM()`.

Assigning Properties to a Replicated Transaction

Transaction properties are used for describing an entire replicated transaction. These parameters are persistent parameters that are defined in the replica database using SAVE PROPERTY statement and who persist until the transaction to which they are attached, has been successfully propagated to the master database and executed there. The SAVE PROPERTY command stores the parameter for a propagateable transaction in the replica database. When the propagated transaction is later executed in the master, these parameters are put to the parameter bulletin board (in the master db) in the beginning of the transaction. Any procedure of the transaction may query the value of this parameter using the `GET_PARAM()` function.

The transaction properties can be used for two purposes.

- act as a selection criteria for selecting, which transactions to propagated in a synchronization message
- internally by the procedures of the transaction when it is executed in the master database.

The syntax for saving a property to a transaction is:

```
SAVE PROPERTY property_name VALUE property_value
```

In the excerpt from Example 6.5, “Create Order Transaction”, the transaction has one saved property with the name 'priority' and value '1'.

```
-- Save a parameter to the transaction
SAVE PROPERTY priority VALUE '1';
```

This parameter can be used as a search criterion of the transaction propagation process ("propagate only those transactions that have parameter 'priority' with value '1'"). For example:

```
MESSAGE APPEND PROPAGATE TRANSACTIONS WHERE priority = '1';
```

When the transaction has been propagated to the master database, the values of all defined properties of this transaction appear on the parameter bulletin board of the transaction when the transaction is executed in the master database. Thus, you can query the value of the 'priority' property from a procedure using the `GET_PARAMI()` function.

```
DECLARE priority_value CHAR(1);
priority_value := GET_PARAM('priority');
```

This information may be used for the application's own purposes, e.g. for determining how a possible update conflict should be resolved with this particular transaction.

Defining catalog-level persistent synchronization parameters

To define a parameter that has a per-catalog scope, you use SET SYNC PARAMETER command. This command specifies a parameter that can be read by any transaction (using the `GET_PARAM()` function) that executes within that catalog. Each transaction in that catalog can also alter the value of the parameter, but those updated values are only seen within the current transaction. Subsequent transactions do not see the updated values; subsequent transactions only see the "original" values.

SmartFlow System Parameters

The SmartFlow system itself pre-defines some parameters known as "system parameters". System parameters are something that the server knows about and can act upon (e.g. by terminating transaction). (The generic catalog-scope parameters are something the application-level intelligent transactions must know about. They don't affect the server's own behavior.) An example of a system parameter is SYS_ROLLBACK. Setting *`SYS_ROLLBACK`* to 'YES' causes the server to terminate that transaction. The value of *`SYS_ROLLBACK`* is reset back to default 'NO' for the next transaction. System parameters are merely parameters that use names that are reserved by the system. System parameters are visible on the bulletin board, just like any other parameters. Your transaction can read and write system parameters just as it can read and write other parameters.

6.6.4 Creating Stored Procedures

To ensure the physical and logical database consistency, you must write stored procedures using the Solid SQL stored procedure language. These procedures can use the parameter bulletin board to communicate with each other. In Example 6.6, “INSERT_ORDER Stored Procedure”, the account balance update must not be made if the insert of the new order fails.

Following is a simplified example of the procedures called in Example 6.5, “Create Order Transaction”.

Example 6.6. INSERT_ORDER Stored Procedure

```
"CREATE PROCEDURE INSERT_ORDER
   (ORDER_ID VARCHAR, CUST_ID VARCHAR, ...)
BEGIN
DECLARE ORDER_FAILED VARCHAR ;
DECLARE CUST_OK INTEGER ;
DECLARE STATUS VARCHAR ;
ORDER_FAILED := 'N' ;
STATUS := 'OK' ;

-- Validate the order
-- For instance, it must have a valid customer
EXEC SQL PREPARE CHECK_CUST
CALL CHECK_CUSTOMER (?) ;
EXEC SQL EXECUTE CHECK_CUST
USING (CUST_ID)
INTO (CUST_OK) ;
IF CUST_OK = 0 THEN
ORDER_FAILED := 'Y' ;
STATUS := 'FAIL' ;
END IF ;
-- Other validation checking should go here...
-- ...
-- End of validation

-- If the validation fails, put a parameter to the bulletin board to
-- inform subsequent stored procedures about the validation failure.
IF ORDER_FAILED = 'Y' THEN
PUT_PARAM('ORDER_FAILED', 'Y');
END IF;

-- Insert the order row into the database. The STATUS value in the
-- row may be either 'OK' or 'FAIL'.
EXEC SQL PREPARE INS_ORD
INSERT INTO CUST_ORDER (ORD_ID, CUST_ID, STATUS, ...)
VALUES (?,?,? ...);
EXEC SQL EXECUTE INS_ORD
USING (ORD_ID, CUST_ID, STATUS...);
EXEC SQL CLOSE INS_ORD;
```

```
EXEC SQL DROP INS_ORD;
END";
```

The following procedure updates the balance of a given account:

```
"CREATE PROCEDURE UPDATE_CUST_CREDIT (ACC_NUM VARCHAR, AMOUNT FLOAT)
BEGIN
DECLARE ORDER_FAILED VARCHAR;
-- Check from the bulletin board, whether the order
-- was inserted/modified successfully.
-- In case of failure, don't update the account balance
ORDER_FAILED := GET_PARAM('ORDER_FAILED');
IF ORDER_FAILED = 'Y' THEN
   RETURN
END IF;

EXEC SQL PREPARE UPD_CREDIT
   UPDATE ACCOUNT
   SET BALANCE = BALANCE + ?
   WHERE ACC_NUM = ?;
EXEC SQL EXECUTE UPD_CREDIT
USING (AMOUNT, ACC_NUM);
EXEC SQL CLOSE UPD_CREDIT;
EXEC SQL DROP UPD_CREDIT;
END";
```

6.6.5 Creating a Synchronization Error Log Table for an Application

It is highly recommended that a synchronization error log table be created as part of the database schema of a decentralized system. The log table allows application developers to store information about the application-level errors that may occur during synchronization.

If the error requires manual resolution, the error log serves as a source of information required to correct the error. For instance, if an update conflict occurs, a log can contain information on which row was conflicting and what was done to the conflicted row by the transaction. Below is an example of an error log table:

```
CREATE TABLE ERRLOG
   (ID CHAR(20) NOT NULL,
   LOG_TITLE CHAR(80) NOT NULL,
```

```
LOG_DESCRIPTION VARCHAR NOT NULL,
UPDATETIME DATETIME NOT NULL,
PRIMARY KEY (ID));
```

The ID is a generated, globally unique key of the log entry.

The LOG_TITLE and LOG_DESCRIPTION columns contain the information about the error, such as an update conflict.

The UPDATETIME column contains the timestamp of the last update operation of the error log row.

6.7 Validating Intelligent Transactions

Data synchronization brings a new aspect to the functionality of the business application. The major difference between centralized and decentralized systems is the existence of tentative data in decentralized systems. There can be multiple different versions of the same data item in the system.

The master database has the officially correct version of the data. The replicas may have a different, unofficial version of the same data. When a replica transaction (which is based on the unofficial replica version of data) is propagated to the master database, transaction validation errors such as update conflicts may occur in the master. In this kind of situation, transactions need to act in a way that meets the requirements of the business rules of the application.

When a transaction validation error occurs, there are different options for handling the situation:

- Resolve the error automatically without user intervention. For instance in the case of an update conflict, select the most recent update.
- Leave the master version of the data intact and save a sufficient amount of information about the conflicted or otherwise erroneous operation to allow error correction through manual user intervention.

The first approach does not require anything special from the data model, because transaction validation errors are automatically resolved as they occur and do not require manual attention.

However, the first approach does not always take into consideration all imaginable transaction validation errors, some of which cannot be resolved automatically. For example, if an order is updated in both the master and the replica databases of an Order Entry system, it is very hard to determine automatically which one of the updates is the correct one. Sometimes, user intervention is required to fix the error.

In order to enable the user to fix the error, a sufficient amount of information about the failed transaction must be made persistent. This information can be stored in a separate error log table or the data model can be designed to accommodate multiple versions of the same data item.

6.7.1 Designing Complex Validation Logic

Although transactions of business applications are typically small groups of atomic read and write operations, they can also be complex. For example, in a decentralized Order Entry system, the transaction may contain the following operations:

- insert a row into CUST_ORDER table
- insert multiple rows into ORDER_LINE table
- update USED_CREDIT information of CUSTOMER table
- insert an entry into the ACCOUNT_TRANSACTION table of the bookkeeping part of the application
- for each type of product ordered, update the STOCK_BALANCE column of a row of the PRODUCT table to update the warehouse balance of the ordered product

In this example, various things can go wrong when the transaction is executed in the master database. For instance:

- According to the master database, the customer does not have sufficient credit available for the order. Therefore the entire order is invalid or must be separately approved.
- There is insufficient inventory given the amount of the products ordered. This makes part of the order incomplete. It may be that the entire order must be put on hold or another order (back order) is required for the missing product.
- The accounting transaction is invalid because the transaction has been propagated after the month end when entries to the previous month are no longer allowed. This creates a mismatch between corporate bookkeeping and order systems.

There are different approaches to solving these challenges when using Solid Intelligent Transaction. Two are discussed here: pre-validation and compensation.

Pre-validation

The individual operations of the transaction can be split into two parts: validation operations and writing operations.

You can implement the transaction so that the validation parts are executed before any of the writing parts. In the parameter bulletin board, the validation parts leave all the necessary information for the writing parts to behave correctly.

In the above example, the validation part of the transaction could look as follows:

```
VALIDATE_ORDER
VALIDATE_ORDER_LINE (multiple)
VALIDATE_CREDIT_UPDATE
VALIDATE_ACCOUNT_TRANSACTION
VALIDATE_STOCK_UPDATE
```

At this point, no write operations have been made yet, but the parameter bulletin board now has all the information that the write operations need in order make the whole transaction valid. The rest of the transaction would look as follows:

```
INSERT_ORDER
INSERT_ORDER_LINE (multiple)
UPDATE_CUSTOMER_CREDIT
INSERT_ACCOUNT_TRANSACTION
UPDATE_PRODUCT_STOCK_BALANCE
```

Compensation

Another way to solve the above problem is to add compensating operations in the end of the transactions. They can be used, for instance, in a scenario where a product to be ordered in one of the order lines does not exist any more. Therefore the entire order becomes incomplete. However, the row to the ORDER table has already been inserted with STATUS column value 'OK'.

In this case, the VALIDATE_AND_INSERT_ORDER_LINE must leave a parameter on the bulletin board that informs the last operation (COMPENSATE_ORDER) of the transaction required to change the status of the order to 'INVALID'. The overall Intelligent Transaction implementation could in this example case look as follows:

```
VALIDATE_AND_INSERT_ORDER
VALIDATE_AND_INSERT_ORDER_LINE (multiple)
VALIDATE_AND_UPDATE_CUSTOMER_CREDIT
VALIDATE_AND_INSERT_ACCOUNT_TRANSACTION
```

```
VALIDATE_AND_UPDATE_PRODUCT_STOCK_BALANCE
COMPENSATE_ORDER
```

6.7.2 Error Handling in the Application

Because the synchronization of databases is done by the application, the error handling must also be implemented in the application, that is, in the stored procedures. Transactions can generate errors that appear either at the system or at the application level. The system level errors are typically fatal; that is, they cannot be recovered automatically.

Application level error occurs when the original behavior of the transaction is no longer valid. Typically this occurs when a conflict is detected. For instance, an order is inserted for a customer who no longer exists in the master database.

There are numerous options on how to recover from this kind of error:

- Resolve the error automatically inside the transaction using a conflict resolution rule, such as "the current master version wins."
- Leave the resolution of the error to the user or system administrator. An error log table can be used for storing sufficient information about the error.
- Combinations of the above options — for example, resolve the conflict automatically but inform a user about the resolution using an error log table.

The method to use depends on the application and its requirements.

6.7.3 Specifying Recovery from Fatal Errors

The most important rule of error handling is that all transactions must commit in the master database. Any DBMS error is a fatal error and by default, causes the execution of the synchronization message to halt. These errors are system-level errors. For instance, if a write operation fails in the master database during synchronization because of a unique constraint violation, then the execution of the message is halted and an error code is returned to the replica database.

If a fatal error is detected by the business logic of a transaction, the transaction can be aborted and the further execution of the synchronization message halted by putting a rollback request to the bulletin board of the transaction. Remember that the COMMIT WORK and ROLLBACK WORK statements are not allowed in propagated transactions. However, the rollback request can be issued with the following system bulletin board parameters:

```
SYS_ROLLBACK = 'YES'
```

```
SYS_ERROR_CODE = user_defined_error_code
SYS_ERROR_TEXT = user_defined_error_text
```

The SYS_ROLLBACK parameter is a system-recognized parameter. If the transaction sets the value to 'YES', then the server will automatically roll back the transaction. The previously committed transactions of the same synch message will remain committed. What happens to the rest of the synchronization message depends on the mode of the PROPAGATE TRANSACTIONS operation. (See the section called “IGNORE_ERRORS, FAIL_ERRORS, and LOG_ERRORS flags for Propagate Transactions command” for details about the error-handling mode of the PROPAGATE TRANSACTIONS operation.)

Here is a sample scenario:

Let us assume that a transaction detects a violation of referential integrity of the database; for example, the customer of an order does not exist in the master database. The transaction can put the following parameters on the bulletin board in order to request rollback and return application-specific error codes:

```
Put_param('SYS_ROLLBACK', 'YES');
Put_param('SYS_ERROR_CODE', '90001');
Put_param('SYS_ERROR_TEXT', 'Referential integrity violation detected');
```

Note that because the transaction management of transactions is done outside the procedure, directly issuing the ROLLBACK WORK or COMMIT WORK command inside the procedure is never allowed.

IGNORE_ERRORS, FAIL_ERRORS, and LOG_ERRORS flags for Propagate Transactions command

If an error occurs during transaction propagation, the default behavior is that the server stops processing the message and aborts the current transaction. Any previous transactions in the message remain in effect. This means that you may wind up with just part of a message executed.

solidDB supports three error-handling modes for propagated messages.

- IGNORE_ERRORS - This option means that if an error occurs, the transaction is aborted. Execution continues with the next transaction. In other words, an error does not abort the entire message.
- LOG_ERRORS - Like IGNORE_ERRORS, this option means that if an error occurs, the transaction is aborted, and execution continues with the next transaction. In addition, failed transaction statements are saved in SYS_SYNC_RECEIVED_STMTS system table for later execution or investigation.
- FAIL_ERRORS - This option means that if a statement fails, the current transaction is rolled back, and the server does not continue on to process subsequent transactions in the same message. (Any transactions that have already been committed are not undone.) This is the default error-handling mode for propagation.

There are three ways to specify which of these error-handling modes you want to apply to a particular message or transaction.

- Use an appropriate keyword in the SAVE command. If you specify the error-handling mode in the SAVE command, then the specified mode applies only to that saved statement (not the entire transaction or the entire message).
- Use a transaction bulletin board parameter to specify the error-handling mode. In this case, the specified mode applies to the current transaction.
- Use a MESSAGE APPEND PROPAGATE TRANSACTIONS command to specify the error behavior. In this case, the specified mode applies to the entire message.

If error-handling options are specified in both the SAVE and the PROPAGATE TRANSACTIONS statement, then the error-handling options specified in the PROPAGATE TRANSACTIONS statement take precedence.

The failed messages can be examined using SYNC_FAILED_MESSAGES system view and they can be re-executed from there using statement MESSAGE *<msg_id>* FROM REPLICA *<replica_name>* RESTART *<error_options>*.

The syntax for setting values in the parameter bulletin board is shown below.

The parameter name for error handling is

```
SYNC_DEFAULT_PROPAGATE_ERRORMODE
```

The values can be:

```
IGNORE_ERRORS
LOG_ERRORS
FAIL_ERRORS
```

The parameter name used for autosave in the master is:

```
SYNC_DEFAULT_PROPAGATE_SAVEMODE
```

The values can be:

```
AUTOSAVE
AUTOSAVEONLY
```

Autosave is used in hierarchies that have more than two levels. If a replica needs to propagate a transaction not to its direct master, but to a master above that, then the replica may use AUTOSAVE. AUTOSAVE is discussed in more detail later in this chapter.

For example,

```
PUT_PARAM('SYNC_DEFAULT_PROPAGATE_ERRORMODE', 'LOG_ERRORS');
PUT_PARAM('SYNC_DEFAULT_PROPAGATE_SAVEMODE', 'AUTOSAVE');
```

The syntax for setting values in messages is shown below.

```
MESSAGE <message_name> APPEND PROPAGATE TRANSACTIONS
[ { IGNORE_ERRORS | LOG_ERRORS | FAIL_ERRORS } ] [WHERE ...]
```

Note that the autosave option is not possible in this statement.

Syntax (in save):

```
SAVE [NO CHECK] [ { IGNORE_ERRORS | LOG_ERRORS | FAIL_ERRORS } ]
[ { AUTOSAVE | AUTOSAVEONLY } ] <sqlstring>
```

- NO CHECK: This option means that the statement is not prepared in the replica. This option is useful if the command would not make sense on the replica. For example, if the SQL command calls a stored procedure that exists on the master but not on the replica, then you do not want the replica to try to prepare the statement. If you use this option, then the statement can not have parameter markers.

- IGNORE_ERRORS: This option means that if a statement fails while executing on the master, then the failed statement is ignored and the transaction is aborted. However, only the transaction, not the entire message, is aborted. The master continues executing the message, resuming with the first transaction after the failed one.

- LOG_ERRORS: This means that if a statement failed while executing on the master, then the failed statement is ignored and the current transaction is aborted. The failed transaction's statements are saved in SYS_SYNC_RECEIVED_STMTS system table for later execution or investigation. The failed transactions can be examined using SYNC_FAILED_MESSAGES system view and they can be re-executed from there using MESSAGE <*msg_id*> FROM REPLICA <*replica_name*> RESTART statement.

Note that, as with the IGNORE_ERRORS option, aborting the transaction does not abort the entire message. The master continues executing the message, resuming with the first transaction after the failed one.

- FAIL_ERRORS: This option means that if a statement fails, the master stops executing the message. This is the default behavior.
- AUTOSAVE: This option means that the statement is executed in the master and automatically saved for further propagation if the master is also a replica to some other master (i.e. a middle-tier node).
- AUTOSAVEONLY: This option means that the statement is NOT executed in the master but instead is automatically saved for further propagation if the master is also a replica to some other master (i.e. is a middle-tier node).

When a master database is propagating a message, the autosave setting is ignored if the node is not also a replica. In other words, the autosave setting is ignored for the topmost master in the hierarchy. The setting affects one node only, e.g. all nodes must set it separately if needed.

Example:

```
SAVE NO CHECK IGNORE_ERRORS insert into mytab values(1, 2)
```

The table sys_sync_master_msginfo has a new column FAILED_MSG_ID which is part of the primary key. The value is zero for normal messages. The value is msg_id if LOG_ERRORS option is ON and any errors exists. Also the SYS_SYNC_RECEIVED_STMTS table has errcode and err_str columns where actual errors are logged.

The autosave option takes effect if defined in SAVE or if defined in the master bulletin board.

Note

If a replica registration or a publication registration message fails, the hung messages are automatically deleted from both the master and the replica, and the status is reset back to what it was before the message execution. The IGNORE_ERRORS, FAIL_ERRORS, and SAVE_ERRORS flags do not apply to these two types of messages.

Re-executing or Deleting Logged Errors in Master

If transactions were propagated with the LOG_ERRORS option, then the server saves the statements that were in the transaction that failed. These statements can be examined, and in some cases the cause of the problem can be fixed and then the failed statements can be re-executed.

MESSAGE *msgid* FROM REPLICA *replicaname* RESTART <*err-options*>;

Where <*err-options*> can be IGNORE_ERRORS or LOG_ERRORS or FAIL_ERRORS

If the problem cannot be corrected, or if you choose not to re-execute the transactions that failed, then you may delete the entire message, or delete one or more transactions within that message.

MESSAGE *msgid* FROM REPLICA *replicaname* DELETE;

MESSAGE *msgid* FROM REPLICA *replicaname* DELETE CURRENT TRANSACTION;

Important

Do not use command:

```
MESSAGE message_name FROM REPLICA replicaname DELETE;
```

as the deletion command. The reason is that message_name is not valid any more after the message has completed its replica-perceived life cycle. *message_name* is removed from the system, for example, after the command

```
MESSAGE ... GET REPLY
```

However, *msgid* applies for as long as the message is logged at the master.

Creating and Sending a Synchronization Message from a Propagated Transaction

You can create and send a synchronization message from a propagated transaction. Although in most situations explicit commits are not allowed in a propagated transaction, they are allowed when creating and sending a synchronization message from inside a propagated transaction. You may issue an explicit COMMIT in the middle of a propagated transaction if the transaction has not executed any DML statements and if the previous statement was one of the following:

```
MESSAGE ... END
MESSAGE ... FORWARD, or
MESSAGE ... GET REPLY
```

Chapter 7. Updating and Maintaining the Schema of a Distributed System

This chapter is divided into two major sections. The first discusses some of the "mechanics" of using SmartFlow. The second discusses the specific issue of upgrading a schema across a distributed system — that is, upgrading a schema that is shared across as master and its replicas.

7.1 Managing solidDB Tables and Databases

After your initial implementation of a SmartFlow system, you may need to alter your database schema, add a new master, or drop replicas. This section provides step-by-step procedures to manage tables and databases. Before you perform database maintenance, be sure to close all database connections. Depending on what type of maintenance you are performing, you may want to be sure all databases are synchronized.

7.1.1 Modifying the Database Schema

You can modify the database schema in the master database (see the CREATE OR REPLACE PUBLICATION command, which allows you not only to create new publications, but also to modify existing publications). You may also modify indexing and user access rights even if the table is referenced by publications.

7.1.2 Changing Master or Replica Database Location

You can easily change the database location of a master and replica by copying the database and log files to the target directory.

1. Shut down the server to release the operating system file locks on the database file and log files.
2. Copy the database and log files to the target directory.
3. Copy the `solid.ini` file to the target directory.
4. Check that the database file directory, log file directory, and backup directory, are correctly defined in the `solid.ini` configuration file.
5. If the database you moved is the master, issue the SET SYNC CONNECT command in all the replicas.

   This must be set in all replicas to ensure connection before the next message to the master.

6. Start solidDB at the new location using the target directory as the current working directory with the command line option **-c *directory_name***.

7. If the database you moved is a replica, check that you are able to access the master with the connection string in MESSAGE FORWARD or MESSAGE APPEND REGISTER REPLICA.

7.1.3 Unregistering a Replica Database

When a replica database is not used in a SmartFlow system, it is strongly recommended that it is unregistered, i.e. the synchronization relationship between the replica and master databases is removed. After unregistering the replica, the master database knows that it does not need to accumulate synchronization history data for this replica database any more. This may save significant amounts of disk space in the master database.

To unregister a replica database:

1. Drop the subscription(s) in the replica if data is no longer needed in the replica database.

2. In the replica, use the following command to unregister the replica with the master database:

```
MESSAGE message_name BEGIN;
MESSAGE message_name APPEND UNREGISTER REPLICA;
MESSAGE message_name END;
COMMIT WORK;
```

3. Send the message to the master database.

```
MESSAGE message_name FORWARD TIMEOUT seconds;
COMMIT WORK;
```

   The replica database can no longer synchronize with the master database.

A replica database can also be dropped from the master database using the following command in the master database:

```
DROP REPLICA replica_name;
```

If you drop the replica using DROP REPLICA, you should also drop the master from the replica with DROP MASTER.

This method may be necessary if access to the master database needs to be denied from the replica database or the replica database was not able to successfully unregister itself.

7.1.4 Creating Large Replica Databases

When creating large replicas that are greater than 2GB from a master database, use the EXPORT SUBSCRIPTION and IMPORT commands. However, when synchronizing BLOb data, the limitation of 2GB per synchronization message doesn't apply. You can export any subscription from a master database to a file and later import that file to a replica. For details, see Section 7.1.5, “Managing Data with Synchronization Bookmarks”.

Note

A replica can be created by subscribing to a publication from a master database, but because data is sent in one logical chunk, this method is problematic for replicas larger than 2GB unless the data is BLOb data. Downloading large amounts of data may require a substantial amount of time, and there are limitations in sending the large data in a single SmartFlow message.

7.1.5 Managing Data with Synchronization Bookmarks

A synchronization bookmark is a state of the solidDB database that you define for future reference purposes. Bookmarks are created with the following command in the master database:

```
CREATE SYNC BOOKMARK bookmark_name
```

This command automatically associates other attributes such as creator of the bookmark, create date and time, and unique bookmark ID.

Bookmarks are more than placeholders in the database. In a sense, a bookmark can be thought of as a user-defined persistent snapshot of a database. They are used also to export data from the master and import data into a replica. For more information, read Section 7.1.6, “Exporting and Importing Subscriptions”.

Bookmarks are created only in the master database. You cannot create a bookmark in a replica database. Attempting to do this results in an error. Note that there is no practical limit to the number of bookmarks you can create in a database.

Creating bookmarks requires Database Administrator (DBA) or solidDB Administrator (SYS_SYNC_ADMIN_ROLE) privileges.

Bookmarks retain history information for all tables that have history versions defined. For this reason, it is recommended that bookmarks be dropped when they are no longer needed. Otherwise, disk space usage increases considerably to accommodate the extra history versions.

Retrieving Bookmark Information

Before creating a new bookmark or dropping an existing bookmark, you can query the solidDB catalog table SYS_SYNC_BOOKMARKS to see a list of existing bookmarks. For example, the following query gives the bookmark name, creation date, and the creator of the bookmark:

```
SELECT BM_NAME, BM_VERSION, BM_CREATOR, BM_CREATIME FROM SYS_SYNC_BOOKMARKS;
```

Administrative privileges are not required to retrieve bookmarks from the system catalog table.

Dropping Bookmarks

Bookmarks are dropped with the following command in the master database:

```
DROP SYNC BOOKMARK bookmark_name
bookmark_name ::= literal
```

Note

Bookmarks should only be dropped after the exported data is imported into all intended replicas, not just one. After the import, the replicas need to refresh the imported subscription once before the bookmark may be dropped in the master. Drop a bookmark only when you no longer have any replicas to import.

After using a bookmark to successfully import a file to a replica or to receive the first refresh of the data from the master database, it is recommended that you drop the bookmark that you used to export the data to a file. For details on importing and exporting subscriptions, see the following section.

If a bookmark remains, then all subsequent changes to data on the master including deletes and updates are tracked on the master database for each bookmark to facilitate incremental refreshes.

By dropping a bookmark, you allow the server to delete the history data that was required by the bookmark. If you do not drop bookmarks, more disk space is consumed for each bookmark registered in the master database. This may result in performance degradation.

For more details, see the description of the SQL command DROP SYNC BOOKMARK in the *solidDB SQL Guide*.

7.1.6 Exporting and Importing Subscriptions

The SmartFlow EXPORT SUBSCRIPTION command let you export a version of the data from a master database to a replica database or to a disk file. If the data is exported to a disk file, it can be imported into a replica database with IMPORT command. These commands assume you have created subscriptions in your database and bookmarks to reference the state of the database you want to export.

Specifying a Subscription for Export

The concept and procedures for using the EXPORT SUBSCRIPTION command are similar to refreshing from a publication. You use the EXPORT SUBSCRIPTION command instead of the MESSAGE APPEND REFRESH command in the following circumstances:

- You need to create a large replica database from an existing master. This procedure requires that you export a subscription with or without data to a file first, then import the subscription to the replica. For details, read the section called “Creating a Replica by Exporting a Subscription with Data” or the section called “Creating a Replica by Exporting a Subscription without Data”.
- You want to export specific versions of the data to a replica.
- You want to export metadata information only without the actual row data.

Note the difference between using the EXPORT SUBSCRIPTION command vs. refreshing from a publication with MESSAGE APPEND REFRESH:

- The EXPORT SUBSCRIPTION command is executed in the master, whereas a refresh is requested from a replica.
- The export output is saved to a user specified file, whereas output of a REFRESH command is stored in a SmartFlow reply message.
- The export file can be created with no data (actual rows are not included in output) as well as with data.
- The export file is never incremental (for example, if the data for the export contains rows, all rows are included in the export file, as in a refresh based on a full publication).
- The export file is based on a given bookmark; this means that export data is consistent up to a given bookmark and refreshes based on incremental publications are possible from that bookmark.

EXPORT SUBSCRIPTION COMMAND

There are two different ways to export data from master to replica database.

If you want to export data of a subscription from the master database to a file for later import to one or multiple replicas, use the following EXPORT SUBSCRIPTION syntax:

```
EXPORT SUBSCRIPTION publication_name [( arguments )]
   TO 'filename'
   USING BOOKMARK bookmark_name
   [WITH [NO] DATA];
```

Once the export operation to a file has completed, the data of the file can be imported to a replica database with IMPORT command.

If you want to export data from a master database directly to specified existing replica database, use the following syntax:

```
EXPORT SUBSCRIPTION publication_name [( arguments )]
   TO REPLICA replica_name
   USING BOOKMARK bookmark_name
   [COMMITBLOCK #rows] ;
```

Note that EXPORT SUBSCRIPTION TO REPLICA command does not use files as the means to transfer data from master to replica database. Instead, it writes the data to the replica database directly. Hence, no separate import step is needed. The replica database must exist and be available in the network.

The *publication_name* and *bookmark_name* are identifiers that must exist in the database. The filename represents a literal value enclosed in single quotes. An export file can contain more than one subscription. You can export subscriptions "WITH DATA" and "WITH NO DATA" options. If there is more than one publication specified, the exported file can have a combination of "WITH DATA" and "WITH NO DATA."

For more details, including rules for usage, read the description of the SQL commands "EXPORT SUBSCRIPTION TO <*file*>" and "EXPORT SUBSCRIPTION TO REPLICA" in the *solidDB SQL Guide*. For the procedure to create a replica WITH DATA, read read the section called “Creating a Replica by Exporting a Subscription with Data” and to create a replica WITH NO DATA, read the section called “Creating a Replica by Exporting a Subscription without Data”.

Specifying a Subscription for Import

The IMPORT command is used on a replica database to import the data from a data file created by the EXPORT SUBSCRIPTION command.

IMPORT Command

The IMPORT command is created using the following syntax:

```
IMPORT 'filename' [COMMITBLOCK #rows]
```

The *filename* represents a literal value enclosed in single quotes. The import command can accept a single filename only. All publication data for import to a replica must fit in a single file. However, you can use multiple import statements to import multiple files.

The *#rows* is an integer value used with the optional COMMITBLOCK clause to indicate the commitblock size.

The COMMITBLOCK clause indicates the number of rows processed before the data is committed. If COMMITBLOCK is not specified, the IMPORT command takes all rows in the publication as one transaction. If the file contains a large number of rows, the use of COMMITBLOCK is recommended.

For more details, including rules for usage, read the description of the "IMPORT" command in the *solidDB SQL Guide*.

Creating a Replica by Exporting a Subscription with Data

When you have an existing (replica) database that needs a subset of the master's data but does not yet have it, then you use the EXPORT SUBSCRIPTION command using the WITH DATA option to export data for the replica.

The following procedure requires that you include data in the export file(s) and load the data to the replica with the IMPORT command.

Note

- When using the EXPORT SUBSCRIPTION command, you can export data to the same file more than once. With each command, the data is appended to the file. If you plan to do this, be sure you have enough disk space to accommodate the exported data for each export command. If you run out of data in the middle of the export, the EXPORT SUBSCRIPTION command will fail with an error and the export file will not be usable.
- solidDB requires that autocommit be set OFF when using the EXPORT SUBSCRIPTION command.

Procedure at a Master

Perform these steps in the master database:

1. Create a bookmark if one does not exist. If a bookmark already exists and meets your needs, you can use it. Refer to Section 7.1.5, “Managing Data with Synchronization Bookmarks” for information on creating bookmarks.

   You can also perform queries to see what bookmarks and publications currently exist in your system. Refer to the section called “Retrieving Bookmark Information”.

2. Execute the EXPORT SUBSCRIPTION command WITH DATA option for every needed publication to create export file(s).

   If a bookmark is associated with more than one publication at the master, then be sure to execute the EXPORT SUBSCRIPTION commands for each publication separately.

   For each EXPORT SUBSCRIPTION command WITH DATA option, the metadata and versioned data corresponding to that publication and bookmark are added to the export file.

Error Messages

If you receive an error that you have run out of disk space, delete the previous file and execute the EXPORT SUBSCRIPTION command again with sufficient disk space.

Note

You cannot suspend and resume an EXPORT SUBSCRIPTION command. If the execution did not complete for some reason, you need to execute the EXPORT SUBSCRIPTION command again.

Possible errors you may encounter include:

- Error message 25067, indicating that the SmartFlow bookmark could not be found. Check to see that you have entered the bookmark name correctly.
- Error message 25068, indicating that the filename you specified in the EXPORT SUBSCRIPTION or IMPORT command cannot be opened or cannot be opened in the append mode. Check to see that you have entered the filename correctly and it is not currently in use.

Procedure at a Replica

Perform these steps at the replica site:

1. Register this replica with the master database.
2. Set the replica catalog to be current catalog using SET CATALOG command.
3. Register the publication whose subscription(s) will be imported.

4. Import the file(s) you created using the EXPORT SUBSCRIPTION command. The procedure for import is described in the section called “Specifying a Subscription for Import”.

   The IMPORT command accepts only one file at a time. If you have multiple export files, execute a separate IMPORT command for each file. Remember that one file may include multiple exports.

   You can import the same subscription to the same replica more than once as this has the same effect as subscribing with the FULL publication option.

   Note

   You cannot suspend and resume an IMPORT command. If the execution did not complete for some reason, you need to execute the IMPORT command again.

5. REFRESH the publication(s) from the master database using SmartFlow's MESSAGE commands (e.g. MESSAGE APPEND REFRESH).

Creating a Replica by Exporting a Subscription without Data

In some cases, you might have a database that already contains all the appropriate data, but is not already configured to be a replica. For example, you might have a server that has a backup copy of an existing master, and you might want to convert that copy to a replica. If the database already has the appropriate data, then you don't want to have to discard and then refresh all that data. One solution is to export just the subscription metadata from the master, then import that into the copy of the master (i.e. into the database that you want to convert to a replica). You use the EXPORT SUBSCRIPTION command with the WITH NO DATA option to export the schema (the "metadata").

Caution

Be sure valid data exists on the replica before you use this option, or you risk the consequence that an application accessing the replica will use the wrong set of data.

Note

solidDB requires that autocommit be set OFF when using the EXPORT SUBSCRIPTION command.

The following procedure requires that you export file(s) with no data and load the files containing the metadata and publication information to the replica with the IMPORT command.

Procedure at a Master

Perform these steps in the master database:

1. Create a bookmark if one does not exist. If a bookmark already exists and meets your needs, you can use it. Refer to Section 7.1.5, “Managing Data with Synchronization Bookmarks” for information on creating bookmarks.

   You can also perform queries to see what bookmarks and publications currently exist in your system. Refer to the section called “Retrieving Bookmark Information”.

2. Execute the EXPORT SUBSCRIPTION command WITH NO DATA option for every needed publication to create export file(s).

   Only one bookmark is required even if several publications are exported. You can export several publications to a single file by specifying the same file name.

   For each EXPORT SUBSCRIPTION command WITH NO DATA option, the metadata and history data corresponding to that publication and bookmark are added to the export file.

Procedure at a Replica

Perform these steps at each applicable replica site:

1. Create a backup of an existing master database (for example, by using the **ADMIN COMMAND 'backup'**) command.

2. Start the backup database and drop all replicas using the DROP REPLICA command and all publications using the DROP PUBLICATION command.

3. Change the node name using the **SET SYNC NODE *unique_node_name*** command (since the database is a backup of an existing master database and the original master is currently using that node name).

4. Configure the database as a replica by executing the following commands:

```
SET SYNC MASTER NO
SET SYNC REPLICA YES
```

Note

If you are switching the database to be a replica and the database is already a backup/copy of the master database, the backup is already set as a master database. If you execute the command SET SYNC REPLICA YES, then the database will be defined as a dual role (master and replica) database, instead of a dedicated replica-only database. If you want to make this database a replica

exclusively, then you must execute SET SYNC MASTER NO as well as SET SYNC REPLICA YES.

5. Register this replica to the master database.

6. Import the file(s) you created using the EXPORT SUBSCRIPTION command. The procedure for import is described in the section called “Specifying a Subscription for Import”.

7. Refresh from the publication(s) from the master database using SmartFlow's MESSAGE commands.

The newly created replica is ready for use. If this is the last replica you are creating, drop the bookmark from the master database as described in the section called “Dropping Bookmarks”.

7.1.7 Modifying Publications and Tables in Publications

You can modify existing publications. For more information, see the description of the command CREATE [OR REPLACE] PUBLICATION in *solidDB SQL Guide*. This command allows you not only to create new publications, but also to modify existing publications.

You may also modify tables that are in publications. For more information about this, see the description of the command SET SYNC MODE in *solidDB SQL Guide*.

Depending upon the modifications that you make, the next refresh from each replica may be either an incremental refresh or a full refresh.

Note

If you use CREATE OR REPLACE PUBLICATION to alter the contents of an existing SmartFlow publication, you have to take care of removing invalid rows from Replica.

Incremental vs. Full Refresh

If you change a publication using the command "CREATE OR REPLACE PUBLICATION..." then replicas will receive updated data the next time that they refresh. Depending upon the changes to the publication, subscribers may get a full refresh or an incremental refresh. Below is a summary of the rules that control whether the next refresh is incremental or full. Keep in mind that a single publication may contain multiple result sets. Each result set is relatively independent, so changes to one result set do not necessarily require that the subscriber get any information for the other result sets.

- Adding a new result set to a publication requires only an incremental refresh; the data of the new result set is sent to subscribers.

- Dropping a result set does not require sending data over the network.
- Modifying a result set generally requires full data of that result set.
- Of course, if you do drop a publication and then re-create it (rather than "replace" it using CREATE OR REPLACE), then subscribers must re-register with the new publication and will get a full refresh the first time that they refresh.

Note

You cannot switch a replica to use another existing master database. This would lead to a mismatch in synchronization of incremental publications. If you need to do this, you must create a database from scratch and define the new master database as well as re-create tables, procedures, and publications.

7.1.8 Modifying SQL Procedures of Intelligent Transaction

The SQL stored procedures of transactions are identified by their call interface only. Therefore you can freely modify them as long as the call interface remains the same. If the call interface (that is, the parameter list) of the procedure changes, then the name of the procedure must typically be changed also. In this case, the previous version of the procedure should be left available in the master database to ensure that all those transactions that are still on their way to master database, can be successfully executed there.

7.2 Upgrading the Schema of a Distributed System

7.2.1 Introduction

SmartFlow architecture allows you to distribute your data between a master and one or more replicas. The master creates publications, and replicas may subscribe to those publications.

In some cases, you may need to change the schema of your publication by altering the tables used by that publication. For example, you might want to add a new column to an existing table.

solidDB provides a set of schema management and synchronization features that allow you to change the schema of master and replica databases in your system. Schema management features in solidDB do not fully automate the schema upgrade process, but instead provide a powerful set of programming tools that make it possible to build a solution that best matches the application's needs. Hence, for example, creating a new table or altering an existing one in the master database does not mean that the table is also automatically created or altered in the replica database. solidDB lets the system administrator decide whether the table must be modified in the replica and, if so, when the modification should be done.

This chapter documents four features that are loosely categorized under the name "Maintenance Mode for Publications" (called "Maintenance Mode" for short). The primary purpose of these features is to reduce the effort required to upgrade the schema of a distributed system — i.e. to alter the structure of a table that is used in a publication.

If you do not use the Maintenance Mode features, you cannot change a table without dropping the publication referencing to the table first. Re-creating the publication forces all replicas to do a full refresh (rather than an incremental refresh) the next time that they refresh from a publication that contains that table. This is true even if the publication uses only a subset of the columns in that table, and the ALTER TABLE command you used did not affect any of those columns.

Using Maintenance Mode, you can often avoid forcing full refreshes after altering tables. Maintenance mode also allows large 'maintenance' updates to databases without causing large amounts of data to be synchronized after schema upgrade. Write operations made to a master database's data in Maintenance Mode are not synchronized at all. Instead, it is assumed that the same updates are done in the replica database using similar kind of script that modified the master database's schema and data.

This chapter provides an example of implementing the schema upgrade functionality. To understand most of this chapter, you should be familiar with the solidDB synchronization feature, including publications. For details about the various SQL statements used in the example, please refer to *solidDB SQL Guide*.

7.2.2 Major Features and Functionality

Below are the features that enable administrator-controllable version upgrades of a distributed system:

- Sync Mode: Setting the Sync Mode to "Maintenance" allows schema changes (DDL operations) on tables that are referenced by a publication. It also temporarily disables "sync history" tracking.
- The "REPLACE" option in the CREATE PUBLICATION command: This allows you to change an existing publication without necessarily requiring a full refresh afterward. When possible, incremental refreshes will still be allowed after the change.
- Table-Level Locking: You may explicitly lock an entire table. This makes it easier to change the schema.
- Schema Version Tracking: If either the master or replica sets the persistent catalog-level parameter `SYNC_APP_SCHEMA_VERSION` (using SET SYNC PARAMETER command), then the two servers will refuse to synchronize unless both have the same value for the version. This prevents synchronization when the schema of the replica does not match the schema of the master.

These four features are explained in more detail below. Later in this chapter, we will explain how to use these features to change the schema of a distributed system — i.e. to change the schema on both master and replica databases.

Sync Mode

Setting the Sync Mode to "Maintenance" allows schema changes (DDL operations) on tables that are referenced by a publication. If the catalog's sync mode is not Maintenance, then the server prohibits DDL operations on tables that are used by publications. This means that to change a table, you would have to drop the publication, change the table, and re-create the publication — even if the change to the table (e.g. adding a new column) did not affect the publication. Since dropping and re-creating a publication forces replicas to get a full refresh rather than an incremental refresh, you would have to force replicas to get full refreshes every time that you wanted to change a table.

Setting the Sync Mode to Maintenance allows you to alter a table without dropping the publication(s), and therefore without necessarily forcing full refreshes.

Setting the Sync Mode to Maintenance also temporarily disables sync history; in other words, it tells the server not to store data that is used in deciding which records to send to a replica when the replica requests a refresh of the data. This allows you to do some types of major data changes (DML) quickly; you can make the same updates to the master and replica and simply skip over the synchronization step.

However, if you accidentally make the master and replica "out of sync" while you have disabled sync history, the master and replica will not automatically re-synchronize (correct the error) the next time that the replica refreshes. Since there is no synchistory to show what changes were made on the master, the master has no reason to send updates to the replica. If the replica gets out of sync, it may stay out of sync indefinitely or until the next full refresh of the publication. You must be very careful when making changes to the master and replica databases when you have the Sync Mode set to Maintenance.

When you set the Sync Mode back to Normal (the default value), the server will resume tracking sync history information, and will also stop allowing DDL operations on tables in publications.

Note that Sync Mode Maintenance does not guarantee that replicas won't be required to get a full refresh when a table is changed. Some changes to a table (for example, dropping a column that is used in a publication) may affect the table (or the publication) enough that the replicas will have to get a full refresh.

The "REPLACE" Option in the CREATE PUBLICATION Command:

The advantage of the REPLACE option is that it allows you to change a publication without necessarily forcing replicas to re-register and get a full refresh. In some cases, replicas may continue getting incremental refreshes.

If you do not use the REPLACE option, then any time you want to change a publication you must drop that publication and re-create it. When a publication is dropped and re-created, replicas must re-register for that publication and must get a full refresh.

When you use the REPLACE option, however, you can modify a publication that already exists. You can change the definition of an existing publication without dropping and re-creating the publication. Since you

didn't drop and re-create the publication, replicas can continue getting incremental refreshes instead of being forced to get a full refresh.

Note that the REPLACE option does not guarantee that replicas won't be required to get a full refresh when a publication is changed. Some changes to a publication may be significant enough that the replicas will have to get a full refresh. Also, not all operations can be performed with a REPLACE statement. For example, the REPLACE does not allow you to change the publication argument list. If you do need to change the publication argument list, then you will have to drop and re-create the publication, and of course this means that replicas will have to re-register and get a full refresh.

Note: Both the REPLACE option and the SYNC MODE MAINTENANCE setting allow you to make changes that don't necessarily require replicas to get a full refresh. The difference is that SYNC MODE MAINTENANCE allows you to make changes to the schema of a table, while the REPLACE option allows you to make changes to a publication.

Table-Level Locking

Table-level locking allows you to lock (and unlock) an entire table explicitly, e.g. for the duration of critical schema upgrade operations. This allows you to ensure that the schema upgrade operations are not interfered with by other operations on the affected tables.

You may lock a table in EXCLUSIVE or SHARED mode. If you plan to change the schema of a table, you will probably want to lock the table in EXCLUSIVE mode. Note that, as with any other exclusive lock you cannot acquire an exclusive table lock if any other user has locked the table. In a busy system, it may be very difficult to get an exclusive lock on a table. You may have to ask other users to stop using the system (or at least stop using that table) temporarily so that you can get the lock. Once you have the exclusive lock, of course, it will prevent any other users from using that table until you unlock it.

In most situations, locks are released at the end of a transaction. The LOCK TABLE command, however, gives you the option of holding a lock past the end of a transaction. If you do hold a lock past the end of a transaction, then you must explicitly UNLOCK the table to release that lock. Otherwise, the lock will persist until the client application who has obtained the lock, disconnects.

Note that although table locking is used primarily to make it easier and safer to upgrade schemas, you may use it for other purposes as well. You may use table locks at any time, not just when the Sync Mode is set to Maintenance.

For more details, including the exact syntax to use to get an EXCLUSIVE lock or to get a lock that lasts past the end of the current transaction, see the descriptions of the SQL commands LOCK TABLE and UNLOCK TABLE in the *solidDB SQL Guide*.

Version Checking with SYNC_APP_SCHEMA_VERSION

When you change the schema on a master and a replica, you usually cannot do these changes simultaneously. Typically, if you've changed the schema on the master and not the replicas, you do not want the replicas to refresh until they, too, have updated their schema. You may use the bulletin board parameter titled *SYNC_APP_SCHEMA_VERSION* to help prevent synchronization when the master and replica do not have the same schema.

If either the master or the replica has this parameter set, then when the master and replica try to synchronize they will compare values of this parameter. If the value on the master does not match the value on the replica, then the two will refuse to synchronize. Instead, the replica will merely store the synchronization message. You may re-send that message later by using the MESSAGE FORWARD command after you've updated the replica's schema.

For example, if the master sets its SYNC_APP_SCHEMA_VERSION to "Version2" while the replica has its value set to "Version1", then the two will refuse to synchronize.

Note that the master and replica servers merely compare their values for this parameter; they do not actually compare their schemas. If you accidentally set the master's and replica's SYNC_APP_SCHEMA_VERSION to the same value when the servers do not actually have the same schema, then the master and replica will try to synchronize.

The servers only use this bulletin board parameter if it is set. This bulletin board parameter is optional, and is not set automatically. There is no default value, and the servers do not automatically "increment" the value each time that their schemas are updated. The actual values are meaningless; the only thing that matters is whether or not the master and replica have the same value or a different value. You may use any value you want, e.g. "Version1", "VersionA", "XYZ" or any other string (up to the maximum legal length for bulletin board parameter values, of course).

Using These Features to Update a Distributed Schema

The combination of the first 3 features (SYNC MODE MAINTENANCE, the REPLACE option in CREATE OR REPLACE PUBLICATION, and Table Locking) allows you to update the structure of a table in a publication without requiring a full refresh to each replica that uses that publication.

One possible process for updating the tables in the publication is outlined below. First the master is upgraded, and then the replica(s) are upgraded.

1. Set the catalog's sync mode to Maintenance so that the synchronization history feature is temporarily turned off.

2. Lock a table or set of tables in the publication.

3. Make whatever table or schema changes are required. For example, modify a table, add a new table, or modify a publication.

4. Update the value of the bulletin board parameter SYNC_APP_SCHEMA_VERSION.

5. Upgrade application programs (if necessary).

6. Unlock the table(s).

7. Change the catalog's sync mode from Maintenance back to Normal.

After the master has been upgraded, a nearly identical process is used to update the replica.

Below, we describe the typical sequence of steps used to create and update the schema of a distributed system. First we will describe how to create the schema (without using the Maintenance Mode features), and after that we will describe how to upgrade the schema (with use of the Maintenance Mode features).

7.2.3 Example of Upgrading a Distributed Schema

Creating the Initial Schema

When the distributed system is first created, the administrator of the overall system defines for each database type (master, replicas) a set of scripts (which may be stored procedures) that create the schema of the database catalog. These scripts are responsible for creating all database objects of the schema, including procedures, triggers, events, publications, etc. Many of the scripts may be the same for the replica as for the master. As an example, some or all CREATE TABLE commands may be the same on the replica and master. Other scripts will be different for master and replicas. For example, the scripts that create publications are run only on the master, while the scripts that register a replica are run only on the replicas.

Each synchronizable catalog may have a schema version name as one of its properties. If master and replica catalogs have different schema version names, sending the synchronization message from replica to master fails. The name of the property is SYNC_APP_SCHEMA_VERSION. This version name can be set using the SET SYNC PARAMETER statement.

Creating the Initial Master Schema

The script to create the initial master database is below:

```
CREATE TABLE MYTABLE (
   ID INTEGER NOT NULL PRIMARY KEY,
   STATUS INTEGER NOT NULL,
   TEXTDATA VARCHAR NOT NULL);
```

```
ALTER TABLE MYTABLE SET SYNCHISTORY ;
COMMIT WORK ;

"CREATE PUBLICATION
MYPUBLICATION
BEGIN
   RESULT SET FOR MYTABLE
   BEGIN
      SELECT * FROM MYTABLE ;
   END
END";
COMMIT WORK ;

SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION 'VER1';
COMMIT WORK ;
```

Creating a Replica Schema

When each replica is first created, the replica's administrator connects to the replica server, sets the current catalog, and executes the scripts. As part of the schema creation process, the replica is registered with the master.

After this process, both the master and the replica catalogs have the same version name, which means that their schemas are compatible with each other. You should set the synchronization parameter *SYNC_APP_SCHEMA_VERSION* to the same value in both the master and replica(s) so that they can recognize that their schemas are compatible.

In the replica, the initial schema looks as follows:

```
CREATE TABLE MYTABLE (
   ID INTEGER NOT NULL,
   STATUS INTEGER NOT NULL,
   TEXTDATA VARCHAR NOT NULL,
   PRIMARY KEY (ID, STATUS));

ALTER TABLE MYTABLE SET SYNCHISTORY ;
COMMIT WORK ;

CALL SYNC_REGISTER_PUBLICATION (NULL, 'MYPUBLICATION');
COMMIT WORK ;
```

```
SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION 'VER1';
COMMIT WORK ;
```

In the above script, the table MYTABLE is set to support incremental publications by setting the SYNCHISTORY on. Additionally, the replica database registers to the MYPUBLICATION publication by calling the SYNC_REGISTER_PUBLICATION system procedure. The version name for both the master and the replica database catalogs is set to be 'VER1'.

Specifying and Distributing a Schema Upgrade

The administrator of the overall system creates for each database type (master and replicas) a set of scripts that change the schema from the current version to the upgraded version. Upon completion of the scripts, the version name is upgraded to a new one.

The new scripts can be distributed to the replicas using any data distribution mechanism, including synchronization across a SmartFlow database hierarchy.

When defining a schema upgrade, the following rules apply:

- Any new database objects may be added to the schema. Any database object may also be dropped from the schema. If a table is dropped from the schema, it must be removed from publication definitions first.
- The call interface (i.e. parameter list) of stored procedures should not be changed. If such a change is needed, the new procedure should have a different name. The old procedure should remain in the system for a while to guarantee successful execution of transactions that have been saved in replicas but that have not yet been propagated to the master.
- Publications can be changed by adding/removing result sets or adding/removing columns of a result set. Changing search criteria of a result set is also possible; however, it may force the next refresh of that result set to be a full refresh rather than an incremental refresh.
- The parameter list of publications must not change. If you must change the parameter list, then you must drop the old publication and create a new one. The refreshes from the new publication will be full refreshes rather than incremental refreshes.

If you write scripts to make changes to the schema, most of the commands will be the same for the replica as for the master. For example, if you add a new column to a table on the master, then you may want to add that new column to the corresponding tables on each replica. However, there are some commands that will not be the same on the master and replica. For example, the commands to change the publications do not apply to the replicas. Similarly, if you make changes that do require to completely drop and re-create a publication, and thus that require a replica to re-register with the master, the re-register commands will be executed only by the replica(s), not by the master. As you write your scripts, you may want to organize them in such a way

that you make maximum re-use of the common elements (such as ALTER TABLE statements) without running other statements (such as statements to create publications or register replicas) on the wrong servers.

Below, we show a simple example of upgrading a schema of a distributed database system using the schema upgrade capabilities of solidDB.

Typically the first database schema to be upgraded is that of the master database. After upgrading the master, the databases of the replicas are upgraded. In a system that has more than 2 tiers (and thus where intermediate-level nodes are both masters and replicas), the process is to start at the topmost tier and upgrade it. Upon completion of the scripts in a database, the version name of the affected database catalog is upgraded to a new one. This indicates to the next tier of databases that they need to be upgraded as well.

Upgrading the Distributed Schema

Upgrading the Master Schema

The master schema should be upgraded prior to upgrading the replica schemas. To do this, the administrator of the master database server executes the upgrade scripts. During the upgrade, the administrator usually should deny concurrent write access to the tables in the publication (by using the LOCK TABLE statement) and all synchronization access to the catalog (by using MAINTENANCE MODE).

The following script shows how to add a new table to the schema and include it in the existing publication.

```
-- Set the sync mode to Maintenance to allow changes to tables that are
-- referenced by publications. Setting the sync mode to Maintenance also
-- blocks synchronization access to the master database.
SET SYNC MODE MAINTENANCE ;
COMMIT WORK ;

-- Alter the existing table by adding a new column to it
LOCK TABLE MYTABLE IN LONG EXCLUSIVE MODE ;
COMMIT WORK ;
ALTER TABLE MYTABLE ADD COLUMN NEWCOL INTEGER ;
COMMIT WORK ;

-- Set a default value to the new column.
-- While the sync mode is set to Maintenance, updates are not sent to
-- replicas. Therefore, if any updates were done on the master, the same
-- updates must be done locally on each replica while its sync mode is
-- set to Maintenance.
UPDATE MYTABLE SET NEWCOL = 1 ;
```

```
COMMIT WORK ;

-- Release the lock in the MYTABLE table.
UNLOCK TABLE MYTABLE ;
COMMIT WORK ;

-- Create a new table in the schema.
CREATE TABLE MYSECONDTABLE (
   ID INTEGER NOT NULL,
   MYTABLEID INTEGER NOT NULL,
   STATUS INTEGER NOT NULL,
   TEXTDATA VARCHAR NOT NULL,
   UPDATETIME TIMESTAMP NOT NULL,
   PRIMARY KEY (ID, MYTABLEID, STATUS)) ;
ALTER TABLE MYSECONDTABLE SET SYNCHISTORY ;
COMMIT WORK ;

-- Create a new version of the publication.
"CREATE OR REPLACE PUBLICATION MYPUBLICATION
BEGIN
RESULT SET FOR MYTABLE
BEGIN
   SELECT * FROM MYTABLE ;
   RESULT SET FOR MYSECONDTABLE
   BEGIN
      SELECT * FROM MYSECONDTABLE
      WHERE MYTABLEID = MYTABLE.ID ;
   END
END
END";
COMMIT WORK ;

-- Change the version information of the master catalog.
SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION 'VER2';
COMMIT WORK ;
-- Set the sync mode back from MAINTENANCE to NORMAL.
SET SYNC MODE NORMAL ;
COMMIT WORK ;
```

After successful execution of the script in the master database, the schema of the master database has been upgraded and the version name of the catalog has been changed to 'VER2'. The database is also opened again

for synchronization access by setting the sync mode to NORMAL. However, the replica databases cannot synchronize with the master before they upgrade their schema to the same level.

Detecting the Need for Upgrading Replica Schema

If the master and replica have each defined their version by setting the bulletin board parameter *SYNC_APP_SCHEMA_VERSION*, and if the master and replica versions don't match, then an error is returned when the replica attempts to synchronize with the master next time. Typically, the data is synchronized by executing the following kind of SQL script in the replica database.

```
MESSAGE syncmsg BEGIN ;
MESSAGE syncmsg APPEND PROPAGATE TRANSACTIONS ;
MESSAGE syncmsg APPEND REFRESH MYPUBLICATION ;
MESSAGE syncmsg END ;
COMMIT WORK ;
MESSAGE syncmsg FORWARD TIMEOUT 10 ;
COMMIT WORK ;
```

If the version name of the master database does not match the version name of the replica database, then the statement

```
MESSAGE <msgname> FORWARD
```

will fail with error:

25092 - User version strings are not equal in master and replica, operation failed.

Although sending the message to the master failed, the message stays persistent in the replica database. After the replica schema is upgraded to match the master's schema, the failed message can be re-sent to the master by using the statement

```
MESSAGE <msgname> FORWARD;
```

After the need for replica schema upgrade has been detected, the administrator of the replica server needs to upgrade the schema to the new version using version upgrade scripts developed for that replica database.

Upgrading the Replica Schema

In a typical upgrade process, the administrator of the master database server writes a set of scripts to modify the schema, and then sends the appropriate scripts to the administrator of the replica database server.

After a successful upgrade, the schema version name is changed to the new one. The possibly hanging synchronization message(s) can now be re-sent to the master.

In our example, the script for upgrading the replica to match with the new version of the master schema looks like the following:

```
-- Set the sync mode to Maintenance to allow changes to tables that
-- are referenced by publications.
-- The synchronization functions of the replica database are suspended.
SET SYNC MODE MAINTENANCE ;
COMMIT WORK ;

-- Alter the existing table by adding a new column to it.
-- Updates done in maintenance mode are not rolled back in the next
-- synchronization.
-- Corresponding updates have already been done in the master DB.
LOCK TABLE MYTABLE IN LONG EXCLUSIVE MODE ;
ALTER TABLE MYTABLE ADD COLUMN NEWCOL INTEGER
;
COMMIT WORK ;
UPDATE MYTABLE SET NEWCOL = 1 ;
COMMIT WORK ;

-- Release the lock on the MYTABLE table.
UNLOCK TABLE MYTABLE ;
COMMIT WORK ;

-- Create a new table in the replica schema.
CREATE TABLE MYSECONDTABLE (
   ID INTEGER NOT NULL,
   MYTABLEID INTEGER NOT NULL,
   STATUS INTEGER NOT NULL,
   TEXTDATA VARCHAR NOT NULL,
   UPDATETIME TIMESTAMP NOT NULL,
   PRIMARY KEY (ID, MYTABLEID, STATUS)) ;
ALTER TABLE MYSECONDTABLE SET SYNCHISTORY ;
COMMIT WORK ;

-- Note that changes in the publication definition don't require any
-- actions in the replica end. The changes in the publication's meta
-- data as well as data of the added tables are automatically sent
```

```
-- to the replicas.

-- Change the version information of the replica database catalog.
SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION 'VER2';
COMMIT WORK ;
-- Set the sync mode back from MAINTENANCE to NORMAL.
SET SYNC MODE NORMAL ;
COMMIT WORK ;
```

After the script has been executed successfully in the replica database, the possibly stopped synchronization messages can be re-sent to the master database by executing the statement

```
MESSAGE <msgname> FORWARD
```

for each stopped message. In this example, re-execute the following statements:

```
MESSAGE syncmsg FORWARD TIMEOUT 10 ;
COMMIT WORK ;
```

7.2.4 Cautions for Maintenance Mode

Maintenance Mode is a powerful feature; use it carefully.

When you SET SYNC MODE MAINTENANCE in a database, you tell the server not to update synchistory info. Since the server is not recording changes, the next incremental refresh will not necessarily copy all changes from the master to the replica. It is assumed that the changes that were made in the Maintenance Mode in the master database during schema version upgrade process, are done also in the replica databases (similarly, in Maintenance Mode) in their corresponding schema upgrade processes before the next synchronization takes place.

This is an important difference from the way that a solidDB synchronized system normally behaves. One of the major advantages of Solid's proprietary synchronization technology is that it allows a system to be "self-healing" in some situations. Incorrect data on a replica tends to be replaced with data from the master, and therefore errors on replicas tend to disappear over time.

In Maintenance Mode, however, you lose this self-healing property. Since the master and replica do not store sync history data, they do not know what changes were made, and the master does not send all the updates to the replica the next time that the replica refreshes.

There are two possible sources of error:

1. The first possible error is that you might accidentally make a different change on the master than on the replica. For example, when you add a new column to a table, you might set the default value to 1 on the master, but accidentally set it to 2 on the replica.

2. The second possible error is that even if you perform the same operations without error on both the master and the replica, certain types of errors may occur (and not be automatically repaired) if the replica and master do not have the same starting values of data. (Imagine that the master and replica will each calculate sales tax on an invoice. If the master and replica have different values for the total price on the invoice, then they will calculate different sales tax, even if they use the same formula.) This situation is easy to get into because replicas and masters are updated independently (asynchronously) during Maintenance Mode operations. The replica is not necessarily "in sync" with the master at the time that the replica start its updates.

Therefore, when you are in Maintenance Mode, it is not safe to perform operations that rely on the replica and master being exactly in sync. The types of operations that you do in Maintenance Mode should be insensitive to whether the data in the replica is completely up-to-date. For example, if you add a new column, that doesn't affect existing data values. However, if you change the values of existing columns while in Maintenance Mode, those changes might not match. If you perform DML operations while in Maintenance Mod, you should be very careful.

7.2.5 Upgrading the Server Version

When the solidDB server is upgraded (for example from version 4.20 to version 4.50), the master server must be upgraded prior to upgrading any replica servers. To ensure that data is converted from the previous format to the newer format, you should start the new server and use either the **-x convert** or **-x autoconvert** option on the command line. See *solidDB Administration Guide* for more details about these command-line options. Note that after conversion, you cannot use the database with an older server version.

If you are using the Solid CarrierGrade option (formerly called HotStandby), then you must first set the Primary server to PRIMARY ALONE state to allow upgrade of the Secondary server. After the Primary server has been switched to PRIMARY ALONE state, the Secondary server can be shut down and upgraded. After the Secondary is upgraded, it is re-started. After successful catchup, the original Primary is shut down and the Secondary server is immediately switched to PRIMARY ALONE state. The original Primary server can be upgraded while former Secondary runs in PRIMARY ALONE state. Finally, the old Primary is started as a Secondary and runs in catchup mode using the transaction log of the new Primary. (See *solidDB High Availability User Guide* for more details.)

When you use AcceleratorLib, the application typically changes when the schema changes. This means that a new build of the application and the AcceleratorLib library is needed as part of the schema upgrade process.

Chapter 8. Administering solidDB with SmartFlow

This chapter describes how to maintain your solidDB with SmartFlow technology. The administration tasks covered in this chapter include managing synchronization errors, and tips on backing up masters and replicas.

Important

In the solidDB with AcceleratorLib, there are some differences in administration from standard solidDB. This chapter assumes that if you will be using AcceleratorLib with SmartFlow, then you have already read *solidDB AcceleratorLib User Guide*.

8.1 What You Should Know

This section describes what you need to know about solidDB before you begin SmartFlow administration and maintenance.

- Installing solidDB

  If you have not yet installed solidDB, refer to the release notes file delivered with the software or included on the Solid Web site at:

  http://www.solidDB.com

  The release notes file contains a detailed description of the installation.

- Special roles for database administration

  To perform some synchronization-related operations, you may need to have been granted the role named SYS_SYNC_ADMIN_ROLE. For more information, see the *solidDB Administration Guide*.

8.2 Monitoring solidDB SmartFlow

The following sections describe the methods used for querying the status of the data synchronization of a solidDB database.

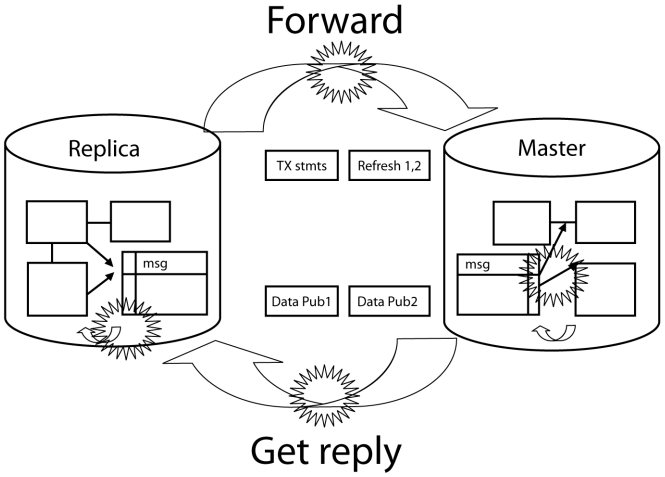
8.2.1 Monitoring the Status of Synchronization Messages

Because synchronization is implemented using synchronization messages, you can monitor the status of the synchronization process by checking the status of the currently existing messages.

When a message is active, it is always persistent in some state in the system. The message is deleted from the databases once it has been successfully processed. A message that is hanging in either the master database or a replica database, is never completely processed. In most cases, an idle, persistent message means that a synchronization error has occurred.

Figure 8.1, “Error-prone areas in synchronization messaging” illustrates store and forward messaging and shows the points in the process where messaging errors can occur.

Figure 8.1. Error-prone areas in synchronization messaging



As Figure 8.1, “Error-prone areas in synchronization messaging” shows, there are four error prone areas:

1. When a message is being forwarded from replica to master
2. When the message is executed in the master
3. When the reply message is being received from the master
4. When the reply message is being executed in the replica

In all these cases, the failure stops the synchronization. A query of the following message information system tables will provide the reason for the failure:

- SYS_SYNC_REPLICA_MSGINFO in replica databases
- SYS_SYNC_MASTER_MSGINFO in the master database

For a detailed description of these tables, see the appropriate appendix in the *solidDB SQL Guide*. Refer to the following section for ways to resolve errors that you see in the tables.

8.2.2 Managing Synchronization Errors

A synchronization messaging error occurs when a message delivery or receipt fails. This section describes the procedures to manage synchronization errors. Depending on where the synchronization error occurred, the way to recover from errors can vary.

Error in Forwarding a Message to the Master

When sending a message from the replica to the master fails, the message remains in the replica database and it can be re-sent to the master. The value of the STATE column of row in SYS_SYNC_REPLICA_MSGINFO table is in this case 22 - R_SAVED. You can query names of those messages that have not been successfully sent to the master with the following SQL statement:

```
SELECT MSG_NAME
FROM SYS_SYNC_REPLICA_MSGINFO
WHERE STATE = 22;
```

Failed messages can be re-sent to the master database with the following command in the replica database:

```
MESSAGE message_name FORWARD;
```

The possible values of the STATE column of the SYS_SYNC_REPLICA_MSGINFO table are documented in the Appendix D - System Tables of *solidDB SQL Guide*.

Error in Execution of a Synchronization Message in the Master

A message execution can fail in the master database, if:

- a SQL statement of a transaction fails
- refreshing data from a publication fails

- sending a reply message back to the replica fails

The method used to handle each of these reasons for a failed message execution is covered in this section.

Error Handling in Solid Intelligent Transaction

If an intelligent transaction fails because of a fatal error, then the execution of the message is stopped in the master database and the transaction is rolled back. The error code of the failed operation is returned to the replica as the error code of the synchronization messaging command that was supposed to return the reply message to the replica database.

The error code is returned to the replica as a return code from either of these statements:

```
MESSAGE message_name FORWARD TIMEOUT timeout_in_seconds
```

or

```
MESSAGE message_name GET REPLY TIMEOUT timeout_in_seconds
```

In the master database, the system table SYS_SYNC_MASTER_MSGINFO contains infor-mation about all messages that currently reside in the master database. If the execution of a message has failed because of an error, that message will have value 1 in the STATE column. The ERROR_CODE and ERROR_TEXT columns contain information about the error that caused the message execution to halt. You can query these hanging messages, their originating replica database as well as the statement that caused the message to halt from the master database, by querying the view named SYNC_FAILED_MESSAGES. This view exists on the master and shows the replica name, message name, statement string, error information, and other information.

The proper way to recover from a hanging message in the master is to fix the error in the master database. For example, the reason for the error could be a unique constraint violation. To fix this error, the existing data of the master database must be modified to allow the new row to be inserted. (It is not possible to modify the contents of the transaction that resides in the transaction queue.) Alternatively, there could be a programming error in a stored procedure that needs to be corrected by recreating the stored procedure in the master database.

Once the error is corrected, restart the message in the master database with the following command:

```
MESSAGE message_name FROM REPLICA replica_name EXECUTE
```

After the message is successfully executed in the master database, the reply message can be requested to the replica database with the following command:

```
MESSAGE message_name GET REPLY TIMEOUT timeout_in_seconds
```

Alternatively, the entire halted message can be deleted from the master database with the following command:

```
MESSAGE message_name [FROM REPLICA replica_name] DELETE
```

or, just the current transaction in the message can be deleted from the master database with the following command:

```
MESSAGE message_name FROM REPLICA replica_name DELETE CURRENT TRANSACTION
```

However, using this alternative will cause loss of data and should be used only as a last resort when there is no other means for resolving the error.

Note that MESSAGE DELETE CURRENT TRANSACTION is a transactional operation and must be committed before message execution may continue. To restart the message (where it left off) after the deletion is committed, use the following statement:

```
MESSAGE msgname FROM REPLICA replicaname EXECUTE
```

In general, transactions should be written to avoid concurrency conflicts and deadlocks. Because conflicts can still occur when transactions update or delete rows, we recommend you specify the *SYS_TRAN_MAXRETRY* bulletin board parameter in the master database using the SET SYNC PARAMETER command. The *SYS_TRAN_MAXRETRY* parameter retries a transaction that has failed due to a concurrency conflict or a deadlock based on a user configurable maximum number of attempts. For details, read the description of the *SYS_TRAN_MAXRETRY* in Appendix A, *Bulletin Board Parameters*.

Error Handling During Refresh Operations

Unlike transactions, an error while executing a REFRESH request on the master does not cause the entire message to stop. Instead, the error is reported back to the replica database in the result set of the messaging command. The error code is returned in the ERRCODE column of the result set. Similarly, the error text can be found from the ERRSTR column of the result set.

The result set of the messaging commands should always be fetched and the ERRCODE checked. All non-zero values mean that an error has occurred during the message execution in the master database.

One possible source of error is that the version of the publication has changed in the master database. If the publication was dropped and then re-created, the subscription(s) to the old version of the publication must be

dropped in the replica database prior to subscribing to the new version. Dropping a subscription is done with the following command:

```
DROP SUBSCRIPTION publication_name [{(parameter_list) | ALL}]
   [COMMITBLOCK number_of_rows] [OPTIMISTIC | PESSIMISTIC];
```

When a subscription is dropped, all data for that subscription is deleted from the replica database. Subscribing to the new version and then requesting a refresh always brings the full publication to the replica.

In many cases, you can avoid this problem by using the "OR REPLACE" option of the CREATE PUBLICATION command when you need to update a publication. If you update a publication by using CREATE OR REPLACE PUBLICATION, rather than by dropping and re-creating the publication, then you will not necessarily need to drop and re-create the subscriptions. If you do not need to re-create the subscriptions, then in some cases the master can send the replicas an incremental refresh rather than a full refresh. Incremental refreshes reduce network traffic.

Error in Receiving a Reply Message to a Replica

A transfer of a reply message from master to replica can fail because of a networking error. In this case, the message remains in the master database.

You can use the following SQL statement to list messages whose replies have not been successfully received by the replica:

```
SELECT MSG_NAME
FROM SYS_SYNC_REPLICA_MSGINFO
WHERE STATE = 23
```

You can request the message again from the master database using the following command:

```
MESSAGE message_name GET REPLY TIMEOUT timeout_in_seconds
```

The possible values of the STATE column of the SYS_SYNC_REPLICA_MSGINFO table are documented in the Appendix D - System Tables of *solidDB SQL Guide*.

Error in Executing a Reply Message in a Replica

An execution of a reply message in a replica database can fail because of a concurrency con-flict. Records applicable to a transaction may be locked so that an operation can't be performed in the replica. In this case, the message remains in the replica database. For example, when other transactions are updating a table, a

concurrent refresh operation may fail. In this case, the REFRESH command remains in the replica database and must be re-executed.

You can allow the message to be re-executed from the replica database using the following command:

```
MESSAGE message_name EXECUTE
```

For example:

```
MESSAGE MyMsg0002 EXECUTE;
```

Sometimes it may be necessary to execute the message using pessimistic locking. This way, you avoid concurrency conflict handling, especially during REFRESH operations. For more information, read the Section 5.2.5, “Handling Concurrency Conflict in Synchronized Tables in Replica”.

Deleting a Message for Error Recovery

You can also explicitly delete a message from a replica database to recover from an error. When you delete a message, you can specify that the entire contents or only the current transaction that is propagated to the master database in the message be permanently deleted. The command to delete the entire message is:

```
MESSAGE message_name [FROM REPLICA replica_name] DELETE
```

The command to delete the current transaction is:

```
MESSAGE message_name FROM REPLICA replica_name DELETE CURRENT TRANSACTION
```

Note that the above statement can be used only in the master database.

When deleting the message from the master database, be sure to specify the replica name in the clause FROM REPLICA *replica_name*.

For example:

```
Message MyMsg0001 FROM REPLICA bills_laptop DELETE;
```

8.3 Performing Backup and Recovery

This section describes how to back up your databases and recover from system failure.

See also *solidDB Administration Guide*.

8.3.1 Making Backups

Backups are made to secure the information stored in your database files. If you have lost your database files because of a system failure, you can restore your database from the backup files.

When you use the database server's own "backup" command, the following files are backed up:

- The files containing the database itself. These are the files specified in the "FileSpec*" files in the *[IndexFile]* section of the `solid.ini` configuration file.
- The log files that contain information used in "rolling forward" after a failure. These log files are named `sol#####.log`. They are specified in the *[Logging]* section of the `solid.ini` configuration file

  `("FileNameTemplate=<log_file_path>sol#####.log")`.

The backup operation also copies the following files, although they are not required as part of the "restore" process.

- By default, `solmsg.out` is copied along with other files. This is for convenience in diagnosing problems. This file is not required during a "restore".
- The `solid.ini` configuration file is also copied by default because after a disk crash the original `solid.ini` might be destroyed.

Note that the `solid.lic` file is not automatically copied.

You can initiate a backup in the following ways:

- Automate the backup using a timed command that initiates the backup according to a pre-defined schedule. Read the section on Entering Timed Commands in *solidDB Administration Guide*.
- Select the Status option from the SolidConsole Administration window, and click the Backup icon to initiate the backup from the Backup dialog box.

- Issue the following command in SolidConsole or Solid SQL Editor (teletype):

  ADMIN COMMAND 'backup'

Note

Be sure you have enough disk space in the backup directory for your database and log files.

8.3.2 Viewing Solid Messages in the Backup Directory

The system copies the Solid messages file (`solmsg.out`) file to the backup directory (parameter *`BackupCopySolmsgout`* in the *`[General]`* section of `solid.ini` is set to yes by default). This provides a convenient way to view what operations were performed with a solidDB server before performing a backup. This also allows the Solid messages file to exist in the backup directory for viewing before restoring the database from a corresponding backup file.

8.3.3 Backing up and Restoring the Master and Replica Databases

The normal disaster prevention and recovery tasks you use on a non-synchronized solidDB database also apply to synchronized databases. It is advisable that you automate backups to be run at non-busy hours. After completing the backup, copy your backup files on tape using your backup software for protection against disk crashes.

After restarting the database, verify that any possible ongoing synchronization messaging has been completed successfully. For instructions on checking for synchronization errors, see Section 8.2.1, “Monitoring the Status of Synchronization Messages”. For instructions on correcting stopped synchronization messages, see Section 8.2.2, “Managing Synchronization Errors”.

If the database file of a master or replica database is corrupted, then it is necessary to restore the database from a backup file. You can make an on-line backup of any database of a solidDB system. Upon recovery, solidDB uses transaction log files to roll forward the backup database from the state of the backup to the state of the last committed transaction. The last persistent state of the synchronization is also restored at that time.

Note

- You can query programmatically the status of the most recently started backup in SolidConsole or Solid SQL Editor (teletype) by using the command: **ADMIN COMMAND 'status backup'** To query the list of all completed backups and their success status, use the command: **ADMIN COMMAND 'backuplist'**

You can also query backup status in SolidConsole by selecting the Status option in the Administration window or menu and clicking the Backup icon. A backup status listing is displayed in a dialog box.

- The backup directory you enter must be a valid path name in the server operating system. For example, if the server runs on a UNIX operating system, path separators must be slashes, not backslashes.
- The time needed for making a backup is the time that passed between the messages `Backup started` and `Backup completed successfully`, which is written to the `solmsg.out` log files. These messages are also displayed on the SolidConsole Messages page.

Before starting the backup process, a checkpoint is created automatically. This guarantees that the state of a backup database is from the moment the backup process was started. The following files are then copied to the backup directory:

- database file(s)
- configuration file (`solid.ini`)
- log file(s) modified or created after the previous backup (parameter *`BackupCopyLog`* in the *`[General]`* section of `solid.ini` is set to yes by default)
- backup of Solid messages file `solmsg.out` (parameter *`BackupCopySolmsgout`* in the *`[General]`* section of `solid.ini` is set to yes by default).

The unnecessary log files are deleted from the original directory after successful backup (parameter *`BackupDeleteLog`* in the

`General`

section of `solid.ini` is set to yes by default).

8.3.4 Backup Guidelines

Using backups in the SmartFlow system is a straightforward and powerful way to ensure the high level of data availability and security. Therefore, you should regularly back up all critical databases.

To ensure that data is secure in the event of a system failure, always back up the master and possibly also the replica databases on a periodic basis. Note the following guidelines when backing up a SmartFlow system:

- Because the master database has the official version of data and the data in the replicas are tentative, do not use a replica database as a backup for the master database. The last persistent state of a master database can be restored only from the backup files of the master database itself.

- If the master database is recovered from a backup without data loss (in other words, if rollforward recovery is able to recover all the transactions from the log files), then synchronization can proceed normally. If there is data loss (for example, due to missing transaction logs), it is also reflected in the replicas during the next refresh. If synchronization data is not in sync between the master and replicas, a full publication, which means all data in the publication, is sent to the replica database during the next refresh. This occurs regardless of whether the tables in the publication are set for *incremental publication*.

- A replica database can be reconstructed either by restoring a backup of that particular replica database or alternatively, the database can be re-built from scratch by refreshing data from the master database. The former approach is feasible if the database is a large one and there is disk space available for the backup. The latter approach can be used if the replica database is a small one and there is no local-only data in the replica database. For details on restoring backups, see the section on restoring backups in *solidDB Administration Guide*.

- When you restore any backup of a solidDB database, make sure that the restore is rolling forward all transactions to the latest committed transaction. This ensures that the synchronization continues from the point where it was at the time of the database failure.

- Once the restore is complete, make sure there are no uncompleted synchronization messages in either the master or replica database.

  For instructions on checking for synchronization errors, see Section 8.2.1, “Monitoring the Status of Synchronization Messages”. For instructions on correcting stopped synchronization messages, see Section 8.2.2, “Managing Synchronization Errors”.

Chapter 9. Performance Monitoring and Tuning

This chapter discusses techniques that you can use to improve the performance of the SmartFlow feature of solidDB. For information about tuning the performance of other aspects of solidDB, see *solidDB Administration Guide*.

9.1 Monitoring the Progress of Messages

solidDB provides a number of events that let you monitor the processes of propagating and refreshing data between a master and a replica. Two of those messages are specifically for tracking how many bytes of a message have been sent or received so far. These events are useful primarily when sending very large messages, such as messages containing BLOBs, or when sending messages over very slow communication channels. If you are sending a BLOB, for example, you could use this to notify you after every 20K of data has been sent, and you could then update a screen display showing the amount of data that has been downloaded.

The two events are:

```
SYNC_MSGBYTES_SENT(
   sender_nodename              WVARCHAR,
   receiver_nodename            WVARCHAR,
   message_name                 WVARCHAR,
   cumulative_bytes_sent        INTEGER,
   total_bytes                  INTEGER);

SYNC_MSGBYTES_RECEIVED(
   sender_nodename              WVARCHAR,
   receiver_nodename            WVARCHAR,
   message_name                 WVARCHAR,
   cumulative_bytes_received    INTEGER,
   total_bytes                  INTEGER);
```

Both events essentially have the same parameters.

Both messages are posted 0 or more times during the process of forwarding a synchronization message between a master and a replica. SYNC_MSGBYTES_SENT events are posted in the transmitting node, and SYNC_MSGBYTES_RECEIVED events are posted in the receiving node.

The event lists the cumulative number of bytes sent/received so far as well as the total number of bytes to be sent/received inside the corresponding sync message. The user can monitor the progress of the sending/receiving process by catching these events and comparing the cumulative byte count sent so far to the total bytes in the message.

To control how frequently these messages are sent, you set the `solid.ini` configuration parameter

```
[Synchronizer]
RpcEventThresholdBytecount=<value>
```

The value is specified in bytes, and must be greater than or equal to 0. Note that abbreviations, such as K for kilobytes, are not accepted in the value.

If the value is 0, then neither SYNC_MSGBYTES_SENT nor SYNC_MSGBYTES_RECEIVED events will be posted inside the corresponding node. The default value is 0 (i.e. no events are posted).

This parameter specifies the minimum number of bytes that must be sent before the first event is posted and in between each successive event. The server will post the first event after it has sent or received approximately the specified number of bytes, and will post another event each time it has sent or received approximately the specified number of additional bytes. For example, if *`RpcEventThresholdByteCount`* is set to 1000, then the server will post events at approximately the following times:

After 1000 bytes have been sent.

After 2000 bytes have been sent.

After 3000 bytes have been sent.

...

The *`RpcEventThresholdByteCount`* sets the MINIMUM number of bytes that must be transmitted before the first event and between each subsequent event. Events are not necessarily posted after exactly as many bytes as are specified in *`RpcEventThresholdByteCount`*. The server only checks the byte count and considers posting an event after it has completed the sending of each communication packet (not each byte). These packets are not necessarily the same size as *`RpcEventThresholdByteCount`*.

For example, suppose that *`RpcEventThresholdByteCount`* is 1000, and that you are sending 3500 bytes. Suppose also that each packet is 1500 bytes. Instead of getting events after exactly 1000, 2000, and 3000 bytes, you will get an event after the first packet (1500 bytes) and the second packet (3000 bytes).

As another example, suppose again that *`RpcEventThresholdByteCount`* is 1000, and that you are sending 3500 bytes. However, suppose this time that each packet is 600 bytes. Instead of getting events after

exactly 1000, 2000, and 3000 bytes, you will get an event after the second packet (1200 bytes) and the 4th packet (2400 bytes) and the fifth packet (3000 bytes).

Note that the server does not post SYNC_MSGBYTES_SENT or SYNC_MSGBYTES_RECEIVED after the final packet has been sent (or received). Instead, after each complete message has been sent or received, the server will post another event, such as SYNC_MASTER_MESSAGE_RECEIVE_END.

If you plan to use SYNC_MSGBYTES_SENT and SYNC_MSGBYTES_RECEIVED, then we recommend that you also catch the starting and finishing events of the corresponding message. When you get an event that tells you that the specific synchronization message has been completely received, then you may want to unregister for that event, or at least stop executing a WAIT statement to wait for the next such event. Similarly, when you get an event saying that a new message is being forwarded (SYNC_MASTER_MESSAGE_RECEIVE_BEGIN) then you may want to start monitoring SYNC_MSGBYTES_RECEIVED.

Below is an outline of some of the message-related events that will occur.

1. Forward message from replica: monitoring of sent bytes.

   SYNC_REPLICA_MESSAGE_FORWARD_BEGIN

   ...

   0 or more occurrences of SYNC_MSGBYTES_SENT

   ...

   SYNC_REPLICA_MESSAGE_FORWARD_END

2. Master receives the forwarded message and monitors the number of bytes received.

   SYNC_MASTER_MESSAGE_RECEIVE_BEGIN

   ...

   0 or more occurrences of SYNC_MSGBYTES_RECEIVED

   ...

   SYNC_MASTER_MESSAGE_RECEIVE_END

3. Master sends a reply message to the replica.

   SYNC_MASTER_MESSAGE_SENDREPLY_BEGIN

   ...

0 or more occurrences of SYNC_MSGBYTES_SENT

...

SYNC_MASTER_MESSAGE_SENDREPLY_END

4. Replica receives the reply message from the master.

   SYNC_REPLICA_MESSAGE_REPLY_BEGIN

   ...

   0 or more occurrences of SYNC_MSGBYTES_RECEIVED

   ...

   SYNC_REPLICA_MESSAGE_REPLY_END

Note

- There is only a single variable to control both the send and receive intervals. Within a node, the interval (byte count) for send and receive will be the same. Furthermore, "receive" events and "send" events will either both be on or both be off at the same time. You cannot turn on only one of them.
- Since the sender and receiver are normally different nodes, they may have different values for `RpcEventThresholdByteCount`. In fact, one server can have such events turned off (`RpcEventThresholdByteCount=0`) while the other server has the events turned on.
- For more information on waiting for synchronization events, see *solidDB SQL Guide*, especially the sections on the ADMIN EVENT and CREATE PROCEDURE commands.

9.2 Tuning for Data Synchronization

To be sure solidDB is well-tuned for data synchronization, review the following solidDB guidelines for these areas of synchronization:

- Tuning publication definitions
- Optimizing synchronization history data management
- Optimizing synchronization messages

Each of these topics is described in the following sections.

9.2.1 Tuning Publication Definitions

solidDB SmartFlow uses publications for downloading incremental data from the master to replica databases. In a publication, you define the data for propagation from the master to the replica. Using the CREATE PUBLICATION statement, you specify the tables and the search criteria for selecting data to the replica. For details on publications, read Section 6.5, “Creating Publications”.

Below is a simple publication definition example:

```
CREATE PUBLICATION configuration_of_device (device_name VARCHAR)
BEGIN
   RESULT SET FOR device
   BEGIN
      SELECT * FROM device WHERE name = :device_name;
      RESULT SET FOR device_cfg_parameter
      BEGIN
         SELECT * FROM device_cfg_parameter WHERE device_id = device.id;
      END
   END
END
```

Internally, the queries of the RESULT SET FOR paragraphs are executed as regular SELECTs. Nested result sets always produce a join between an outer and inner result set. Therefore, in optimizing performance, the same indexing rules apply here as with other queries, which means you should:

- create an index on columns of large tables that are used as search criteria
- create an index on columns that are used for joins in the nested result set

In addition, we recommend that you avoid nesting result sets in publication definitions. See the following sections of *solidDB SmartFlow Data Replication Guide* for examples of nested vs. unnested result sets:

- "Nested Publication version"
- "Unnested Publication version"

You can extract the SQL that is actually generated from the publication definitions by setting SQL trace on in the master database using the following command:

ADMIN COMMAND 'trace on sql'

The output of the trace goes to the standard trace file of solidDB. The default name of the file is `soltrace.out`.

9.2.2 Optimizing Synchronization History Data Management

The first time that a replica refreshes from a publication in the master, the replica must download a copy of all the information in the publication; i.e. the replica must get a "full refresh". After that first download, each time the replica refreshes, the replica only needs to download the records that have changed since the previous download, i.e. an "incremental refresh".

When synchronizing changed data from the master to a replica, solidDB (in both the master and replica) must know the data written to these databases since the previous synchronization. If the data updates occurred, then solidDB must have a record of the previous version of the row before the update. These old versions of updated rows are recorded to a synchronization history table.

The use of incremental publications is highly recommended for optimum performance. For details on setting up incremental publications, read Section 6.5.1, “Creating Incremental Publications”.

Tuning Synchronized History Tables

In a system where update operations are frequent (relative to how often the replicas request refreshes), the history tables can grow large. By default, solidDB creates a new row in the history table whenever a row of the main table is updated in the master database. This is not always necessary, however. You can reduce the amount of data stored in the history table by specifying which columns of a synchronized table need to cause a new entry to the history table during data updates. Only those columns in the publication that are used as search criteria (WHERE clause or join columns) need to be specified as history columns. (See Table 9.1, “Inside Information: SET HISTORY COLUMNS” for technical details.) To specify these columns, use the following command:

```
ALTER TABLE tablename SET HISTORY COLUMNS (col1, col2, colN...)
```

If your publication definition contains all rows of a table, then specify the primary key column(s) as the HISTORY COLUMNS of that table.

Without this definition, all update operations in the master database cause a new entry to the history table when the corresponding synchronized table is updated. If you have rows that are frequently updated, setting history columns can significantly reduce overhead in terms of performance and disk space consumption in the master database.

Note

In order for ALTER TABLE ... SET HISTORY COLUMNS to succeed, the statement ALTER TABLE ... SET SYNCHISTORY has to be executed first. Executing ALTER TABLE ... SET NOSYNCHISTORY removes also the effect of ALTER TABLE ... SET HISTORY COLUMNS.

Example

Assume you have set the following table, which has already been set for synchronization history:

```
CREATE TABLE account
   (accountid VARCHAR NOT NULL PRIMARY KEY,
   balance numeric(12,2));
```

You can now use the following command to specify that the history entry occurs only if the *accountid* column value is changed by an update operation.

```
ALTER TABLE account SET HISTORY COLUMNS (accountid);
```

Now any changes to the balance column value do not cause a history table row update.

Table 9.1. Inside Information: SET HISTORY COLUMNS

Inside Information: SET HISTORY COLUMNS
You might wonder why the history table only needs to track changes in certain columns. After all, each replica needs to be notified of ANY changes to the data, not just changes to certain columns. A slightly simplified example and explanation are below. Suppose that your replica database contains information only for customers of the London branch office. Your replica must be notified of each of the following types of changes: 1) Changes to information about customers of the London office (e.g. changes to customer phone numbers). 2) Addition of new customers to (or deletion of old customers from) the London office. For example, if you open up a Liverpool office and assign the Liverpool office some customers who were formerly assigned to the London office, then those customers should no longer be received by the London office when it requests refreshes of updated data. In fact, the master must explicitly notify the London replica that it should delete its copy of information about those customers.

Inside Information: SET HISTORY COLUMNS

In other words, there are changes to records *within* a set (the London set), and there are changes that cause records to *move from one set to another set* (e.g. from the London set to the Liverpool set). Each of these 2 possible types of changes is tracked using a separate mechanism.

Changes within a set are tracked using the server's general versioning system. For example, if the London office last refreshed data at 12:01 AM December 3, 2001, then the London office will want updates for any records that are in London's set and that have changed since 12:01 AM December 3, 2001.

Changes that affect *which* set a record appears in (London, Liverpool, etc.) are tracked in the history table. Since the history table only needs to track addition of or removal of customers from a particular set, it only needs to track changes to the fields that determine which set the record is in. Those fields are the "search criteria" — i.e. the WHERE clause and the join fields that you specified when you defined the publication. Since the only way that a customer record can suddenly appear in, or disappear from, the London branch's publication is if there is a change to one of the columns used in the WHERE clause (or the join clause), the history table only needs to record changes to values in those columns.

An example is below:

```
CREATE TABLE customer (
   id VARCHAR NOT NULL PRIMARY KEY,
   name VARCHAR NOT NULL,
   salesman_id VARCHAR NOT NULL) ;

CREATE TABLE invoice (
   customer_id VARCHAR NOT NULL,
   invoice_number VARCHAR NOT NULL,
   invoce_date DATE NOT NULL,
   invoice_total NUMERIC (12,2),
   PRIMARY KEY (customer_id, invoice_number));

CREATE PUBLICATION customers_by_salesman (salesman_id VARCHAR)
BEGIN
   RESULT SET FOR customer
   BEGIN
      SELECT * FROM customer WHERE salesman_id = :salesman_id ;
      RESULT SET FOR invoice
      BEGIN
        SELECT * FROM invoice WHERE customer_id = customer.id ;
      END
```

Inside Information: SET HISTORY COLUMNS

```
   END
END
```

To optimize this for update performance, the following ALTER TABLEs are needed:

```
ALTER TABLE customer SET HISTORY COLUMNS (salesman_id)
ALTER TABLE invoice SET HISTORY COLUMNS (customer_id)
```

Caution

If you execute the command ALTER TABLE tablename SET HISTORY COLUMNS, but you make a mistake and do not specify all the appropriate columns, then the refresh mechanism will not work properly, and records may not get stored in the proper replica(s). Each time you add a new publication, you must consider whether you need to execute the ...SET HISTORY COLUMNS command again to take into account additional columns used in the search criteria of the new publications.

The ALTER TABLE ... SET SYNC HISTORY command is never required. If the HISTORY COLUMN property is not set for a table, the synchronization will work properly but may not be optimized for performance.

Discarding History Data

After synchronization, a background process purges the obsolete data (i.e. data that is older than the most recent synchronization) from the history tables. You do not need to manually delete old history data to prevent it from accumulating.

9.2.3 Read-Only Replica

If the nature of the Replica database is such that the replicated data is used in a read-only manner or, more precisely, no replica changes are to be propagated to the Master, maintaining of the history data may be avoided by setting:

```
set sync parameter SYS_SYNC_KEEPLOCALCHANGES 'Yes';
```

In that case, the statement:

```
ALTER TABLE ... SET SYNCHISTORY
```

is not needed. If it has been already executed, the statement:

```
ALTER TABLE ... SET NOSYNCHISTORY
```

reverts the effect.

Note also that, in this case, ALTER TABLE ... SET HISTORY COLUMNS cannot be used.

9.2.4 Optimizing Synchronization Messages

solidDB SmartFlow data synchronization ensures that once data is committed in one database of a SmartFlow configuration, data is never lost during synchronization between databases. The SmartFlow Store and Forward Messaging feature ensures that before sending a synchronization message from one database to another, the message is stored on the originating database. Similarly, the message is stored in the receiving database before it is executed in the receiving database. Once the stored messages become obsolete, they are deleted.

Because Store and Forward Messaging stores data persistently to disk, it causes some overhead in the synchronization process. When a message holds less data for synchronization, the overhead is significantly higher. For example, when you send a synchronization message that contains one transaction between databases, it may take up to one second to complete the messaging roundtrip, whereas when you synchronize a few dozen transactions within one message, it still typically takes less than one second.

To minimize the overhead caused from Store and Forward Messaging, be sure to create synchronization messages that contain more than one transaction. While you are not prohibited from synchronizing a single transaction in a message, doing so has significant adverse performance implications, especially if your site has high transaction volume. Before you use single transactions (or few transactions) in a single synchronization message, consider if there are any critical performance and scalability requirements for the databases at your site.

Using RPC Message Compression with Synchronization

solidDB supports message compression for all network traffic between client and server. In data synchronization, the replica database server acts as a client and master database server acts as a client. Therefore, the message compression utility that is available for Client/Server communication, is also available for SmartFlow's Server-to-Server communication.

You can set the message compression on by specifying "-z" parameter in the connect string given in the SET SYNC CONNECT command in replica database server. For example:

```
SET SYNC CONNECT 'tcp -z masterserver 1315' TO MASTER myMaster
```

The data compression's effect on performance depends heavily on the compressability of the data and on the available bandwidth. In very fast networks the increased CPU consumption caused by compression & decompression of the messages may outweigh the performance gain achieved by smaller network messages. Generally, the slower the network is, the more positive impact network traffic compression may have in the overall performance.

Appendix A. Bulletin Board Parameters

solidDB allows both master and replica servers to use bulletin board parameters to store information that can be used when processing propagated transactions. This appendix lists the bulletin board parameters that are defined by solidDB.

Remember that you may create additional bulletin board parameters of your own when you create and use intelligent transactions.

Bulletin board parameters can be set by using the `PUT_PARAM()` function and can be read using the `GET_PARAM()` function.

A.1 SmartFlow System Parameter Categories

SmartFlow system parameters are divided into the following categories:

- *Read only system parameters* that are maintained by solidDB and can only be read by using the following syntax:

  `GET_PARAM(`*`parameter_name`*`)`

  The life cycle of parameters in this category is one transaction, that is, values of these parameters will always be initialized at the beginning of the transaction.

- *Updatable transaction-level system parameters* that can be set and updated by the user through

  `PUT_PARAM(`*`parameter_name, value`*`)`

  function call inside the transaction. Updatable system parameters are used by solidDB SmartFlow to configure synchronization-related operations.

  Like the category above, the life cycle of these parameters is also one transaction.

- *Database catalog level system parameters* that are set by using the followinf syntax:

`SET SYNC PARAMETER(`*`parameter_name value`*`)`

Parameters in this category are database catalog level parameters that are valid until changed or removed. They are specified as bulletin board parameters.

Full syntax and examples of usage of `GET_PARAM()`, `PUT_PARAM()` and `SET SYNC PARAMETER` functions are described in *solidDB SQL Guide*.

In the tables below, the "Duration" column indicates whether this parameter's value lasts for only the current transaction ("T") or whether it lasts until changed ("C").

The "R/W" column indicates whether the value can be only be read ("R/O") or can be read and written/updated ("R/W").

A.2 Parameters on Replica

Table A.1. Parameters on Replica

Name	Description	Factory Value	Duration	R/W
SYS_SYNC_ID	This parameter is for internal use only. Do not set it.	N/A	N/A	R/O
SYS_R_MAX-BYTES_OUT	The maximum size of a single synchronization message can be set by database level system parameters. The *SYS_R_MAXBYTES_OUT* parameter sets the maximum length of messages sent from a replica database to the master. The value of this parameter can be set only in the replica database. If the master database receives a message longer than the value of *SYS_R_MAXBYTES_OUT*, solidDB issues the following error message: 25042 - Message is too long (<*number*> bytes) to forward. Maximum is set to <*number*> bytes. Example:	2GB Valid values are between 0 - 2 GB. If 0 is specified, 2GB is used.	C (until changed)	R/W

Name	Description	Factory Value	Duration	R/W
	`SET SYNC PARAMETER SYS_R_MAXBYTES_OUT '1048576000';`			
SYS_R_MAXBYTES_IN	The maximum size of a single synchronization message can be set by database level system parameters. The *SYS_R_MAXBYTES_IN* parameter sets the maximum length of messages that can be received by a replica database. If the master database sends a message longer than value of *SYS_R_MAXBYTES_IN*, solidDB issues the following error message: 25043 - Reply message is too long (<*number*> bytes). Maximum is set to <*number*> bytes. Example: `SET SYNC PARAMETER SYS_R_MAXBYTES_IN '1048576000';`	2GB Valid values are between 0 - 2 GB. If 0 is specified, 2GB is used.	C (until changed)	R/W
SYS_SYNC_KEEPLOCALCHANGES	If the replica is read-only, setting this parameter to "Yes" reduces processing load and storage requirements at Replica. The ALTER TABLE SET SYNCHISTORY statement is not required for replica tables in that case, and thus history table are not created at the replica. With this setting, all the local changes and row inserts (if made) are kept despite the refresh operation.	No	Valid until changed	R/W
SYS_SYNC_OPERATION_TYPE	When the replica gets a `REFRESH` from the master, the parameters *SYS_SYNC_OPERATION_TYPE* and *SYS_SYNC_RESULTSET_TYPE* both provide the replica with information about what operations the master originally performed on this data.	None	T	R/W

Name	Description	Factory Value	Duration	R/W
	The primary reason for these parameters is to indicate whether an UPDATE operation occurred on the master and was "converted" into a DELETE+INSERT pair on the replica. For a more extensive discussion of these two parameters, see Section 5.2.4, “Handling UPDATE Triggers”. The possible values of the *SYS_SYNC_OPERATION_TYPE* parameter in DELETE triggers are: • *CURRENT_TENTATIVE_DELETE* (set when deleting the current locally updated value of a row prior to executing the reply message in replica) • *OLD_OFFICIAL_DELETE* (set when deleting a row that was deleted in master) • *OLD_OFFICIAL_UNIQUE_DELETE* (set when the master sends a row to be added to the replica, but a similar row already exists on the replica. With this parameter value, the old official row is deleted before the new row is added to the replica) • *OLD_OFFICIAL_UPDATE* (set when executing a delete that was created as a result of an update in master) The possible values of the *SYS_SYNC_OPERATION_TYPE* parameter in INSERT triggers are: • *OLD_OFFICIAL_INSERT* (set when restoring the old master value prior to executing the reply message in replica) • *NEW_OFFICIAL_INSERT* (set when inserting row that was inserted in master)			

Name	Description	Factory Value	Duration	R/W
	• *NEW_OFFICIAL_UPDATE* (set when executing an insert that was created as a result of an update in master) If the trigger is fired by a local transaction (that is, not by synchronization logic), then the value of this parameter is NULL.			
SYS_SYNC_RESULTSET_TYPE	This indicates whether the resultset of a SmartFlow REFRESH operation is full or incremental. Possible values of this parameter are: • FULL • INCREMENTAL See also the discussion of *SYS_SYNC_OPERATION_TYPE*. For a more extensive discussion of *SYS_SYNC_RESULTSET_TYPE* and *SYS_SYNC_OPERATION_TYPE*, see Section 5.2.4, “Handling UPDATE Triggers”.	None	T	R/W

A.3 Parameters on Master

Table A.2. Parameters on Master

Name	Description	Factory Value	Duration	R/W
SYNC_DEFAULT_PROPAGATE_ERRORMODE	This parameter controls what the server does when an error occurs while propagating a message. The possible values are *IGNORE_ERRORS*, *LOG_ERRORS*, or *FAIL_ERRORS*. The meanings of these values are the same as for the SAVE command. See the description of the SQL command SAVE in *solidDB SQL Guide*. Note that the error-handling mode may be set in three different ways (by setting a bulletin	By default, the system behaves as though the error mode is *FAIL_ERRORS*.	C	R/W

Name	Description	Factory Value	Duration	R/W
	board parameter, by specifying an optional keyword with the SAVE command, or by using the `MESSAGE APPEND PROPAGATE TRANSACTIONS` statement). See Section 6.7.3, “Specifying Recovery from Fatal Errors” for more details.			
SYNC_DEFAULT_PROPAGATE_SAVEMODE	The possible values are *AUTOSAVE*, *AUTOSAVEONLY*, and *NULL*. Null means that the propagated transaction is not automatically saved. See Section 6.7.3, “Specifying Recovery from Fatal Errors” for more details.	NULL	C	R/W
SYS_ERROR_CODE	This parameter can be used together with the *SYS_ROLLBACK* parameter. Users can set their own error code in this parameter to indicate the reason why the transaction was rolled back. This error code is returned after roll back. The error code specified using this parameter is also returned to the replica database as part of the MESSAGE FORWARD or MESSAGE GET REPLY command. Example: `PUT_PARAM( 'SYS_ERROR_CODE','99000' );`	None	T	R/W
SYS_ERROR_TEXT	This parameter can be used together with *SYS_ROLLBACK* parameter. Users can put their own error text in this parameter to indicate the reason why the transaction was rolled back. This error text is returned after roll back. Example:	None	T	R/W

Name	Description	Factory Value	Duration	R/W
	`PUT_PARAM( 'SYS_ERROR_TEXT', 'User defined error text' );`			
SYS_IS_PROPAGATE	This parameter value is *YES* if the transaction is a propagated transaction that is being executed in the master. For non-SmartFlow transactions, the value of this parameter is *NULL*.	None	T	R/O
SYS_NOSYNCESTIMATE	In many cases, either an incremental refresh or a full refresh would synchronize the replica. An incremental refresh is usually the most efficient choice. However, there are cases where a full refresh is more efficient. As an extreme example, if 99% of the rows in a table have been deleted, then it is more efficient to send a full refresh (1% of the rows) than to send each individual row that was deleted (99% of the rows). By default (*SYS_NOSYNCESTIMATOR=None*), the server calculates whether an incremental refresh is more efficient than a full refresh and then chooses the one that it thinks is more efficient. You can turn off the calculation and thus force the server to use an incremental refresh by setting *SYS_NOSYNCESTIMATOR* to *Yes*. This parameter is used only rarely. To disable the estimator, execute on master: `SET SYNC PARAMETER`	"None" (This means that the estimator is not disabled. This does not mean that there is no factory value.)		R/W

Name	Description	Factory Value	Duration	R/W
	SYS_NOSYNCESTIMATE 'YES'; COMMIT WORK; To enable the estimator (=default), execute on master: SET SYNC PARAMETER SYS_NOSYNCESTIMATE NONE; COMMIT WORK;			
SYS_ROLLBACK	This parameter is used inside a transaction when the execution of the transaction should be rolled back. If the value of this parameter is set to *YES*, then using the PUT_PARAM function causes solidDB to stop execution of the transaction and roll back all statements executed already. Rollback of the transaction will cause the execution of the synchronization message to halt. Note that COMMIT WORK and ROLLBACK WORK commands are not allowed in propagated transactions. *SYS_ROLLBACK* can be used, for instance, if a transaction detects a fatal error such as a referential integrity error in the database. Example: PUT_PARAM('SYS_ROLLBACK','YES');	The factory value is "No".	T	R/W
SYS_TRAN_ID	This parameter is valid only for propagated transactions, that is, when a propagated transaction is executed in the master database *SYS_TRAN_ID* will contain the original transaction's id from replica database. For non-SmartFlow transactions, the value of this parameter is NULL.	None	T	R/O

Name	Description	Factory Value	Duration	R/W
SYS_TRAN_USERID	This parameter is valid only for propagated transactions, that is, when the propagated transaction is executed in the master database *SYS_TRAN_USERID* will contain the original user id used when the transaction was executed in the replica database. The transaction is executed in the master database using the access rights defined to this user id. If the user id was mapped from the replica to a master user id, the access rights used during execution in the master database are the access rights of the master user id (not the original replica user id). For non-SmartFlow transactions, the value of this parameter is *NULL*.	None	T	R/O

A.4 Parameters on both Master and Replica

Table A.3. Parameters on both Master and Replica

Name	Description	Factory Value	Duration	R/W
SYNC_APP_SCHEMA_VERSION	Each synchronizable catalog can have a schema version name as one of its properties. If master and replica catalogs have different schema version names, sending the synchronization messages between master and replica nodes fails. The name of the property is *SYNC_APP_SCHEMA_VERSION*. This version name can be set using SET SYNC PARAMETER statement. Examples: `SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION 'sputnik';`	None	C	R/W

Name	Description	Factory Value	Duration	R/W
	`SET SYNC PARAMETER SYNC_APP_SCHEMA_VERSION NONE;`			
SYNC_MODE	Each catalog has a read-only parameter named *SYNC_MODE*. Applications use this parameter to check the catalog's mode. The parameter values are: • *MAINTENANCE*, if the catalog is in maintenance sync mode • *NORMAL*, if the catalog is not in maintenance sync mode • *NULL*, if the catalog is not a master or a replica You can change the catalog mode with command: `SET SYNC MODE{NORMAL | MAINTENANCE}`	NORMAL	C	RO
SYS_TRAN_MAXRETRY	This parameter provides a way to handle concurrency conflicts and deadlocks, which occur when transactions update or delete rows. When this parameter is set, the server retries a transaction that has failed execution at the master due to a concurrency conflict. The value specifies the maximum number of retries. You can set this value on the master by using the SET SYNC PARAMETER statement. If you set it on the master, it will be the default value used for all transactions propagated to that master. Note that this parameter only ap-	The factory value is zero (0), which means that if the transaction fails, it is not retried. Valid values for the parameter are integers between 0 and 2147483647 (inclusive).	C	R/W

Name	Description	Factory Value	Duration	R/W
	plies when the master executes propagated transactions received from the replica. You can set this value on the replica by using the command. `SAVE PROPERTY ...` to put the value on the individual transaction's bulletin board. The value is propagated to the master's bulletin board for that same transaction. Setting the parameter this way means that the value applies only to this particular transaction. If the value is set both by the SAVE PROPERTY command on the replica and the SET SYNC PARAMETER command on the master, the SAVE PROPERTY command on the replica takes precedence. The value that you set for a specific transaction takes precedence over the general default value on the master. When *SYS_TRAN_MAXRETRY* reaches its retry' maximum, the offending transaction is marked as failed. A separate MESSAGE EXECUTE command must be executed to start the transaction again. *SYS_TRAN_RETRYTIMEOUT* is closely related to the *SYS_TRAN_MAXRETRY* parameter. See the discussion of *SYS_TRAN_RETRYTIMEOUT* below for details.			
SYS_TRAN_RETRYTIMEOUT	This parameter is used in conjunction with the *SYS_TRAN_MAXRETRY* parameter. The *SYS_TRAN_MAXRETRY* parameter specifies the number of transaction retry attempts that occur after the transaction has failed at the	The factory value is zero (0), which means that the master server does not wait between re-	C	R/W

Name	Description	Factory Value	Duration	R/W
	master due to a concurrency conflict. The *SYS_TRAN_RETRYTIMEOUT* parameter sets the timeout (in seconds) that the master server waits before it actually retries the failed transaction that was received from the replica. The value of this parameter can be set only in the master database using the SET SYNC PARAMETER statement. For details on *SET SYNC PARAMETER*, read the chapter on "SET SYNC PARAMETER" in Appendix B of *solidDB SQL Guide*.	tries. Valid values for the parameter are any integer between 0 and 2147483647 (inclusive).		

Appendix B. Synchronization Events

This chapter documents Synchronization (SmartFlow) Events. These events are provided with the solidDB to allow programs to be notified when certain SmartFlow-related actions occur. You can use these events to monitor the progress of synchronization between master and replica databases.

These events follow most of the same rules as any other events. The main difference is that you cannot register to synchronization events, because the

ADMIN EVENT 'wait'

command is not able to return variable resultsets. Instead, you must use stored procedures to handle synchronization events. For information about events in general, see the *solidDB SQL Guide*, especially the sections on:

- the CREATE EVENT command
- the CREATE PROCEDURE command (which describes how to post events and wait on events)
- the chapter titled "SQL Programming: Stored Procedures, Events, Triggers, and Sequences", which discusses events extensively.

Because these events are pre-defined, you do not create them. Furthermore, you should not post any system event. You should only register for and wait on system events.

Many, although not all, system events have the same five parameters:

- *ename*: The event name.
- *postsrvtime*: The time that the server posted the event.
- *uid*: The user ID (if applicable).
- *numdatainfo*: Miscellaneous numeric data — the exact meaning depends upon the event. For example, the event SYS_EVENT_BACKUP is posted both when a backup is started and when a backup is completed. The value in the *numdatainfo* parameter indicates which case applies — i.e. whether the backup has just started or has just completed. This parameter may be NULL if there is no numeric data.
- *textdata*: Miscellaneous text data — the exact meaning depends upon the event. This parameter may be NULL if there is no numeric data.

This appendix contains the following tables:

1. Synchronization-Related Events: The order in which such events occur when a replica sends a message to a master.

2. Synchronization-Related Events: The parameters of each type of Synchronization-related event.

3. Events that help you monitor the progress of messages, i.e. how many bytes have been sent or received so far.

B.1 Events when Replica Propagates Messages to Master

B.1.1 Sequence of Events

The table below shows the sequence of events posted as the replica and master process a message that is sent from the replica to the master.

Note that some events do not always occur. For example, this sequence shows some places that the event SYNC_MASTER_MESSAGE_ERROR_OCCURRED may occur; however, the event does not always occur there.

Note also that in some cases the order of events may vary slightly. For example, the time that a master deletes an old message is partly independent of the activities of the replica (and vice-versa), and therefore may not always be done in exactly the order shown here.

Note

Applications cannot monitor sync events unless they have administrator's rights.

A separate table later gives more information about the parameters used in each of these synchronization-related events.

Table B.1. Events when Replica Propagates Messages to Master

Action, command, or situation on Replica	EVENT Posted on Replica	Action or situation on Master	EVENT Posted on Master
Message Begin			
Message Append			
Message end / commit	When message is persistent (i.e. assembled and committed):		

Action, command, or situation on Replica	EVENT Posted on Replica	Action or situation on Master	EVENT Posted on Master
	SYNC_REPLICA_MESSAGE_ASSEMBLED		
Message forward	When replica starts sending the message: SYNC_REPLICA_MESSAGE_FORWARD_BEGIN When replica has finished sending the message to the master: SYNC_REPLICA_MESSAGE_FORWARD_END		
		Master starts receiving a message	When master starts receiving the message: SYNC_MASTER_MESSAGE_RECEIVE_BEGIN
		Master has received a message	When master has received and made message persistent: SYNC_MASTER_MESSAGE_RECEIVE_END
		Master starts processing the message	SYNC_MASTER_MESSAGE_REPLY_BEGIN
		Master processed the message	SYNC_MASTER_MESSAGE_REPLY_END
		If an error occurred while processing the message.	SYNC_MASTER_MESSAGE_ERROR_OCCURRED
Message get reply	When replica sends get-reply request to master: SYNC_REPLICA_MESSAGE_GETREPLY		
		Master receives "get reply" request from replica	SYNC_MASTER_MESSAGE_GETREPLY_REQUEST
If "get reply" request timed out.	SYNC_REPLICA_MESSAGE_GETREPLY_TIMEDOUT		
		Master starts sending reply message	SYNC_MASTER_MESSAGE_SENDREPLY_BEGIN

Action, command, or situation on Replica	EVENT Posted on Replica	Action or situation on Master	EVENT Posted on Master
Replica starts receiving the reply from the master.	SYNC_REPLICA_MESSAGE_REPLY_BEGIN		
		Master finishes sending reply.	SYNC_MASTER_MESSAGE_SENDREPLY_END
Reply received and made persistent	SYNC_REPLICA_MESSAGE_REPLY_END		
		Message is deleted	SYNC_MASTER_MESSAGE_DELETED. Note that the message also can be deleted explicitly using the MESSAGE DELETE statement.
Replica starts processing the reply.	SYNC_REPLICA_MESSAGE_PROCESS_BEGIN		
If an error occurred while processing	SYNC_REPLICA_MESSAGE_ERROR_OCCURRED		
Reply is processed	SYNC_REPLICA_MESSAGE_PROCESS_END		
Message is deleted	SYNC_REPLICA_MESSAGE_DELETED. Note that the message also can be deleted explicitly using the MESSAGE DELETE statement.		

B.1.2 Parameters of Synchronization-Related Events

The table below shows the parameters associated with each synchronization-related event.

Table B.2. Parameters Associated with Synchronization-Related Events

EVENT NAME	Purpose	PARAMETERS
SYNC_MASTER_MESSAGE_DELETED	This event is posted on the master when the master has deleted a message.	master_name WVARCHAR, replica_name WVARCHAR,

EVENT NAME	Purpose	PARAMETERS
		message_name WVARCHAR
SYNC_MASTER_MESSAGE_ERROR_OCCURRED	This event is posted on the master when an error occurred while processing the message.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR, error_code INTEGER, error_message WVARCHAR
SYNC_MASTER_MESSAGE_GETREPLY_REQUEST	This event is posted on the master when the master receives a reply request from a replica. The parameter *request_timeout* holds the requested reply timeout in seconds. The parameter *iftimedout* holds one of the following values: 0 - if request has not timed out in the master 1 - if request timed out in the master The master may not start processing the reply request immediately. Timeout is set to 1 if the master is unable to start processing the reply within the timeout period requested by the replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR, request_timeout INTEGER, iftimedout INTEGER
SYNC_MASTER_MESSAGE_RECEIVE_BEGIN	This event is posted on the master when the master begins to receive a new message from a replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_MESSAGE_RECEIVE_END	This event is posted on the master when it finishes receiving a new message from a replica.	master_name WVARCHAR, replica_name WVARCHAR,

EVENT NAME	Purpose	PARAMETERS
		message_name WVARCHAR
SYNC_MASTER_MESSAGE_REPLY_BEGIN	This event is posted on the master when it starts creating a reply message to a replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_MESSAGE_REPLY_END	This event is posted on the master when it finishes creating a reply message to a replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_MESSAGE_SENDREPLY_BEGIN	This event is posted on the master when it starts sending a reply message to a replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_MESSAGE_SENDREPLY_END	This event is posted on the master when master has finished sending a reply message to a replica.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_REGISTER_REPLICA	This event is posted on the master when a new replica is registered to the master.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_MASTER_UNREGISTER_REPLICA	This event is posted on the master when a replica is unregistered from the master.	master_name WVARCHAR, replica_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_ASSEMBLED	This event is posted by the replica when it creates a new message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_DELETED	The replica posts this event when it deletes a message.	replica_name WVARCHAR, master_name WVARCHAR,

EVENT NAME	Purpose	PARAMETERS
		message_name WVARCHAR
SYNC_REPLICA_MESSAGE_ERROR_OCCURRED	This event is posted on the replica when an error occurred while processing the message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR, error_code INTEGER, error_message WVARCHAR
SYNC_REPLICA_MESSAGE_FORWARD_BEGIN	This event is posted by a replica when it starts forwarding a message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_FORWARD_END	This event is posted by a replica when it finishes forwarding a message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_GETREPLY	A replica posts this event when it requests a reply message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_GETREPLY_TIMEDOUT	A replica posts this event when the replica's "get reply" has timed out.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_PROCESS_BEGIN	The replica posts this event when it starts processing a reply message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_PROCESS_END	The replica posts this event when it finishes processing a reply message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR

EVENT NAME	Purpose	PARAMETERS
SYNC_REPLICA_MESSAGE_REPLY_BEGIN	The replica posts this event when it starts receiving a reply message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR
SYNC_REPLICA_MESSAGE_REPLY_END	The replica posts this event when it finished receiving a reply message.	replica_name WVARCHAR, master_name WVARCHAR, message_name WVARCHAR

B.1.3 Parameters of Message Progress Events

The events described in this table allow you to track how many bytes of a message have been sent or received so far. These events are useful primarily when sending very large messages, such as messages containing blobs, or when sending messages over very slow communication channels.

Table B.3. Parameters of Message Progress Events

EVENT NAME	Purpose	PARAMETERS
SYNC_MSGBYTES_SENT	The event lists the cumulative number of bytes sent/received so far as well as the total number of bytes to be sent/received inside the corresponding sync message. The user can monitor the progress of the sending/receiving process by catching these events and comparing the cumulative byte count sent so far to the total bytes in the message. For more information, see Section 9.1, “Monitoring the Progress of Messages”	sender_nodename WVARCHAR receiver_nodename WVARCHAR message_name WVARCHAR cumulative_bytes_sent INTEGER total_bytes INTEGER
SYNC_MSGBYTES_RECEIVED	The event lists the cumulative number of bytes sent/received so far as well as the total number of bytes to be sent/received inside the corresponding sync message. The user can monitor the progress of the sending/receiving process by catching these events and comparing the cumulative byte count sent so far to the total bytes in the message. For	sender_nodename WVARCHAR receiver_nodename WVARCHAR message_name WVARCHAR cumulative_bytes_received INTEGER total_bytes INTEGER

EVENT NAME	Purpose	PARAMETERS
	more information, see Section 9.1, “Monitoring the Progress of Messages”	

Glossary

A

Asynchronous store and forward messaging
: The messaging architecture of solidDB SmartFlow. All messages are stored to solidDB database prior to sending them to the other database. This ensures that no messages are ever lost in the synchronization process.

B

Binary Large Object (BLOB)
: "BLOB" is an acronym for Binary Large OBject. A BLOB is a large block of information such as a picture, video clip, sound excerpt, or a document that contains non-printable formatting characters as well as printable characters.

  BLOB information is usually stored in a high capacity, variable-length binary data type. With solidDB, BLOB data is usually stored in VARBINARY. However, this is not always necessary. Although BLOBs are generally Binary and Large, and are usually stored in variable-length data types, none of these characteristics are required. Depending upon the actual data value, you might store your data in a fixed-length BINARY field rather than a variable-length VARBINARY field. If your data is composed entirely of standard characters, then you might store the data in one of the various high-capacity character data types, such as VARCHAR. (BLOBs that are composed entirely of printable characters are sometimes called CLOBs. Since BINARY fields can store any data that CHAR fields can store, CLOBs can be stored in either CHAR or BINARY fields. CLOBs are a subset of BLOBs.)

  For a complete list of the BINARY and CHAR data types supported by solidDB, see the *solidDB SQL Guide*.

  With solidDB, BLOB/CLOB data is treated the same way as any other BINARY/CHAR data. You do not need to do anything special to store or retrieve such data. BLOB usage is thus transparent with solidDB.

C

Catalog
: See also Schema.

A catalog logically partitions a solidDB database so that data is organized in ways that meet business or application requirements. Each logical database is a *catalog* and contains a complete, independent group of database objects, such as tables, indexes, procedures, triggers, etc. Note, however, that a solidDB catalog contains a variety of data objects, not just indexes (as in the traditional sense of a library card catalog, which serves to locate an item without containing the full contents of the item).

Each of these catalogs can act as an independent master or replica database. This makes it possible, for example, to create two or more independent replica databases in one physical local database. It is also possible to have one or more catalogs in this same local database that represent master database(s).

A catalog is also referred to as a *node* when the catalog has been defined in a master or replica using the SET SYNC NODE command. Each catalog of a SmartFlow environment must have a node name that is unique within the domain. Assigning the node name is part of the registration process of a replica database.

A catalog can qualify one or more schemas. A schema is a persistent database object that provides a definition for the entire database; it represents a collection of database objects associated with that specific schema name. The catalog name is used to qualify a database object name, such as tables, views, indexes, stored procedures, triggers, and sequences. They are qualified as: *catalog_name.schema_name.database_object* or *catalog_name.user_id.database_object*.

Inside each catalog there may be multiple schemas. It is legal to use the same schema name in more than one catalog. Typically, each user in a catalog is allowed to have his or her own schema(s). Providing users with their own schema allows each user to have his or her own tables (or other database objects) without naming overlaps.

Character Large OBject (CLOB)
: See also Binary Large Object.

  CLOBs are a subset of BLOBs.

Communication protocol
: A communication protocol is a set of rules and conventions used in the communication between servers and clients. The server and client have to use the same communication protocol in order to establish a connection.

D

Database administrator
: The database administrator is a person responsible for tasks such as:

  - managing users, tables, and indices

- backing up data
- allocating disk space for the database files

F

Full publication
: A full publication returns all subscribed rows of the publication from the master to a replica database. The initial REFRESH is always a full publication. If the tables in the publication are set for incremental publication, subsequent REFRESHes for the same publication contain only the data that has been changed since the prior REFRESH.

I

Incremental publication
: A publication that returns only those rows of a publication that have been inserted, updated or deleted in the master database since the previous refresh.

Intelligent Transaction
: A Solid Intelligent Transaction is an extension to the traditional transaction model. It is a collection of SQL statements that may contain business logic that is typically implemented as Solid stored procedures. These procedures are able to communicate with each other using the Parameter Bulletin Board of the transaction. A transaction that is intelligent is capable of validating itself in the current database and adapting its contents (if required) according to the rules of the transaction.

  Since an intelligent transaction is created in the replica database, but is finally committed in the master database, it is a long-lived transaction. Therefore all validity checking of the transaction must be done by the transaction itself.

L

Local Data
: See also Local Database.

  Data is considered "local" to a database if that data is not shared with any other database. This means the local data is not visible from any other database. In other words, data is local if the data is neither part of a "replica" of another database nor part of a "master" for another database.

Local Database
: See also Local Data.

In this guide when discussing a specific example code or an SQL command, the local database refers to the database on which the sample code is running (that is, the database to which a user is connected).

In those places where synchronization is discussed in this guide, it is assumed that the user is connected to the "replica", not the "master"; thus the "replica" database and the "local" database refer to the same database in most examples used in this guide.

In those scenarios where there are three or more levels in a synchronization configuration, the "middle" level may be both a replica of one database and a master to another database that is at a lower level. Again, however, the "local" database, in general, refers to the database to which the user is connected and on which the user executes commands.

M

Master database, also known as "master"
See also Local Database.

A database that contains the official version of all data and provides publications for replicas to subscribe to.

Messages
See also Asynchronous store and forward messaging.

See also Synchronization message.

In a multi-master synchronization model, a database (which, for the purpose of this definition, is known as the "local" database) can contain replicated information from multiple masters. For each master catalog, a replica catalog is created in the local database, and the replica catalog stores only data from its corresponding master, not from any other master. This keeps replica data from different masters separated.

Furthermore, the local database may also contain data that is neither a replica of some other database, nor is referenced as "master" data by any other database. A replica or master catalog can contain tables or parts of tables that are not replicated. In this way, the server can keep local data separate from shared data.

Note that in a multi-master environment, the ability to have multiple catalogs allows you to specify multiple logical databases (master or replica) for synchronization within one physical solidDB server.

N

Network name
: The network name of a server consists of a communication protocol and a server name. This combination identifies the server in the network.

  solidDB clients support Logical Data Source Names. These names can be used to give a database a descriptive name. This name is mapped to a network name using either parameter settings in the clients `solid.ini` file or in Microsoft Windows operating systems' registry settings.

Node
: See Catalog.

O

Official data (official version)
: In a SmartFlow system, data of the master database is considered "official." Transactions that modify replica data are always tentative. Once the transactions are propagated and committed to the master database, modifications become the official version.

P

Parameter passing
: The stored procedures of an Intelligent Transaction are able to pass parameters to each other via a parameter bulletin board.

Parameter Bulletin Board
: A data transfer area within a transaction that can be used for passing parameters from one stored procedure to another one. It can also be used for storing transaction properties. Parameters are specific to a catalog. Different replica and master catalogs have their own set of bulletin board parameters that are not visible to each other. The SET SYNC PARAMETER command defines catalog-level parameters.

Publication
: A predefined, consistent subset of master data that a replica can request (subscribe to). A publication allows users to subscribe to different subsets within the publication.

R

Refresh
: When a SmartFlow replica requests data from a master using the command MESSAGE APPEND REFRESH, the operation is called a "refresh". Note that although the word "refresh" implies that the user has gotten the data at least once before (i.e. this is the 2nd or later request) we use the term loosely to apply to all requests, including the initial one.

Replica database
: See also Local Database.

  A SmartFlow database that contains a subset of master data and some tentative local transaction data.

S

Schema
: See also Catalog.

  A schema is a database object that may contain other database objects (such as tables, views, etc.); schemas allow you to organize your database objects and schemas prevent multiple users from conflicting when they choose identical object names (such as table names). Within each schema, each data object (such as a table) must have a unique name. However two different users may use the same table name in different schemas, for example, Sue Lamm and Dan Wong could each have a table named *table1*.

  In this way, schemas are like the directories or operating systems. Each directory contains zero or more files; within each directory, each filename must be unique, but two different directories might contain different files with the same name. Schemas are part of a hierarchy in this order: database, database catalog, schema, database object (for example, table).

  Within each database, each catalog name must be unique. Within each catalog, each schema name must be unique. Within each schema, each database object name must be unique. Note that a schema cannot contain another schema; in this way schemas are unlike directories. (A directory may contain another directory, but a schema may not contain another schema).

  Any table, view, etc. within a database can be uniquely identified by specifying its "fully qualified" name, which includes the catalog name, the schema name, and the database object name, for example:

  sales_catalog.sue_lamm.table1

  sales_catalog.dan_wong.table1

Fully-qualified names are always unique. Note also that each table or other database object belongs to exactly one schema; a table may not be part of more than one schema (or more than one catalog).

Schemas do not provide any privacy or security. By specifying the fully qualified name of a database object (such as a table), you may access database objects in other users' schemas (assuming that you have appropriate privileges on those objects); schemas do not prevent you from accessing data owned by other users, or vice versa.

By default, each user has his or her own schema, the name of which is the same as the user's login name. For example, if Sue Lamm logs in as sue_lamm, then when she connects to a database she will automatically be connected to the sue_lamm schema. She may change to a different schema by using the SET SCHEMA command. Within a particular catalog and schema, you do not need to specify the fully-qualified name; for example, if you have already executed:

```
SET CATALOG 'sales_catalog';
SET SCHEMA 'sue_lamm';
```

then if you specify only *table1*, the database server knows to use the *table1* in the *sue_lamm* schema of the catalog named *sales_catalog*. Although each user has a default schema name based on his or her login, a user is not restricted to owning only that one schema. A user may create additional schemas by using the CREATE SCHEMA command.

Server name
: When you specify the value of the `solid.ini` configuration file, you must specify the network name of the server. The network name of a server consists of a communication protocol and a server name. This combination identifies the server in the network. The protocol must be one of the standard communication protocols, such as TCP/IP ("tcp"), named pipes ("nmpipe"), etc. The valid values for the server name depend upon the protocol and on whether the client and server are running on the same computer. The server name might be a name, such as "calvin" or "chicago_office", or it might be a node name and a service port, such as "hobbes 1313", or it might be just a service port, such as "1313".

Subscription
: The definition of a specific refreshable subset of master data in the replica; that is, an incarnation of a publication in one replica.

Synchronization history
: A set of data containing the history of changes (such as updates and deletes) to the data in a particular table. This history data is stored in a "shadow" table; there is one "shadow" table for each table that history data is recorded for. This history information is required for incremental publications to work.

Synchronization message
: A message consisting of one or several synchronization operations (propagation of transactions or refresh requests).

Synchronization process
: A process that controls the assembly and sending of synchronization messages at a replica database.

T

Tentative data
: All locally-changed data in a replica database that is set up for synchronization. The official version of this data is in the master database.

Three-tier client/server architecture model
: Compared to the two-tier architecture, the three-tier architecture has an additional layer or layers of application servers. This allows splitting the application logic between client processes to a specialized application server process handling the resources management, other I/O, or calculation intensive tasks.

  Instead of sending small SQL statements the client application sends whole procedures for the application server to process. This reduces the number of messages thus minimizing the network load. The application logic is often more easily managed because several applications use centrally maintained procedures.

Transaction propagation
: A mechanism enabling transactions to be transferred and re-executed at the master database after executing them at a replica database.

Two-tier client/server architecture model
: Generally, the two-tier architecture refers to a client/server system, where a client application containing all the business logic is running on a workstation and a database server is taking care of data management.

Two-tier replication model
: The two-tier redundancy model refers to a synchronization architecture that has one master database and multiple replica databases; the replicas are updateable but the replica data is *tentative* until the data has been successfully propagated to the master database and committed there.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U