SOLID *SynchroNet* ™ Guide

June, 2000
Version 2.00

SOLID

Solid Information Technology Ltd.
www.solidtech.com
sales@solidtech.com;techsupp@solidtech.com

Document number SSSG-2.00-0600
Date: June 26, 2000

Contents

Welcome

SOLID SynchroNet^{TM} is a data management and distribution product for today's smart networks.

SOLID *SynchroNet* local data storage capabilities are particularly suited for applications in today's internetworked systems. With SOLID *SynchroNet's* asynchronous, bidirectional data synchronization, you can store data where it makes sense and distribute the data where it adds value.

About This Guide

The **SOLID *SynchroNet* Guide** introduces you to synchronization concepts and architecture and describes how to set up, use, and administer SOLID *SynchroNet*.

Organization

This guide is divided into three parts which contain the following chapters:

Part I, SOLID *SynchroNet* Planning and Designing

- *Chapter 1, SOLID SynchroNet Data Management Engine*, familiarizes you with the underlying components of SOLID *SynchroNet*.
- *Chapter 2, SOLID SynchroNet Data Synchronization Architecture*, familiarizes you with the *SynchroNet* features, components, concepts, and process architecture.

Part II, SOLID *SynchroNet* Engine Administration

- *Chapter 3, Administering SOLID SynchroNet*, covers the typical administration tasks such as starting, connecting to, and shutting down servers. It also explains how to perform routine maintenance such as creating backups and checkpoints, and using timed commands. In addition, it shows you how to maintain your synchronization environment.

- *Chapter 4, Using SOLID SQL for Data Management*, gives readers the information they need to manage users, tables, indexes, transactions, and database objects, such as catalogs and schemas.
- *Chapter 5, Using SOLID Data Management Tools,* describes the available utilities for performing database administration tasks, specifying SQL commands and queries, and performing specific database operations, such as loading and unloading databases.
- *Chapter 6, Managing Network Connections,* describes how to connect to SOLID *SynchroNet* using different communication protocols.
- *Chapter 7, Configuring SOLID SynchroNet,* describes how to set SOLID *SynchroNet* parameters for customization to meet your own environment, performance, and operations needs.
- *Chapter 8, Performance Tuning,* describes how to optimize SOLID *SynchroNet* to improve performance.
- *Chapter 9, Diagnostics and Troubleshooting,* describes tools to use for observing and tracing performance problems.

Part III, SOLID *SynchroNet* Implementation and Programming

- *Chapter 10, Getting Started with Data Synchronization,* provides a quick tour (using sample scripts) for setting up synchronization with basic *SynchroNet* commands—proprietary extensions to SOLID SQL.
- *Chapter 11, Planning and Designing for SOLID SynchroNet Applications,* describes the issues and considerations for designing a distributed system using *SynchroNet.*
- *Chapter 12, Implementing a SOLID SynchroNet Application,* describes the basic tasks required to implement synchronization.
- *Chapter 13, SOLID SynchroNet SQL Reference,* provides a command reference of *SynchroNet* extensions to SOLID SQL, allowing you to perform synchronization tasks.

Part IV, Appendixes

The *Appendixes* give you detailed information about system tables, error messages, configuration parameters, reserved words, and standard SOLID SQL syntax and data types.

Glossary

The *Glossary* provides definitions of *SynchroNet* terminology.

Audience

This guide assumes the reader has general DBMS knowledge and a familiarity with SQL.

Document Conventions

This manual uses the following typographic conventions.

Format	Used for
WIN.INI	Uppercase letters indicate filenames, SQL commands, macro names, and terms used at the operating-system command level.

`RETCODE`
`SQLFetch(hdbc)`
	This font is used for sample command lines and program code.

argument	Italicized words indicate information that the user or the application must provide, or word emphasis.
SQLTransact	Bold type indicates that syntax must be typed exactly as shown, including function names.
[ ]	Brackets indicate optional items or syntax elements; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{}	Braces 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.

Other SOLID Documentation

SOLID *SynchroNet* documentation is distributed in electronic format (PDF, HTML, or Windows Help files).

SOLID Online Services on our Web server offer the latest product and technical information free of charge. The service is located at:

`http://www.solidtech.com/`

Electronic Documentation

- **Read Me** contains installation instructions and additional information about the specific product version. This `readme.txt` file is typically copied onto your system when you install the software.
- **Release Notes** contains additional information about the specific product version. This `relnotes.txt` file is typically copied onto your system when you install the software.
- **SOLID Programmer Guide** describes the interfaces (APIs and drivers) available for accessing SOLID *SynchroNet* and how to use them with a SOLID server database.

Part I

SOLID *SynchroNet* Architecture

Part I introduces you to *SynchroNet* fundamentals and shows you how to plan and design your *SynchroNet* installation. It contains the following topics:

1

SOLID *SynchroNet* Data Management Engine

This chapter provides the underlying components and processes that make SOLID *Synchro-Net* the solution to managing distributed data in today's complex distributed system environments. It provides you with the background necessary to administer and maintain SOLID *SynchroNet* in your networked environment.

SOLID *SynchroNet* Data Management Components

SOLID *SynchroNet* includes the components described in the following sections.

Programming interfaces (ODBC and JDBC)

SOLID provides ODBC and JDBC drivers for programming access to SOLID data. SOLID *ODBC Driver* conforms to the Microsoft ODBC 3.51 API standard. SOLID *ODBC Driver* supported functions are accessed with SOLID ODBC API, a Call Level Interface (CLI) for SOLID databases, which is compliant with ANSI X3H2 SQL CLI. The SOLID *JDBC Driver* allows for application development with a Java tool that accesses the database using JDBC. For more details on programming interfaces, read the **SOLID Programmer Guide**.

Network Communications Layer

SOLID *SynchroNet* runs on all major network types and supports all of the main communication protocols. Developers can create distributed applications for use in heterogeneous computing environments. Read *Chapter 6,"Managing Network Connections,"* for more details on network communication.

SQL Parser and Optimizer

The SQL syntax used is based on the ANSI X3H3-1989 Level 2 standard and ANSI X3H3-1992 (SQL2) extensions. SOLID *SynchroNet* contains an advanced cost-based optimizer, which ensures that even complex queries can be run efficiently. The optimizer automatically maintains information about table sizes, the number of rows in tables, the available indices, and the statistical distribution of the index values.

Read *Chapter 9, "Diagnostics and Troubleshooting"* for more details on the SOLID SQL Optimizer.

Optimizer Hints

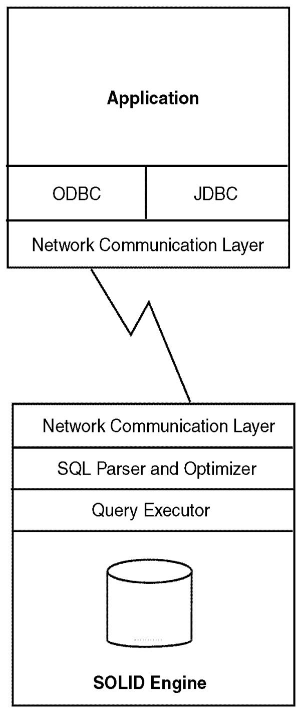
Optimizer hints (which is an extension of SQL) are directives specified through embedded pseudo comments within query statements. The optimizer detects these directives or *hints* and bases its query execution plan accordingly. Optimizer hints allow applications to be optimized under various conditions to the data, query type, and the database. They not only provide solutions to performance problems occasionally encountered with queries, but shift control of response times from the system to the user.

Read *"Using Optimizer Hints"* on page 8-8 for more details on optimizer hints.

Engine

The SOLID engine is the core of the SOLID *SynchroNet* product. It processes the data requests submitted via SOLID SQL. The engine stores data and retrieves it from the database.

Figure 1–1 SOLID SynchroNet Components



System Tools and Utilities

SOLID *SynchroNet* also includes the following tools for data management and administration:

SOLID *DBConsole*

SOLID *DBConsole* is an easy-to-use graphical user interface for administering and monitoring SOLID data management engines and executing SQL queries and commands. With SOLID *DBConsole*, you can:

- execute SQL commands
- administer all database servers in a network from a single workstation
- generate backups either on-line or as a timed command
- obtain server status information

- use either the interactive or batch mode operation
- have multiple active connections to various servers
- save or print query results

SOLID *Remote Contro*l (teletype) and SOLID *SQL Editor* (teletype) are also available to manage databases from the command line.

Tools for handling ASCII data

SOLID *SynchroNet* provides the following tools for handling ASCII data:

- SOLID *Speedloader* (SOLLOAD) loads data from external ASCII files into a SOLID database. It is capable of inserting character data from character format. SOLID *Speed-Loader* bypasses the SQL parser and uses direct writes to the database file with loading, which allows for fast loading speed.
- SOLID *Export* (SOLEXP) writes from a SOLID database to character format files. It is capable of writing control files used by SOLID *SpeedLoader* to perform data load operations.
- SOLID *Data Dictionary* (SOLDD) writes the data dictionary of a database. This tool produces a SQL script that contains data definition statements describing the structure of the database.

Read *Chapter 5, "Using SOLID Data Management Tools"* for details.

SOLID *SynchroNet* High Performance Architecture

What differentiates SOLID *SynchroNet* from other data management products is its high performance architecture. This section provides conceptual information, which can give you an understanding in configuring SOLID *SynchroNet* to meet the needs of your own applications and platforms. It takes a behind the scenes look at the following processes:

- Bonsai Tree Multiversioning and Concurrency Control
- Dynamic SQL Optimization
- Network Services
- Multithread processing

SOLID Bonsai Tree Multiversioning and Concurrency Control

The Bonsai Tree is a small active index that efficiently stores new data (deletes, inserts, updates) in central memory, while maintaining multiversion information. Multiple versions of a row (old and new) can co-exist in the Bonsai Tree. Both the old and new data are used for concurrency control and for ensuring consistent read levels for all transactions without any locking overhead. With the Bonsai Tree, the effort needed to validate transactions is significantly reduced.

When a transaction is started, it is given a transaction start number (TSN). The TSN is used as the read level of the transaction; all key values inserted later into the database from other connections are not visible to searches. This offers consistent index read levels that appear as if the read operation was performed atomically at the time the transaction was started. This guarantees read operations are presented with consistent view of the data without the need for locks.

Later the new committed data is merged to another indexing system, known as the storage server, and removed from the Bonsai Tree. The presorted key values are merged as a background operation concurrently with normal database operations. This offers significant I/O optimization and load balancing. During the merge, the deleted key values are physically removed.

Storage Server

The storage server uses a B-tree variation to store all permanent indices in the database file. It stores both secondary keys and the primary keys. Data rows are stored as the primary key values actually containing all the columns of the rows. There is no separate storage method for data rows, except for BLOBs and other long column values.

Indices are separated from each other by a system-defined index-identification inserted in front of every key value. This mechanism divides the index tree into several logical index subtrees, where the key values of one index are clustered close to each other. For details on data clustering, read *“Data clustering”* on page 4-9.

Index Compression

Two methods are used to store key values in the Bonsai Tree and the storage server. First, only the information that differentiates the key value from the previous key value is saved. The key values are said to be prefix-compressed. Second, in the higher levels of the index tree, the key value borders are truncated from the end; that is, they are suffix-compressed.

SOLID SQL Optimizer

The SOLID SQL Optimizer, a cost-based optimizer, ensures that the execution of SQL statements is done efficiently. It uses the same techniques as a rules-based optimizer, relying on a preprogrammed set of rules in determining the shortest path to the results. For example, the SQL Optimizer considers whether or not an index exists, if it’s unique, and over single or composite table columns. However, unlike a rule-based optimizer, its cost-based feature can adapt to the actual contents of the database; for example, the number of rows and the value distribution of individual columns.

SOLID *SynchroNet* maintains the statistical information about the actual data automatically, ensuring optimal performance. Even when the amount and content of data changes, the optimizer can still determine the most effective route to the data.

Query Processing

Query processing is performed in small steps to ensure that one time-consuming operation does not block another application's request. A query is processed in a sequence containing the following phases.

Syntax analysis

A SQL query is analyzed and either a parse tree for the syntax or a syntax error is produced. When a statement is parsed, the information necessary for its execution is loaded into the statement cache. Statements are executed repeatedly without re-optimization, as long as its execution information remains in the statement cache

Creating the execution graph

The execution graph, with the following features, is created from the query parse tree.

- Complex statements are written to a uniform and more simple form
- If better performance will be realized, OR criteria is converted to UNION clauses
- Intelligent join constraint transfer is performed to produce intermediate join results that reduce the join process execution time.

Read *"The EXPLAIN PLAN Statement"* on page 9-2 for details on each operation or *unit* in the execution plan

Processing the execution graph

Processing of the execution graph is performed in three consecutive phases:

- Type-evaluation phase

  The data types of the columns of the result set are derived from the underlying table and view definitions

- Estimate-evaluation phase

  The cost of retrieving first rows and also entire result sets is evaluated, and an appropriate search strategy is selected dynamically based on the bound parameter values.

The SQL Optimizer bases cost estimates on automatically maintained information of key value distribution, table sizes, and other dynamic statistical data. No manual updates to the index histograms or any other estimation information is required.

- Row-retrieval phase

  The result rows of the query are retrieved and returned to the client application

SOLID Network Services

SOLID Network Services are based on the remote procedure call (RPC) paradigm, which makes the communication interface simple to use. When a client sends a request to the server, it resembles calling a local function. The Network Services invisibly route the request and its parameters to the server, where the actual service function is called by the RPC Server. When the service function completes, the return parameters are sent back to the calling application.

In a distributed system, several applications may request a server to perform multiple operations concurrently. For maximum parallelism, SOLID Network Services use the operating system threads when available to offer a seamless multi-user support. On non-thread operating systems, the Network Services extensively use asynchronous operations for the best possible performance.

Communication Session Layer

SOLID communication protocol DLLs (or static libraries) offer a standard internal interface to each protocol. The protocol interface is BSD-socket like, containing methods, such as listen, accept, select, connect, read, write, disconnect, and control.

The lowest part of the communication session layer works as a wrapper that takes care of choosing the correct protocol DLL or library that relates with the given address information. After this point, the actual protocol information of the session is hidden.

The communication layer detects new communication messages (requests) from applications by selecting sessions that contain unread data. Where available, the process uses threads; elsewhere system-specific methods are used. The sessions that contain requests are written to a message queue (FIFO), where the RPC layer can find them in the order they arrived. If SOLID *SynchroNet* is listening to many protocols simultaneously, the requests arriving through different protocols are all written to the same message queue.

RPC Session Layer

The RPC session layer contains services for typed parameter passing. Both application and server use these services for reading and writing all kinds of data, from standard C-types, up to the most complex internal data types.

For a server, the RPC session layer offers utilities for declaring the RPC service set that the server recognizes. The services are implemented as callback functions. The RPC level identifies the requested service, and the RPC server level is able to call the correct callback routine when requested.

For a client, the RPC session level also allows many simultaneously pending RPC requests. A client can in a single operation wait for any pending request to complete.

Multithread Processing

SOLID *SynchroNet's* multithread architecture provides an efficient way of sharing the processor within an application. A thread is a dispatchable piece of code that merely owns a stack, registers, and its priority. It shares everything else with all other active threads in a process. Creating a thread requires much less system overhead than creating a process, which consists of code, data, an other resources such as open files and open queues.

Threads are loaded into memory as part of the calling program; no disk access is therefore necessary when a thread is invoked by another thread. Threads can communicate using global variables, events, and semaphores.

If the operating system supports symmetric multi-threading between different processors, SOLID *SynchroNet* automatically takes advantage of the multiple processors.

Types of Threads

The SOLID *SynchroNet* threading system consists of general purpose threads and dedicated thread.

General Purpose Threads

General purpose threads execute tasks from the server's tasking system. They execute such tasks as serving user requests, making backups, making checkpoints, making timed commands, and index merging.

General purpose threads take a task from the tasking system, execute the task step to completion and switch to another task from the tasking system. The tasking system works in a round-robin fashion distributing the client operations evenly between different threads.

Dedicated Threads

Dedicated threads are dedicated to a specific operation. The following dedicated threads may exist in the server:

- I/O manager thread

This thread is used for intelligent disk I/O optimization and load balancing. All I/O requests go through the I/O manager which determines whether to pass I/O requests to the cache or to schedule it among other I/O requests. I/O requests are ordered by their logical file address. The ordering optimizes the file I/O since the file addresses accessed on the disk are in close range, reducing the disk read head movement.

- Communication read threads

  Applications always connect to a listener session that is running in the selector thread. After the connection is established, a dedicated read thread can be created for each client.

- One communication select thread per protocol (known as the selector thread)

  There is usually one communication selector thread per protocol. Each running sector thread writes incoming requests into a common message queue.

- Communication server thread (also known as the RPC server main thread)

  This thread reads requests from the common message queue and serves applications by calling the requested service functions.

2

SOLID *SynchroNet* Data Synchronization Architecture

This chapter introduces you to SOLID *SynchroNet*, the solution to managing distributed data in today's smart networks. The SOLID *SynchroNet* solution provides fast, reliable access to data anywhere, at anytime.

About SOLID *SynchroNet*

SOLID *SynchroNet* is a new approach to data synchronization. It applies concepts that are fundamentally different in comparison to the "traditional" data replication solutions.

The fundamental difference is an architecture that uses extensions to the relational data management functions to enable building data synchronization functionality inside the application. This new model recognizes what traditional replication solutions do not, that data synchronization contains aspects that are application specific. For proper functionality, synchronization must be partly implemented inside 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 multidatabase system.

With the SOLID *SynchroNet* solution, application developers now have a set of data management functions that enable them to include the data synchronization into the application with minimal effort.

SOLID *SynchroNet* Features

SOLID *SynchroNet* delivers the following features:

- System-wide information sharing

  With SOLID *SynchroNet*, a local copy of data is available to distributed nodes of the system. There is no need to provide users with on-line access to central data management resources. In addition, each replica database of a *SynchroNet* 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 a transaction processing application.

- Data integrity

  In a multidatabase system, where updates can occur in multiple databases, maintaining the data integrity poses challenges. SOLID *SynchroNet's* own patent-pending transaction management architecture addresses data integrity issues by allowing the transactions's application developer to build transaction validation capabilities. This ensures that information is accurate based on the application's own rules and logic.

- High performance and flexibility

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

SOLID *SynchroNet* Architecture Concepts

The SOLID *SynchroNet* 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 different versions of the same data item can co-exist.

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 the hierarchy.

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

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

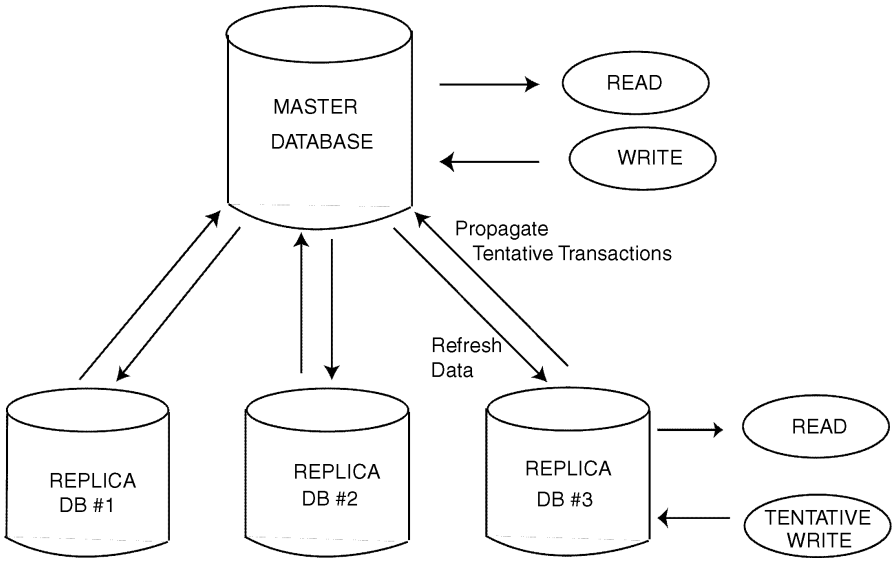
- For each data item, there is one *master version* considered the official version or superset of the data. Other copies of the item are *tentative versions*, that is, *replicas.*
- A replica can act as a master to other replicas below it. A replica that is also a master contains a subset of the data 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 1-1. Note that transactions are always sent to the master databases where they are committed. The changed master data is then sent to the replicas.

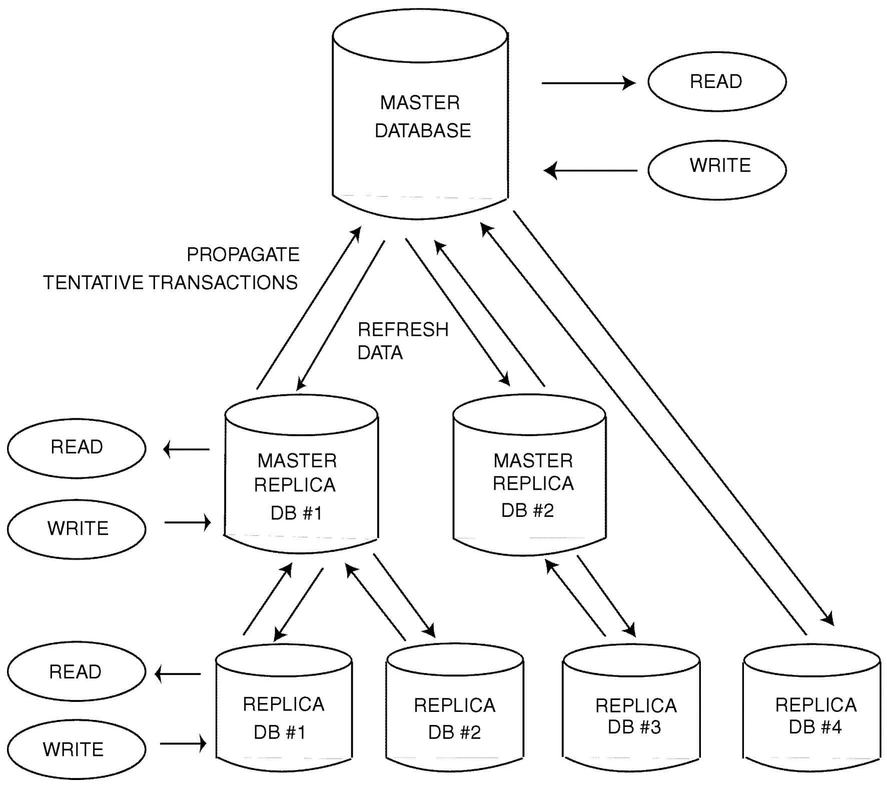
Figure 2–1 Two-tier Data Redundancy Architecture



Advanced Multi-Tier Architecture

A more advanced implementation of this model enables information flow in multiple tiers through replicas that are also masters to some other replica below 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.

Figure 2–2 Multi-tier Data Redundancy Architecture



SOLID *SynchroNet* Transaction model

In both two-tiered and advanced 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 into it, sufficient validation logic to ensure the integrity of the master database.

To address the data integrity issues of a synchronized multidatabase system, SOLID *SynchroNet* 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 the section titled *"SynchroNet's Intelligent Transaction" on page 2-10.*

Note that transactions committed in a replica database are propagated only to its master database. These transactions are not propagated to other replicas as such. Instead, the other replicas can request changed data from a master database by subscribing to one or multiple publications.

Multi-master synchronization model

In today's distributed, networked environments, a system consists of multiple applications. These applications may have their own databases. SOLID *SynchroNet* allows a database to contain data from multiple master databases. For instance, a local database can contain a replica from a billing system and network configuration system. Both two-tier and multi-tier architectures are scalable to accommodate multiple master databases. The advanced multi-tier shown in Figure 2–3 makes use of a multi-master synchronization.

Figure 2–3 Multi-Master Model

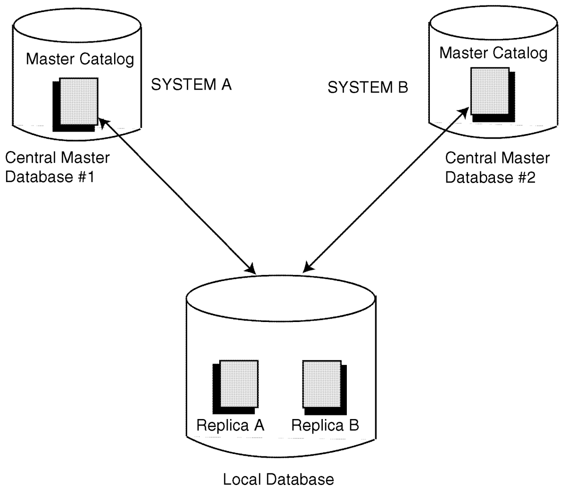


Figure 3-1 further illustrates the concept of multi-master synchronization on a table level. In this figure, note the following:

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

Multi-master Features

SOLID *SynchroNet's* multi-master model:

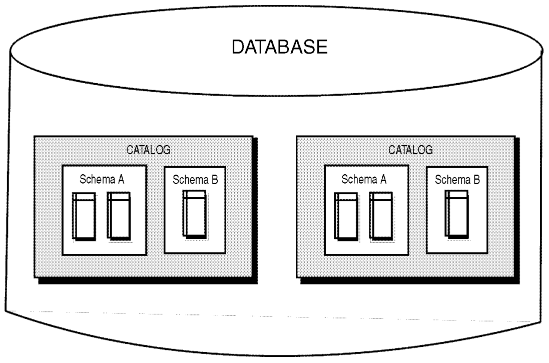
- Allows replicas (through registration) to synchronize data with multiple SOLID *SynchroNet* masters.
- Keeps replica data from different masters separate using catalogs.
- Keeps local data separate from shared data.

Each of these features is described in the following sections.

Managing replica data in a multi-master environment

To distinguish data from different master databases in a replica, SOLID *SynchroNet* uses database catalogs. As shown in Figure 2–4, a database can contain one or more catalogs and each catalog can contain one or more schemas.

Figure 2–4 Database, Catalog, and Schema



A catalog logically partitions a database so that data is organized in ways that meet business or application requirements. 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 a 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 in DML statements as:

catalog_name.schema_name.database_object or catalog_name.user_id.database_object

Data from a single catalog on the replica database is mapped into a specific catalog on the master database. In addition, each catalog on a replica can contain local data, which belongs to the local database only and never gets replicated 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 the replica only are never changed during synchronization.

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

For details on implementing catalogs, read *"Guidelines for multi-master topology"* on page 11-6.

Schemas

Schema names of a master and replica must be identical. This is in keeping 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 for databases are created with the CREATE SCHEMA statement. For details on managing database objects with schemas, read *"Managing Database Objects"* on page 4-14.

Database Objects

In a multi-master environment, catalogs keep data from different masters separated. Object name conflicts between masters 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, SOLID *SynchroNet* 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.

Transactions

In a multi-master environment, a transaction cannot span between two different masters. For example:

```
SAVE update table A in MASTER A

SAVE update table B in Master B

Commit work
```

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

SOLID *SynchroNet* allows local data modifications in the same transaction within a catalog. The SET CATALOG command explicitly defines the master used for all SynchroNet-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.

SOLID *SynchroNet* Architecture Components

SOLID *SynchroNet* 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 SOLID *SynchroNet* 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.
- *Transactions* based on SOLID *Intelligent Transaction* technology 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.

Master and Replica Databases

The master database of the distributed system is the storage of 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 is the storage of local data and saved transactions. All replica data, or a suitable part of it, can be refreshed from the master database whenever needed by subscribing to one or multiple 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*"Multi-master synchronization model"* on page 2-5.

Publications and Subscriptions

The synchronization architecture of a multidatabase 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 request publication data from the master. A publication is registered in a replica. Users can subscribe only to those publications that are registered. In this way publication parameters are validated, preventing users from accidentally subscribing to unwanted or non-existing publications or making ad hoc subscriptions.

The initial subscription 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 *"Creating Publications"* on page 12-22.

Subsequent subscriptions for the same publication contain only the data that has been changed since the prior subscription. This is known as an *incremental publication*. 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 *SynchroNet* commands, extensions to SOLID SQL. See Chapter 12,"Implementing a SOLID SynchroNet Application"for details on the CREATE PUBLICATION command.

SynchroNet's Intelligent Transaction

The main concept in understanding SOLID *SynchroNet* is that the data of all replica databases is unofficial and therefore all modifications done to it are tentative. These modifications become official only when they are 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 multidatabase 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 subscription. 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.

SOLID *SynchroNet's Intelligent Transaction* model provides a framework for implementing transactions with long life spans. Transaction propagation in *SynchroNet's* architecture is based on SOLID *Intelligent Transaction* technology.

*Intelligent Transactio*n 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 multidatabase 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 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 in order to be a valid one. 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 subscribed back to the replica to notify the replica users that the transaction has failed.

Multidatabase Systems versus Centralized Systems

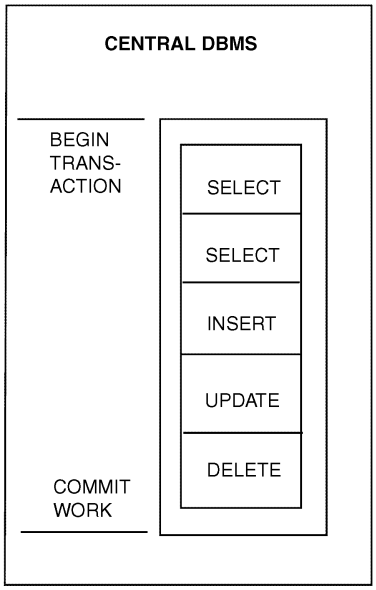
In a traditional client/server system that uses a central database, the validation logic of each transaction is typically in the client application or in the application server's services. For instance, in an Order Entry application, the application logic must check prior to committing the transaction 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 to 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.

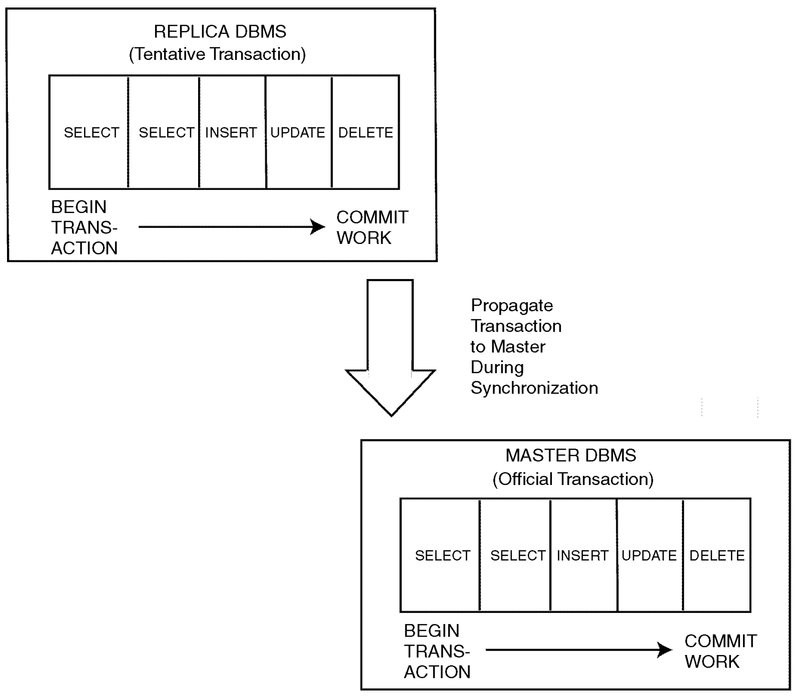
Figure 2–5 A transaction in a central database



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 multidatabase 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 2–6 A transaction in a synchronized database

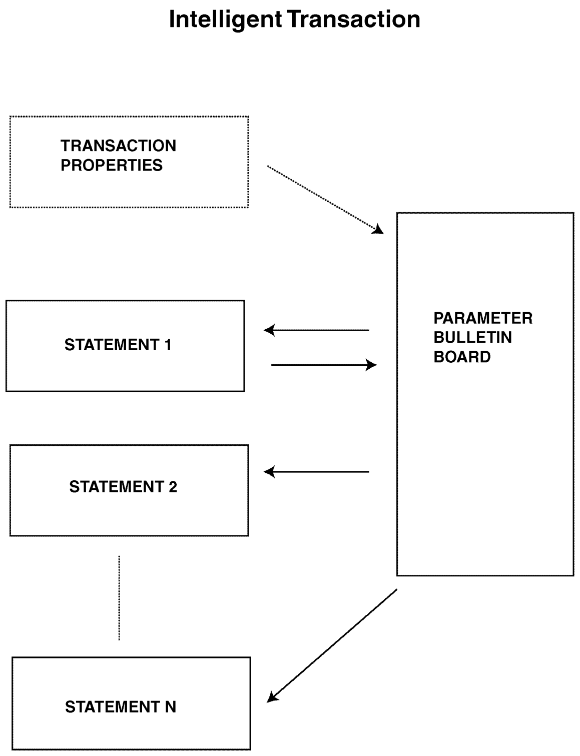


Intelligent Transaction in the Multidatabase System

In a multidatabase 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 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 well known “ACID” transaction model 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 2–7 The Structure of Intelligent Transaction



How *Intelligent Transaction* Works

With SOLID *Intelligent Transaction*, 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 contents of 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.

An Example:

This example applies 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 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 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 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.

Asynchronous Store and Forward Messaging

Communication between the SOLID *SynchroNet* 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 subscribe 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 SOLID *SynchroNet* architecture guarantee that no message is deleted from the sending node before the entire message has arrived at the receiving node.

Part II

SOLID *SynchroNet* Engine Administration

Part III shows you how to administer, maintain, configure and tune SOLID *SynchroNet*. It includes basic server administration. Part III contains the following topics:

3

Administering SOLID *SynchroNet*

This chapter describes how to maintain your *SynchroNet* installation. The administration tasks covered in this chapter are:

- Performing basic *SynchroNet* operations
- Managing synchronization errors
- Managing access rights
- Managing *SynchroNet* tables and databases
- Modifying publications and subscriptions
- Managing data with bookmarks
- Modifying SQL procedures of *Intelligent Transaction*

What You Should Know

This section describes what you need to know about SOLID *SynchroNet* before you begin administration and maintenance.

Installing SOLID *SynchroNet*

If you have not yet installed SOLID *SynchroNet*, refer to the **ReadMe** notice delivered with the software or included on the SOLID Web site at:

`http://www.solidtech.com/`

The **ReadMe** contains a detailed description of the installation.

Using SOLID Databases 2.20 or Prior

Beginning with SOLID version 2.3 to the current version, the default collation sequence is set to the standard Latin-1. SOLID databases that were created with version 2.20 or prior do not match the Latin-1 collation sequence. To convert the data to Latin 1 in a version 2.20 database, you must export the database from its tables, extract data definitions, and load the tables to the new database. Read for details *"Tools Sample: Reloading a Database"* on page 5-25.

Special Roles for Database Administration

SOLID *SynchroNet* has the following roles for administration and maintenance:

- SYS_ADMIN_ROLE

  This is the Database Administrator role and has privileges to all tables, indexes, and users, as well as the right to use SOLID *DBConsole* and SOLID *Remote Control* (teletype). This is also the role of the creator of the database.

- SYS_CONSOLE_ROLE

  This role has the right to use SOLID *Remote Control*, but has no other administration privileges.

- SYS_SYNC_ADMIN_ROLE

  This is an administration role for performing *SynchroNet* administrative operations, such as deleting messages. Anyone with this access has all synchronization roles granted automatically. This role automatically includes the SYS_SYNC_REGISTER_ROLE.

- SYS_SYNC_REGISTER_ROLE

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

You define these roles using the GRANT ROLE statement. For details, read *"Managing User Privileges and Roles"* on page 4-2.

Automated and Manual Administration

SynchroNet is designed for continuous, unattended operation and ease of deployment. It requires minimal maintenance. Administrative operations, including backups can be performed programmatically using SQL extensions, which can run automatically or at an administrator's request.

Sometimes it makes sense to administer systems manually. This chapter refers you to the tools and methods available for performing manual administration. You can issue administrative commands equivalent to SOLID SQL's own ADMIN COMMANDs in SOLID *DBConsole* or in SOLID *Remote Control* (teletype). See *"Administrative Commands"* on page 5-6 for a commands list.

Note that with SOLID *DBConsole*'s Administrative window, you can perform most of the SQL ADMIN COMMAND tasks that you execute on the command line with easy-to-use dialog boxes. For a description of SOLID *DBConsole*, read *"SOLID DBConsole"* on page 5-2. SOLID *SQL Editor* (teletype) also lets you enter administrative commands using the full SQL ADMIN COMMANDs syntax. See *"ADMIN COMMAND"* on page E-1 for a commands list.

Starting SOLID *SynchroNet*

When SOLID *SynchroNet* is started, it checks if a database already exists in the SOLID directory, that is, the directory where you installed SOLID executables. If a database file is found, SOLID *SynchroNet* will automatically open that database. If not, which is the case when you start the server for the first time, a new database will be created.

Operating System	To Start the Server...
UNIX	Enter the command `solid` at the command prompt. When you start the server for the first time, enter the command `solid -f` at the command prompt to force the server to run in the foreground.
Novell Netware	Enter the command `load solid.nlm` at the command prompt.
Open VMS	Enter the command `run solid` at the command prompt.
Windows	Click the icon labeled `SOLID SynchroNet` in the SOLID *SynchroNet* program group.

Creating a New Database

If a database does not exist, SOLID *SynchroNet* will at start up automatically create a new database. In the Windows environment, creating the database begins with a dialog prompting for the database administrator's username, password, and a name for the default database catalog. For details, read *"Managing Database Objects"* on page 4-14.

In other environments, if you do not have an existing database, the following message appears:

```
Database does not exist. Do you want to create a new database (y/n)?
```

Answer y(es), and SOLID *SynchroNet* prompts you for the database administrator's username, password, and a name for the default database catalog.

The username requires at least two characters; the password at least three. For both the username and password, the maximum number of characters is 80. Use lower case letters from a to z, upper case letters from A to Z and the underscore character '_', and numbers from 0 to 9.

▶ **Note**

You must remember your username and password to be able to connect to SOLID *SynchroNet*. There are no default usernames; the username you enter when creating the database is the only username available for connecting to the new database.

After accepting the database administrator's username and password, SOLID *SynchroNet* creates the new database.

By default the database will be created as one file (`solid.db`) in the SOLID working directory, where the current working directory is located. An empty database containing only the system tables and views uses approximately 450 KB of disk space. The time it takes to create the database depends on the hardware platform you are using.

After the database has been created, SOLID *SynchroNet* starts listening to the network for client connection requests. In the Windows environment, a SOLID *SynchroNet* icon appears, but in most environments SOLID *SynchroNet* runs invisibly in the background as a daemon process.

Windows only If in the Windows environment you double-click the icon of a running server, nothing will happen. SOLID *SynchroNet* is a background process that only reacts to messages from clients through the communication interface.

About SOLID Databases

This section describes SOLID database structure and ways you can specify different values when creating SOLID databases.

Setting Database Size and location

By default, SOLID databases set a block size for the database file as 8192 bytes. SOLID SynchroNet uses a multiple of 2 KB. The minimum block size is 2 KB and the maximum is 32 KB. The maximum size of the database is 64 TB.

If you want SOLID *SynchroNet* to create a database with a different block size, you have to set a new constant value before creating a new database. If you have an existing database, be sure to move the old database and log files.

To modify the constant value for the new database, go to the SOLID directory and add the following lines in the `solid.ini` file, providing the size in bytes:

```
[Indexfile]
Blocksize=size_in_bytes
```

After you save the file and start SOLID *SynchroNet*, it creates a new database with the new constant values from the `solid.ini` file.

Similarly, you can also modify the The `FileSpec` parameter to define the following:

- location of the database file to change the default of `solid.db` in the SOLID directory
- maximum size (in bytes) the database can reach to change the default of value of 2147483647, which equals 2GB.

You can also use the FileSpec parameter to divide the database file into multiple files and onto multiple disks.

For details on configuration with the FileSpec parameter, read *"Managing Database Files and Caching (IndexFile section)"* on page 7-2.

Defining Database Objects

SOLID database objects include catalogs, schemas, tables, views, indexes, stored procedures, triggers, and sequences. By default, database object names are qualified with the object owner's user id and a system catalog name that you specify when creating a database for the first time or converting an old database to a new format. You can also specify that database objects be qualified by a schema name. For details, read *"Managing Database Objects"* on page 4-14.

SOLID *SynchroNet* supports a practically unlimited number of tables, rows, and indexes. Character strings and binary data are stored in variable length format. This feature saves disk space because no extra data is stored in the database. It also makes programming easier on developers since the length of strings or binary fields do not have to be fixed. The maximum size for a single attribute is 2GB.

By configuring the `MaxBlobExpression` parameter, you can set the maximum size of LONG VARCHAR columns in KBs that are used in string functions. By default, the size is 65KB. When BLOBs (Binary Large Objects), such as objects, images, video, graphics, or binary fields, are larger than the configured limit, SOLID *SynchroNet* automatically detects this and stores the objects to a a special file area that has optimized block sizes for large files. No administrative action is required.

Connecting to SOLID *SynchroNet*

After starting SOLID *SynchroNet*, you can test the configuration by connecting to the server from your workstation using SOLID *DBConsole* or the SOLID teletype tools, *SQL Editor* or *Remote Control*. Read Chapter 5, "Using SOLID Data Management Tools" for details on these utilities which are part of the SOLID Data Management tools.

▶ **Note**

You need to have SYS_ADMIN_ROLE or SYS_CONSOLE_ROLE privilege to be able to connect to a server using SOLID *DBConsole*. For details on creating these roles, read *"Managing User Privileges and Roles"* on page 4-2.

To connect to SOLID *SynchroNet*:

1. View the `solmsg.out` file in your database directory for valid network names that you can use to connect to SOLID *SynchroNet*.

   The following messages indicate what names you can use.

   ```
   Listening of 'ShMem Solid' started.
   Listening of 'TCP/IP 1313' started.
   ```

2. Start one of the following applications and give the network name of the server as a command line parameter:

Tool	Command
SOLID *DBConsole*	java DBConsole **-D***databasename* **-U***url* **-u***userid* **-p***password* For example: `java DBConsole -Dsolid -Ujdbc:solid://localhost:1313 -udba -pdba` Alternatively, you can start *DBConsole* without any command line option. You are then prompted for the database connection information.

Tool	Command
SOLID *Remote Control* (teletype)	solcon "*networkname*" For example: `solcon "tcp hobbes 1313"` When prompted, enter the database administrator's user name and password.
SOLID *SQL Editor* (teletype)	solsql "*networkname*" For example: `solsql "tcp hobbes 1313"` When prompted, enter the database administrator's user name and password.

After a while you will see a message indicating that a connection to the server has been established.

Viewing the SOLID *SynchroNet* Message Log

Ensure the database started without errors by checking the message log `solmsg.out`, located in the SOLID directory. You can view this file in SOLID *DBConsole*'s Messages page from the Administration window.

SOLID *SynchroNet* maintains the following message log files:

- `The solmsg.out` log file contains normal informational events, such as connects, disconnects, checkpoints, backups, etc. If an internal error occurs, the error is written to the `solmsg.out` file.
- If the error is fatal, the `solerror.out` file contains more detail about the error. Internal errors are not documented.

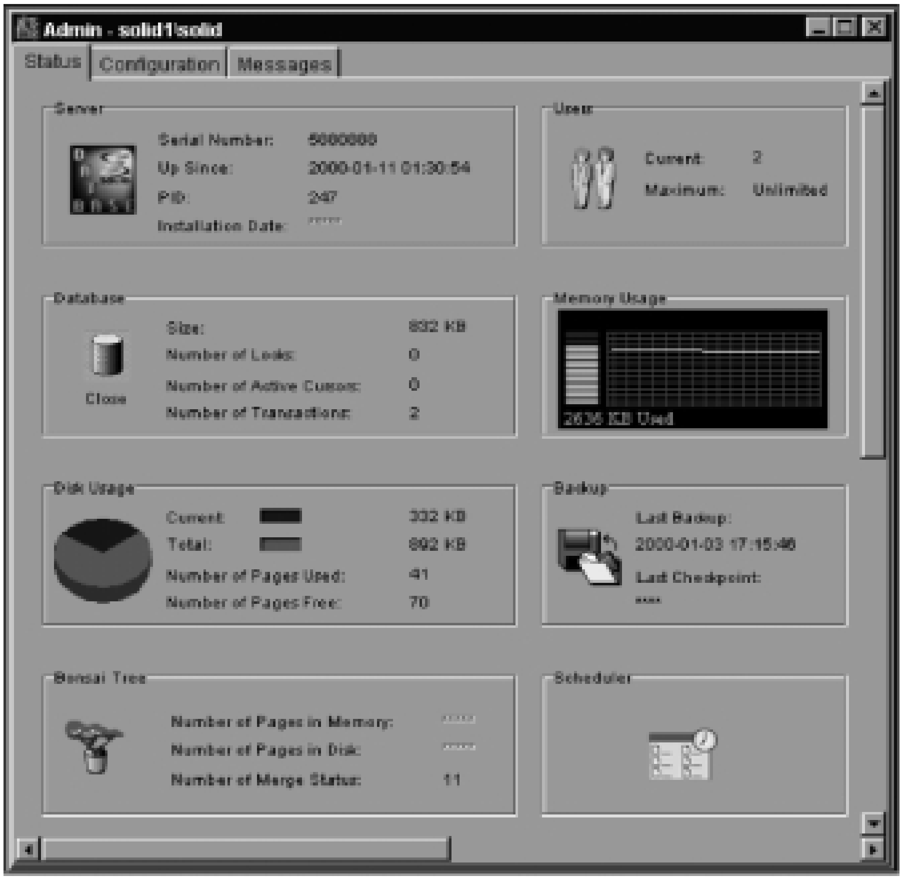
For troubleshooting purposes, SOLID *SynchroNet* can also produce optional trace files that contain information for diagnostics. Monitoring the trace files is not necessary for everyday operation of the server. The trace files are primarily needed for troubleshooting of exceptional events. Refer to Chapter 9, "Diagnostics and Troubleshooting" for more details on SOLID diagnostics.

Monitoring SOLID *SynchroNet*

The following sections describe the methods used for querying the status of a SOLID database.

Checking overall database status

You can select the **Status** option from the SOLID *DBConsole* Administration window. The status page displays the following information as shown below.



Server

Serial Number: 5000000
Up Since: 2000-01-11 01:30:54
PID: 247
Installation Date: ******

Users

Current: 2
Maximum: Unlimited

Database

Size: 892 KB
Number of Locks: 0
Number of Active Cursors: 0
Number of Transactions: 2

Memory Usage

2636 KB Used

Disk Usage

Current: 332 KB
Total: 892 KB
Number of Pages Used: 41
Number of Pages Free: 70

Backup

Last Backup: 2000-01-03 17:15:46
Last Checkpoint: ****

Bonsai Tree

Number of Pages in Memory: ******
Number of Pages in Disk: ******
Number of Merge Status: 11

Scheduler

You can also issue the following command in SOLID *DBConsole* or SOLID *Remote Control* (teletype):

```
status
```

or in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND 'status';
```

The command provides the following statistics information:

```
RC TEXT
-- ----
0 SOLID SynchroNet started at Thu May 27 16:11:20 1999
0 Current directory is D:\solid
0 Using configuration file D:\solid\solid.ini
0 Memory statistics:
0  1400 kilobytes
0 Transaction count statistics:
0      Commit Abort Rollback    Total Read-only  Trxbuf  Active Validate
0      27     0       18       45        45       0       1        0
0 Cache count statistics:
0     Hit rate      Find      Read     Write
0     93.5          445       29        0
0 Database statistics:
0     Index writes           0  After last merge      0
0     Log writes             0  After last cp         0
0     Active searches        0  Average               1
0     Database size      1232 kilobytes
0     Log size            274 kilobytes
0 User count statistics:
0     Current Maximum Total
0           1       1     2
```

Following is a description of the result set fields:

- Memory statistics show the amount of memory SOLID has allocated from the operating system. This number does not include the size of the executable itself.
- Transaction count statistics show the number of different transaction operations since startup.
- Cache count statistics show cache hit rate and number of cache operations since startup. Cache hit rate typically is above 95 per cent.

- Database statistics shows a number of the most important database operations since startup. "Index writes after last merge" is an important figure here. It reveals the size of the multi-versioning storage tree of SOLID, known as the "Bonsai Tree." The smaller this value is, the better the server performance. A large value indicates that there is a long-running transaction active in the engine.
- User count statistics shows current and maximum number of concurrent users.

Obtaining Currently Connected Users

To obtain a list of currently connected users:

1. Select the **Status** option from the SOLID *DBConsole* Administration window or menu.
2. On the Status page, click the **Users** icon.

A Users dialog box displays each user’s name, user id, type, machine id, and login time.

You can also obtain a listing of connected users by entering command `userlist` in SOLID *DBConsole* or SOLID *Remote Control* (teletype) or enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND 'userlist' ;
```

The command provides the following kind of result set:

```
RC TEXT
-- ----
0 User name:     User id: Type:  Machine id:      Login time:
0 DBA            1        SQL    Local            27.05 16:13:22
```

Throwing out a connected SynchroNet user

To disconnect a single user from the server, you can:

- Select the **Status** option from the SOLID *DBConsole* Administration window or menu, click the **Users** icon, and drop a selected user from the Users dialog box.

  or

- Enter command `throwout` *`user_id`* in SOLID *DBConsole* or SOLID *Remote Control* (teletype) or enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

  ```
  ADMIN COMMAND 'throwout user_id'
  ```

Querying the status of the last backups

To obtain a status of the most recently run backup:, you can:

- Select the **Status** option from the SOLID *DBConsole* Administration window, and click the **Backup** icon to view the backup status on the Backup dialog box.

  or

- Enter command `status backup` in SOLID *DBConsole* or SOLID *Remote Control* (teletype) or enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

  ```
  ADMIN COMMAND 'status backup'
  ```

  If the last backup is successful, the result set looks as follows:

  ```
  RC TEXT
  -- ----
  0  SUCCESS
  ```

  If the latest backup has failed, then the RC column returns an error code. Return code 14003 with text "ACTIVE" means that the backup is currently running.

Detailed DBMS monitoring and troubleshooting

Besides checking the SOLID *DBConsole* Status page, you can also take a snapshot that provides additional information on *Embedded Engine* performance. Enter the `perfmon` command in SOLID *DBConsole* or SOLID *Remote Control* (teletype) or enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND 'perfmon'
```

The command returns a result set where each column represents a snapshot of the performance information that reflects the most recent few minutes.

The first column shows average performance information from a period of 55 seconds. The "Total" column shows average information since *SynchroNet* was started. Most numbers are events/second. Those numbers that cannot be expressed as events/second (for example, database size) are expressed as absolute values.

The output information is categorized as follows:

- File operations
- Cache operations
- RPC and communications operations
- SQL operations
- SA (table-level db-operations) operations
- Transaction operations
- Index write (that is, database file write) operations

- Miscellaneous operations

Producing a status report

To create a report about the current status of SOLID *Embedded Engine*, enter the command `report` *`report_filename`* in SOLID *DBConsole* or SOLID *Remote Control* (teletype) or enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND 'report report_filename'
```

This report is primarily meant for SOLID internal use only because it contains information that requires very detailed understanding about the internals of SOLID *SynchroNet*. End users sometimes are requested to produce the report for troubleshooting purposes.

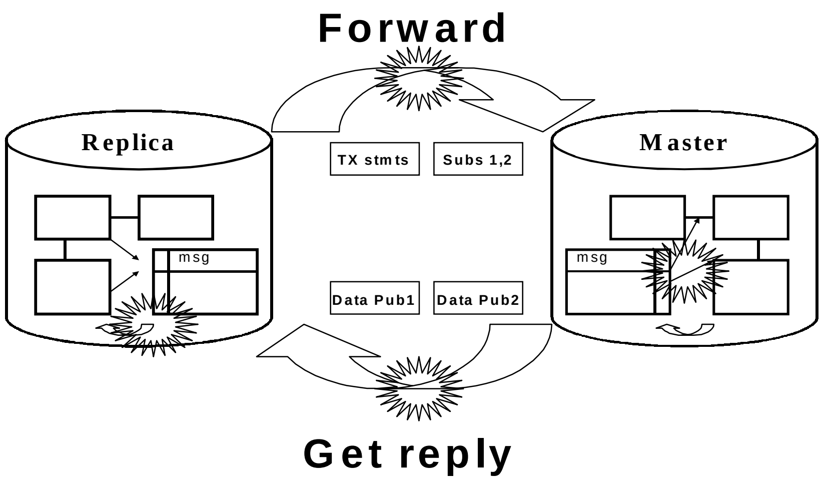
Monitoring the Status of Synchronization Messages

Because synchronization is implemented using synchronization messages, you can monitor the status of the 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 3–1 illustrates store and forward messaging and shows the points in the process where messaging errors can occur.

Figure 3–1 Error points in synchronization messaging



As Figure 3–1 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

See *Appendix A, “System Tables for Data Synchronization”* for a detailed description of these tables. Refer to the following section for ways to resolve errors that you see in the tables.

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 master with the following SQL statement:

```
SELECT MSG_NAME
FROM SYS_SYNC_REPLICA_MSGINFO
WHERE STATE = 22
```

Such messages can be re-sent to the master database with the following command in the replica database:

```
MESSAGE message_name FORWARD;
```

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
- subscription to a publication fails
- a reply message fails due to a networking error

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 can be 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 information about all messages that currently reside in the master database. Those messages, whose execution has failed because of an error, 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 with the following kind of query:

```
SELECT   rep.name replica_name, msg.msg_name, stmt.string stmt_string,
         msg.error_code, msg.error_text
FROM     sys_sync_master_msginfo msg, sys_sync_received_stmts stmt,
         sys_sync_replicas rep
WHERE    msg.msg_id = stmt.msg AND
         msg.trx_id = stmt.trx_id AND
         msg.stmt_id = stmt.id AND
         msg.replica_id = rep.id AND
         msg.state = 1 ;
```

In this case, the proper way to recover from the error is to fix the error in the master database. For example, the reason for the error could be a unique key constraint violation. To fix this error, the existing data of the master database must be modified to allow the new row to be inserted. 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 after the deletion is committed, use the following statement:

```
MESSAGE msgname FROM REPLICA replicaname EXECUTE
```

In general, transactions should be written to prevent 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 *"SYS_TRAN_MAXRETRY"* on page 13-54.

Error Handling in Subscriptions

Unlike transactions, an error in a subscription execution 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.

The most typical error is that the version of the publication has changed in the master database. In this case, 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 always brings the full publication to the replica.

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.

```
MESSAGE message_name GET REPLY TIMEOUT timeout_in_seconds
```

You can query messages, whose replies have not been successfully received by the replica, with the following SQL statement:

```
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*

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 conflict. Records applicable to a transaction may be locked so that an operation is unable to be performed in the replica. In this case, the message remains in the replica database. For example, when other transactions, including user transactions and *SynchroNet* transactions are updating a table, a concurrent subscription operation may fail. In this case, the subscription 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 in subscriptions. For more information, read*"Handling Concurrency Conflict in Synchronized Tables"* on page 11-12.

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 command 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;
```

Shutting Down SOLID *SynchroNet*

You can shut down SOLID *SynchroNet* in these ways:

- Programmatically from an application such as SOLID *DBConsole*, SOLID *Remote Control*, or SOLID *SQL Editor* (teletype)*.

  To do this, perform the following steps:

  1. To prevent new connections to *SynchroNet*, close the database(s) by entering the following command:

     ```
     close
     ```

  2. Exit all users of *SynchroNet* by entering the following command:

     ```
     throwout all
     ```

  3. Stop *SynchroNet* by entering the following command:

     ```
     shutdown
     ```

     * Note that when using SOLID *SQL Editor* (teletype) for steps 1-3, you enter the full SQL Syntax, ADMIN COMMAND ’*command_name*’ (for example, ADMIN COMMAND ’`close`’).

- Clicking the server icon and selecting **Close** from the menu appearing in the Windows environment.
- Remotely, using the command 'net stop' through the Windows NT system services. Note that you may also start up SOLID *SynchroNet* remotely, using the 'net start' command.

All the shutdown mechanisms will start the same routine, which writes all buffered data to the database file, frees cache memory, and finally terminates the server program. Shutting down a server may take a while since the server must write all buffered data from main memory to the disk.

Performing Backup and Recovery

This section describes how to back up your databases and recover from system failure.

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 continue working with the backup database.

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 *"Entering Timed Commands"* in this chapter for details.
- Select the **Status** option from the SOLID *DBConsole* Administration window, and click the **Backup** icon to initiate the backup from the Backup dialog box.
- Issue the following command in SOLID *DBConsole* or SOLID *Remote Control* (teletype):

  ```
  backup
  ```
- Issue the following command in 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.

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 Solid 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.

Backing up and Restoring the Master and Replica Databases

The normal disaster prevention and recovery tasks you use on a non-synchronized SOLID database also apply to synchronized databases.

If the master or replica database is not shut down properly, SOLID *SynchroNet* uses a multi-versioning technique, automatically using the log files to perform a roll-forward recovery during the next start up. No administrative procedures are required to start the recovery. However, 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 *"Monitoring the Status of Synchronization Messages"* on page 3-12. For instructions on correcting stopped synchronization messages, see *"Managing Synchronization Errors"* on page 3-14.

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 *SynchroNet* system. Upon recovery, *SynchroNet* 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.

► Notes

1. You can query programmatically the status of the most recently started backup in SOLID *DBConsole* or SOLID *Remote Control* (teletype) by using the command `status backup`. To query the list of all completed backups and their success status, use the command `backuplist`. To use these same commands in SOLID *SQL Editor*, enter the SOLID SQL syntax (for example, `ADMIN COMMAND 'backuplist'`).

   You can also query backup status in SOLID *DBConsole* 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.

2. 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.

3. 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 displayed on the SOLID *DBConsole* 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).

Backup Guidelines

Using backups in the *SynchroNet* system is a straightforward and powerful way to ensure the high level of data availability and security. Therefore, backups should be available from all critical databases of the system.

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 *SynchroNet* 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 hot standby 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, synchronization can proceed normally. If there is data loss (for example, due to missing transaction logic), it is also reflected in the replicas during the next subscription. 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 subscription. This occurs regardless of whether the tables in the publication are sent 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 subscribing 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. For details on restoring backups, see *"Restoring Backups"* on page 3-22.

- When you restore any backup of a *SynchroNet* 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 *"Monitoring the Status of Synchronization Messages"* on page 3-12. For instructions on correcting stopped synchronization messages, see *"Managing Synchronization Errors"* on page 3-14.

Correcting a Failed Backup

When SOLID *SynchroNet* is performing a backup, the ADMIN COMMAND 'status backup' command returns the value 'ACTIVE'. Once the backup is completed, the command returns either 'OK' or 'FAILED'. You can also query this information using SOLID *DBConsole*.

If the backup failed, you can find the error message that describes the reason for the failure from the `solmsg.out` file in the database directory or in the SOLID *DBConsole* Messages page (accessed through the Administration window or menu). Correct the cause of the error and try again. The most common causes for failed backups are:

- the backup media is out of disk space
- the backup directory does not exist
- a database directory is defined as the backup directory

Restoring Backups

There are two alternative ways to restore a backup. You can either:

- Return to the state when the backup was created, or
- Revive a backup database to the current state by using log files to add data inserted or updated data after the backup was made.

To Return to the State when the Backup was Made

1. Shut down SOLID *SynchroNet*, if it is running.
2. Delete all log files from the log file directory. The default log file names are `sol00001.log`, `sol00002.log`, etc.
3. Copy the database file(s) from the backup directory to the database file directory.
4. Start SOLID *SynchroNet*.

This method will not perform any recovery because no log files exist.

To Revive a Backup Database to the Current State

1. Shut down SOLID *SynchroNet*, if it is running.
2. Copy the database file(s) from the backup directory to the database file directory.
3. Copy the log files from the backup directory to the log file directory. If the same log file exists in both directories, *do not* overwrite files with backup files.
4. Start SOLID *SynchroNet*.

SOLID *SynchroNet* will automatically use the log files to perform a roll-forward recovery.

Recovering from Abnormal Shutdown

If the server was closed abnormally, that is, if it was not shut down using the procedures described earlier, SOLID *SynchroNet* automatically uses the log files to perform a roll-forward recovery during the next start up. No administrative procedures are required to start the recovery.

The message `Starting roll-forward recovery` appears. After the recovery is completed, a message indicates how many transactions were recovered. If no transactions were made since the last checkpoint, this is indicated by the following message:

```
Recovery successfully completed
```

Transaction Logging

The SOLID *SynchroNet* transaction log manager ensures that transaction results are written to permanent storage immediately at commit time. Transaction logging guarantees that no committed operations are lost in case of a system failure. When an operation is executed in the server, the operation is also saved to a transaction log file. The log file is used for recovery in case the server is shut down abnormally.

A backup operation copies the log and database files to the backup directory and deletes the log files from the database directory. You may change this default behavior by changing the values to "no" in the following parameters: `BackupCopyLog` and `BackupDeleteLog` in the `General` section of `solid.ini`.

Tip

For both security and performance reasons, it is a good idea to keep log files and database files on different physical disk devices. If one disk drive is damaged, you will lose either your database files or log files but not both.

Creating Checkpoints

Checkpoints are used to store a consistent state of the database onto the database file. Checkpoints are needed to provide a starting point for roll-forward recovery after a system failure. In the roll-forward recovery, the database will start recovering transactions from the last successful checkpoint. The longer it has been since the last checkpoint was created, the more operations are recovered from the log file(s).

To speed up recoveries, create checkpoints frequently; note, however, that the server performance is reduced during the creation of a checkpoint. Furthermore, the speed of checkpoint creation depends on the amount of database cache used; the more database cache is used, the longer the checkpoint creation will take. You need to consider these issues when deciding the frequency of checkpoints. See *Appendix C, "Configuration Parameters"* for a description of the use of `CacheSize` parameter.

SOLID *SynchroNet* has an automatic checkpoint creation daemon, which creates a checkpoint after a certain number of writes to the log files. The default checkpoint interval is every 5000 log writes. You may change the value of the parameter `CheckpointInterval` in the `General` section of parameters. To learn how to change a parameter value, see *"Managing Parameters"* on page 7-7 in this guide.

Before and after a database operation, you may want to create a checkpoint manually. You can do this programmatically from your application with SQL command ADMIN COMMAND 'makecp'. You can also force a checkpoint using a timed command. Read the section *"Entering Timed Commands"* in this chapter for details.

Note

There can be only one checkpoint in the database at a time. When a new checkpoint is created successfully, the older checkpoint is automatically erased. If the server process is terminated in the middle of checkpoint creation, the previous checkpoint is used for recovery.

Closing a Database

You can close the database which means no new connections to the database are allowed. To do this, issue the following command in SOLID *DBConsole* or SOLID *Remote Control* (teletype):

```
close
```

or in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND ’status’;
```

In some cases you may want to prevent users from connecting to the database. For example, when you are shutting down SOLID *SynchroNet*, you need to prevent new users from connecting to it. After closing the database, only connections from SOLID *DBConsole* will be accepted. Closing the database does not affect existing user connections.

When the database is closed no new connections are accepted (clients will get SOLID Error Message 14506).

SOLID *DBConsole* provides a user interface for managing database connections. For details, For details, refer to *DBConsole* Online Help available by selecting **Help** on the menu bar.

Running Several Servers on One Computer

In some cases, you may want to run two or more databases on one computer. For example, you may need a configuration with a production database and a test database running on the same computer.

SOLID *SynchroNet* is able to provide one database per database server, but you can start several engines each using its own database file. To make these engines use different databases, either start the engine processes from the directories your databases are located in or give the locations of configuration files by using the command line option `-c` *`directory_name`* to change the working directory. Remember to use different network names for each database.

Entering Timed Commands

SOLID *SynchroNet* has a built-in timer, which allows you to automate your administrative tasks. You can use timed commands to execute system commands, to create backups, checkpoints, and database status reports, to open and close databases, and to disconnect users and shut down servers.

To Enter a Timed Command Manually

Edit the `At` parameter of the `[Srv]` section in the `solid.ini` file. The syntax is:

```
At := timed_command[, timed_command]
timed_command := [day] HH:MM command argument
day := sun | mon | tue | wed | thu | fri | sat
```

If the day is not given, the command is executed daily. For details on valid commands, refer to the table at the end of this section.

Example:

```
[Srv]
At=20:30 makecp,21:00 backup,sun 23:00 shutdown
```

▶ **Note**

The format used is HH:MM (24-hour format).

To Enter a Timed Command in SOLID *DBConsole*

Select the **Status** option from the SOLID *DBConsole* Administration window or menu, click the **Scheduler** icon, then click **New** to enter a timed command in the Scheduler dialog box.

In the Scheduler dialog box provide the command, day, time, and arguments in each of the applicable fields. For syntax details, refer to the previous section. Refer to the following section for a list of valid commands.

Arguments and Defaults for the Different Timed Commands

Command	Argument	Default
`backup`	backup directory	the default backup directory that is set in the configuration file
`throwout`	user name, all	no default, argument compulsory
`makecp`	no arguments	no default
`shutdown`	no arguments	no default
`report`	report file name	no default, argument compulsory
`system`	system command	no default
`open`	no argument	no default
`close`	no argument	no default

Managing *SynchroNet* Tables and Databases

After your initial implementation of *SynchroNet*, 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.

Modifying the Database Schema

You can modify the database schema in the master database only if there are no publications referring to that table. However you can modify indexing and user access rights even if the table is referenced by publications.

To modify the database schema:

1. If a publication refers to the table being modified, drop the publication first in the master database using the following command:

   DROP PUBLICATION *publication_name*

   All subscriptions to the publication are dropped automatically as well.

2. Modify the table using the ALTER TABLE or DROP TABLE command. If you are adding a table, you may want to set the SYNCHISTORY TO ON.
3. Re-create the publication(s) that refer(s) to the modified table.
4. When the database schema of the master database is changed, be sure to apply the same alterations to each replica database.

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 *SynchroNet* 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 of MESSAGE APPEND REGISTER REPLICA.

Unregistering a Replica 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.

Creating Large Replica Databases

When creating large replicas that are greater than 2GB from a master database, use the EXPORT SUBSCRIPTION and IMPORT commands. You can export any subscription from a master database to a file and later import that file to a replica. For details, see *"Managing Data with Bookmarks"* on page 3-29.

▶ **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. Large data transfers at a slow rate over the network and there are limitations in sending the large data in one single *SynchroNet* message.

Managing Data with Bookmarks

A bookmark is a state of the SOLID 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 details, read *"Importing and Exporting Subscriptions"* on page 3-31.

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 limit to the number of bookmarks you can create in a database.

Creating bookmarks requires Database Administrator (DBA) or *SynchroNet* Administrator (SYS_SYNC_ADMIN_ROLE) privileges.

Bookmarks retain history information for all tables that have a 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 SOLID system 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 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.

Determining If Bookmarks Are Needed

You can query other system tables to check which replicas and publications require bookmarks. For example:

```
SELECT  r.name, v.tabname, p.name
FROM    sys_sync_replicas r,
        sys_sync_bookmarks b,
        sys_sync_master_versions v,
        sys_publications p
WHERE   r.id = v.replica_id and
        v.version >= b.bm_version and
        v. publ_id = p.id;
```

Dropping Bookmarks

Bookmarks are dropped with the following command in the master database:

```
DROP SYNC BOOKMARK bookmark_name
       bookmark_name := literal
```

After using a bookmark to successfully import a file to a replica or to receive the first subscription 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 subscriptions.

By dropping a bookmark, you delete the history of records. If you do not drop bookmarks, more disk space is consumed and unwanted disk IO is incurred for each bookmark registered in the master database. This may result in unwanted performance degradation.

Note that 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 subscribe to 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.

For more details, *"DROP SYNC BOOKMARK"* on page 13-15.

Importing and Exporting Subscriptions

The SOLID *SynchroNet* IMPORT and EXPORT SUBSCRIPTION commands let you export a version of the data from a master database to a disk file and import that data into a replica. 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 is similar to subscribing to a publication. You use the EXPORT SUBSCRIPTION command instead of the MESSAGE APPEND SUBSCRIBE 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 *"Creating a Replica By Exporting a Subscription with Data"* on page 3-32 or *"Creating a Replica By Exporting a Subscription Without Data"* on page 3-35.
- You want to export specific versions of the data to a replica.
- You want to export meta data information only without the actual row data.

Note the difference in using the EXPORT SUBSCRIPTION command from subscribing to a publication with MESSAGE APPEND SUBSCRIBE:

- The EXPORT SUBSCRIPTION command is executed in the master, whereas a subscription is requested from a replica.
- The export output is saved to a user specified file, whereas a subscription output is stored in a *SynchroNet* 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 subscription 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 subscriptions based on incremental publications are possible from that bookmark.

EXPORT SUBSCRIPTION COMMAND

The EXPORT SUBSCRIPTION command is created using the following syntax:

```
EXPORT SUBSCRIPTION publication_name [(arguments)]
        TO 'filename'
        USING BOOKMARK bookmark_name;
        [WITH [NO] DATA];
```

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 *"EXPORT SUBSCRIPTION"* on page 13-16. For the procedure to create a replica WITH DATA, read *"Creating a Replica By Exporting a Subscription with Data"* on page 3-32 and to create a replica WITH NO DATA, read *"Creating a Replica By Exporting a Subscription Without Data"* on page 3-35.

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 data for import to a replica can be contained only in one file. 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 *"IMPORT"* on page 13-21.

Creating a Replica By Exporting a Subscription with Data

You use the EXPORT SUBSCRIPTION command using the WITH DATA option to create a replica when data is exported to an existing database that does not contain master data and requires a subset of data.

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**

1. 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.
2. SOLID *SynchroNet* 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 *“Managing Data with Bookmarks”* on page 3-29 for information on creating bookmarks.

   You can also perform queries to see what bookmarks and publications currently exist in your system. Refer to *“Retrieving Bookmark Information”* on page 3-29.
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 meta data and versioned data corresponding to that publication and bookmark is 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 25057, indicating that the *SynchroNet* 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. Create a new database or a backup of an existing master database (for example, using ADMIN COMMAND 'backup').
2. If you are using a master backup database, start the database and drop all replicas using the DROP REPLICA command and all publications using the DROP PUBLICATION command.
3. Specify the node name using the SET SYNC NODE *unique_node_name* command. If you are using master backup database, use this same command to change the node name, since the original master is currently using the node name.
4. Configure the database as a replica by executing SET SYNC REPLICA YES. If you are using a master backup database, also specify SET SYNC MASTER NO.

Note

If you are configuring a master backup database, note that the backup is already set as a master database. If you set this backup using only SET SYNC REPLICA YES, the database is then defined as a dual role (master and replica) database, instead of a dedicated replica-only database.

5. Register this replica with the master database.
6. Register the publication whose subscription(s) will be imported.
7. Import the file(s) you created using the EXPORT SUBSCRIPTION command. The procedure for import is described in *"Specifying a Subscription for Import"* on page 3-32.

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.

8. Subscribe to the publication(s) from the master database using *SynchroNet*'s MESSAGE commands.

Creating a Replica By Exporting a Subscription Without Data

You use the EXPORT SUBSCRIPTION command with the WITH NO DATA option to create a replica when data is exported to a database that already contains the master data (for example, using a backup copy of an existing master).

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

SOLID *SynchroNet* 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 meta data 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 *"Managing Data with Bookmarks"* on page 3-29 for information on creating bookmarks.

   You can also perform queries to see what bookmarks and publications currently exist in your system. Refer to *"Retrieving Bookmark Information"* on page 3-29.

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 meta data and versioned data corresponding to that publication and bookmark is 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, using ADMIN COMMAND 'backup').

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

Because you are configuring a master backup database, note that the backup is already set as a master database. If you set this backup using only SET SYNC REPLICA YES, the database is then defined as a dual role (master and replica) database, instead of a dedicated replica-only database.

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 *"Specifying a Subscription for Import"* on page 3-32.
7. Subscribe to the publication(s) from the master database using *SynchroNet*'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 *"Dropping Bookmarks"* on page 3-30.

Modifying Publications and Subscriptions

You can alter publications in the master database by dropping the existing publication and recreating a new one with the same name. All subscriptions that refer to the previous version of the publication must be dropped and re-subscribed in the replica databases. In other words, altering a publication means that the replica's next subscription to that publication is always a full subscription.

An attempt to subscribe to a changed publication without dropping the old subscription produces a version mismatch error during the execution of the synchronization message.

Note also that 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 recreate tables, procedures, and publications.

Managing Access Rights

This section describes tasks required to manage access rights in your *SynchroNet* installation. For a detailed overview on how *SynchroNet* controls security, available access rights, and special *SynchroNet* roles, read *"Implementing Security through Access Rights and Roles"* on page 12-9.

Changing Replica Access to the Master Database

You need to manage user access to the master database so that transactions, including other *SynchroNet* operations, can be sent to the master database. If you have added a new master user to the master database, you will need to download this information to applicable replicas. To do this, you need to execute the MESSAGE APPEND SYNC_CONFIG command in the replicas. In addition, if you add a new replica, 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.

Updating Master Users for *SynchroNet* 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;
```

Changing Access Rights to Publications

To add or remove access rights to publications, use the GRANT SUBSCRIBE and REVOKE SUBSCRIBE commands. When adding a new user for access to a publication, be sure the user also has access to the tables that reference the publication. This gives the user the right to view the publication.

Note

Access rights take effect when the user who is granted the access rights logs on to the database. If the user is already logged on to the database when the access rights are initiated, the user must disconnect and then reconnect to the database for the access rights to take effect.

Granting SUBSCRIBE 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 SUBSCRIBE statement in the master database. The syntax is:

GRANT SUBSCRIBE ON *publication_name* TO {PUBLIC **|** *user_name*,

[*user_name*] ... **|** *role_name*, [*role_name*, [*role_name*] ...}

For example:

```
GRANT SUBSCRIBE ON customers_by_area TO salesman_jones
```

Revoking SUBSCRIBE 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 SUBSCRIBE statement in the master database. The syntax is:

REVOKE SUBSCRIBE ON *publication_name* FROM {PUBLIC **|** *user_name*,

    [*user_name*] ... **|** *role_name*, [*role_name*, [*role_name*] ...}

For example:

```
REVOKE SUBSCRIBE ON customers_by_area FROM salesman_jones
```

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.

4

Using SOLID SQL for Data Management

You manage SOLID databases as well as its users and schema using SOLID SQL statements. This chapter describes the management tasks you perform with SOLID SQL. These tasks include managing roles and privileges, tables, indexes, transactions, catalogs, and schemas.

Using SOLID SQL Syntax

The SQL syntax is based on the ANSI X3H2-1989 level 2 standard including important ANSI X3H2-1992 (SQL2) extensions. User and role management services missing from previous standards are based on the ANSI SQL3 draft. Refer to *Appendix E, "SOLID SQL Syntax"* for a more formal definition of the syntax.

SQL statements must be terminated with a semicolon (;) *only* when using SOLID *SQL Editor* or SOLID *DBConsole*. Otherwise, terminating SQL statements with a semicolon leads to a syntax error.

You can use SOLID *DBConsole* (as well as SOLID *SQL Editor* and ODBC compliant tools) to execute SQL statements. To automate the tasks, you may want to save the SQL statements to a file. You can use these files for rerunning your SQL statements later or as a document of your users, tables, and indexes.

SOLID SQL Data Types

SOLID SQL supports data types specified in the SQL2 Standard Entry Level specifications, as well as important Intermediate Level enhancements. Refer to *Appendix E, "SOLID SQL Syntax"* for a complete description of the supported data types.

You can also define some data types with the optional length, scale, and precision parameters. In that case, the default properties of the corresponding data type are not used.

SOLID SQL Extensions

SOLID SQL provides the extension ADMIN COMMAND '*command*[*command_args*]' to perform basic administrative tasks, such as backups, performance monitoring, and shutdown.

You can use SOLID *SQL Editor* (teletype) to execute the command options provided by ADMIN COMMAND. To access a short description of available ADMIN COMMANDs, execute `ADMIN COMMAND 'help'`. For a formal definition of the syntax of these statements, refer to *Appendix E, "SOLID SQL Syntax"* in this guide.

▶ **Note**

ADMIN COMMAND tasks are also available as administrative commands in SOLID *DBConsole* and SOLID *Remote Control* (teletype). Read Chapter 5,"Using SOLID Data Management Tools" for details.

Managing User Privileges and Roles

You can use SOLID *DBConsole*, SOLID teletype tools, and many ODBC compliant SQL tools to modify user privileges. Users and roles are created and deleted using SQL statements or commands. A file consisting of several SQL statements is called a SQL script.

In the `\samples\procedures` directory, you will find the SQL script `Sample.sql`, which gives an example of creating users and roles. You can run it using SOLID *DBConsole*. To create your own users and roles, you can make your own script describing your user environment.

User Privileges

When using SOLID databases in a multi-user environment, you may want to apply user privileges to hide certain tables from some users. For example, you may not want an employee to see the table in which employee salaries are listed, or you may not want other users to mess with your test tables.

You can apply five different kinds of user privileges. A user may be able to view, delete, insert, update or reference information in a table or view. Any combination of these privileges may also be applied. A user who has none of these privileges to a table is not able to use the table at all.

▶ **Note**

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

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

User Roles

Privileges can also be granted to an entity called a role. A role is a group of privileges that can be granted to users as one unit. You can create roles and assign users to certain roles.

Note

1. The same string cannot be used both as a user name and a role name.
2. 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 when the role is granted, the role takes effect when the user disconnects and then reconnects to the database.

The following user names and roles are reserved:

Reserved Names	Description
PUBLIC	This role grants privileges to all users. When user privileges to a certain table are granted to the role `PUBLIC`, all current and future users have the specified user privileges to this table. This role is granted automatically to all users.
SYS_ADMIN_ROLE	This is the default role for the database administrator. This role has administration privileges to all tables, indexes and users, as well as the right to use SOLID *DBConsole* and SOLID *Remote Control* (teletype). This is also the role of the creator of the database.
_SYSTEM	This is the schema name of all system tables and views.
SYS_CONSOLE_ROLE	This role has the right to use SOLID *DBConsole*, but does not have other administration privileges.
SYS_SYNC_ADMIN_ROLE	This is the administrator role for data synchronization functions.

Reserved Names	Description
SYS_SYNC_REGISTER_ROLE	This role is only for registering and unregistering a replica database to the master.

Examples of SQL Statements

Below are some examples of SQL commands for administering users, roles, and user privileges.

Creating Users

CREATE USER *username* IDENTIFIED BY *password*;

Only an administrator has the privilege to execute this statement. The following example creates a new user named `CALVIN` with the password `HOBBES`.

```
CREATE USER CALVIN IDENTIFIED BY HOBBES;
```

Deleting Users

DROP USER *username*;

Only an administrator has the privilege to execute this statement. The following example deletes the user named `CALVIN`.

```
DROP USER CALVIN;
```

Changing a Password

ALTER USER *username* IDENTIFIED BY *new password*;

The user *`username`* and the administrator have the privilege to execute this command. The following example changes `CALVIN's` password to `GUBBES`.

```
ALTER USER CALVIN IDENTIFIED BY GUBBES;
```

Creating Roles

CREATE ROLE *rolename*;

The following example creates a new user role named `GUEST_USERS`.

```
CREATE ROLE GUEST_USERS;
```

Deleting Roles

DROP ROLE *role_name*;

The following example deletes the user role named `GUEST_USERS`.

```
DROP ROLE GUEST_USERS;
```

Granting Privileges to a User or a Role

GRANT *user_privilege* ON *table_name* TO *username* or *role_name*;

The possible user privileges on tables are SELECT, INSERT, DELETE, UPDATE, REFERENCES and ALL. ALL provides a user or a role all five privileges mentioned above. A new user has no privileges until they are granted.

The following example grants `INSERT` and `DELETE` privileges on a table named `TEST_TABLE` to the `GUEST_USERS` role.

```
GRANT INSERT, DELETE ON TEST_TABLE TO GUEST_USERS;
```

The EXECUTE privilege provides a user the right to execute a stored procedure:

GRANT EXECUTE ON *procedure_name* TO *username* or *role_name*;

The following example grants EXECUTE privilege on a stored procedure named SP_TEST to user CALVIN.

```
GRANT EXECUTE ON SP_TEST TO CALVIN;
```

Granting Privileges to a User by Giving the User a Role

GRANT *role_name* TO *username*;

The following example gives the user `CALVIN` the privileges that are defined for the `GUEST_USERS` role.

```
GRANT GUEST_USERS TO CALVIN;
```

Revoking Privileges from a User or a Role

REVOKE *user_privilege* ON *table_name* FROM *username* or *role_name*;

The following example revokes the `INSERT` privilege on the table named `TEST_TABLE` from the `GUEST_USERS` role.

```
REVOKE INSERT ON TEST_TABLE FROM GUEST_USERS;
```

Revoking Privileges by Revoking the Role of a User

REVOKE *role_name* FROM *username*;

The following example revokes the privileges that are defined for the `GUEST_USERS` role from `CALVIN`.

```
REVOKE GUEST_USERS FROM CALVIN;
```

Granting Administrator Privileges to a User

GRANT SYS_ADMIN_ROLE TO *username*;

The following example grants administrator privileges to `CALVIN`, who now has all privileges to all tables.

```
GRANT SYS_ADMIN_ROLE TO CALVIN;
```

▶ **Note**

If the autocommit mode is set OFF, you need to commit your work. To commit your work use the following SQL statement: `COMMIT WORK;` If the autocommit mode is set ON, the transactions are committed automatically.

Managing Tables

A Solid server has a dynamic data dictionary that allows you to create, delete and alter tables on-line. SOLID database tables are managed using SQL commands.

In the SOLID directory, you can find a SQL script named `sample.sql`, which gives an example of managing tables. You can run the script using SOLID *DBConsole*.

Below are some examples of SQL statements for managing tables. Refer to *Appendix E, "SOLID SQL Syntax"* for a formal definition of the SOLID SQL statements.

If you want to see the names of all tables in your database, issue the SQL statement `SELECT * FROM TABLES` or use predefined command TABLES from SOLID *DBConsole*. The table names can be found in the column `TABLE_NAME.`

Examples of SQL Statements

Below are some examples of SQL commands for administering tables.

Creating Tables

CREATE TABLE *table_name* (*column column type*

[,*column column type*]...);

All users have privileges to create tables.

The following example creates a new table named `TEST` with the column `I` of the column type `INTEGER` and the column `TEXT` of the column type `VARCHAR`.

```
CREATE TABLE TEST (I INTEGER, TEXT VARCHAR);
```

Removing Tables

DROP TABLE *table_name*;

Only the creator of the particular table or users having `SYS_ADMIN_ROLE` have privileges to remove tables.

The following example removes the table named `TEST`.

```
DROP TABLE TEST;
```

Adding Columns to a Table

ALTER TABLE *table_name* ADD COLUMN *column_nam*e

column_type;

Only the creator of the particular table or users having `SYS_ADMIN_ROLE` have privileges to add or delete columns in a table.

The following example adds the column `C` of the column type `CHAR(1)` to the table `TEST`.

```
ALTER TABLE TEST ADD COLUMN C CHAR(1);
```

Deleting Columns from a Table

ALTER TABLE *table_name* DROP COLUMN

column_name;

A column cannot be dropped if it is part of a unique or primary key. For details on primary keys, read *"Managing Indexes"* on page 4-8.

The following example statement deletes the column `C` from the table `TEST`.

```
ALTER TABLE TEST DROP COLUMN C;
```

▶ **Note**

If the autocommit mode is set OFF, you need to commit your work before you can modify the table you altered. To commit your work after altering a table, use the following SQL statement: `COMMIT WORK;`. If the autocommit mode is set ON, transactions are committed automatically.

Managing Indexes

Indexes are used to speed up access to tables. The database engine uses indexes to access the rows in a table directly. Without indexes, the engine would have to search the whole contents of a table to find the desired row. For details on creating indexes to improve performance, read *"Using Indexes to Improve Query Performance"* on page 8-2.

There are two kinds of indexes: non-unique indexes and unique indexes. A unique index is an index where all key values are unique. You can create as many indexes as you like on a single table; however, adding indexes does slow down write operations, such as inserts, deletes, and updates on that table.

You can create and delete indexes using the following SQL statements. Refer to *Appendix E, "SOLID SQL Syntax"* for a formal definition of the syntax for these statement.

Examples of SQL Statements

Below are some examples of SQL commands for administering indexes.

Creating an Index on a Table

CREATE [UNIQUE] INDEX *index_name* ON *base_table_name*

[*column_identifier* [ASC **|** DESC]

[, *column_identifier* [ASC **|** DESC]] ...

Only the creator of the particular table or users having SYS_ADMIN_ROLE have privileges to create or delete indexes.

The following example creates an index named `X_TEST` on the table `TEST` to the column `I`.

```
CREATE INDEX X_TEST ON TEST (I);
```

Creating a Unique Index on a Table

CREATE UNIQUE INDEX *index_name* ON *table_name*

(*column_name*);

The following example creates a unique index named `UX_TEST` on the table `TEST` to the column `I`.

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);
```

Deleting an Index

DROP INDEX *index_name*;

The following example deletes the index named `X_TEST`.

```
DROP INDEX X_TEST;
```

Note

If the autocommit mode is set OFF, you need to commit your work before you can modify the table on which you altered the indexes. To commit your work after modifying indexes, use the following SQL statement: `COMMIT WORK;`. If the autocommit mode is set ON, the transactions are committed automatically.

Primary Keys

A primary key is a column or combination of columns that uniquely identify each record in a table. Primary keys like indexes speed up access to tables.

In a Solid server, the difference between primary keys and indexes is that the primary key clusters data in the database according to the key values. This clustering process is described in the following section. Without a primary key defined to a table, rows are ordered on disk according to the time in which they were inserted into the database.

Data clustering

The storage server, part of a Solid server's indexing system, is used to store both secondary keys and primary keys (containing the actual data values). By defining a primary key for a table, you allow a Solid server to use the key to physically cluster the data rows to the order given by the index.

The set of columns used for clustering is called the row reference. The row reference uniquely identifies the data row. If the user-defined columns for the clustering key are not unique, the system ensures that the reference is unique by adding a unique row number to the reference columns. The row reference is also known as the “row identifier.”

The row reference can be any combination of one or more columns. Each table has a different set of columns that are used for the unique row reference.

Secondary Keys

Some tables also have a secondary key to implement primary key referencing. In this case, a secondary key value refers to a data row using the row reference. If all the requested data is found from the secondary key, no search on the clustering key is performed. Otherwise, the data is searched from the clustering key using the row reference as the search argument.

Foreign Keys

A foreign key is a column or group of columns within a table that refers to, or relates to, some other table through its values. The foreign key must always include enough columns in its definition to uniquely identify a row in the referenced table. A foreign key must contain the same number of columns as the primary key and be in the same order; however, a foreign key can have different column names and default values than the primary key.

The main reason for defining foreign keys is to ensure the validity of references between tables. Rows in one table must always have corresponding rows in another table, thereby maintaining referential integrity,

You define the rules for referential integrity as part of the CREATE TABLE statement through primary and foreign keys. For example:

```
CREATE TABLE DEPT (
                 DEPT INTEGER NOT NULL,
                 DNAME VARCHAR,
                 PRIMARY KEY (DEPTNO));

CREATE TABLE EMP (
                 DEPTNO INTEGER,
                 ENAME VARCHAR,
          FOREIGN KEY (DEPTNO) REFERENCES DEPT (DEPTNO));
```

Refer to *Appendix E, “SOLID SQL Syntax”* for CREATE TABLE syntax detail.

Managing Transactions

A transaction is a group of SQL statements treated as a single unit of work; either all the statements are executed as a group, or none are executed. This section assumes you know the fundamentals for creating transactions using standard SQL statements. It describes how SOLID SQL lets you handle transaction behavior, concurrency control, and isolation levels.

Defining Read-only and Read-write Transactions

To define a transaction to be read-only or read-write, use the following SQL commands:

```
SET TRANSACTION READ ONLY | READ WRITE
```

The following options are available with this command.

- READ ONLY

  Use this option for a read only transaction.

- READ WRITE

  Use this option for a read and write transaction. This option is the default.

Note

To detect conflicts between transactions, use the standard ANSI SQL command SET TRANSACTION ISOLATION LEVEL to define the transaction with a Repeatable Read or Serializable isolation level. For details, read *"Choosing Transaction Isolation Levels"* on page 4-13.

Transactions are ended with the COMMIT WORK or ROLLBACK WORK commands.

Setting Concurrency Control

The primary model used for concurrency is a multiversioning and optimistic concurrency control method. Multiversioning means that multiple versions of the same row can co-exist in the database. This way, users are able to concurrently access the database at the same time, and the view of the data that they access is consistent throughout the transaction. Data is always available to users because locking is not used; access is improved since deadlocks no longer apply. For details, read *"SOLID Bonsai Tree Multiversioning and Concurrency Control"* on page 1-4. The optimistic concurrency control is automatically set for all tables.

Setting Pessimistic and Mixed Concurrency Control

When necessary, you can use pessimistic (row-level locking) or mixed concurrency control methods. There are situations when pessimistic concurrency control is more appropriate. For example, in some applications there are small areas that are very frequently updated. In the case of these so-called *hotspots*, conflicts are so probable that optimistic concurrency control wastes effort in rolling back conflicting transactions.

You can also use mixed concurrency control, a combination of row-level locking and optimistic concurrency control. By turning on row-level locking table-by-table, you can specify that a single transaction use both concurrency control methods simultaneously. This functionality is available for both read-only and read-write transactions.

Note that since pessimistic ordering of SOLID *SynchroNet* is managed on the row level, there is no need to manage page or table level locking.

To set individual tables for optimistic or pessimistic concurrency, use the following SQL command:

ALTER TABLE *base_table_name* SET {OPTIMISTIC | PESSIMISTIC}

Note that by default all tables are set for optimistic.

You can also set a database-wide default in the General section of the configuration file with the following parameter:

```
Pessimistic = yes
```

Locking

To control the level of consistency and concurrency in the application, locks are placed on rows when users are submitting queries or updates to rows. The following lock modes are used only for pessimistic tables:

- SHARED

  Multiple users can hold shared locks on the same row simultaneously. Shared locks are used on queries.

- UPDATE

  When a user accesses a row with the SELECT... FOR UPDATE statement, the row is locked with an update mode lock. This means that no other user can read or update the row, and ensures the current user can later update the row.

- EXCLUSIVE

  Only one user has an exclusive lock on a row at any given time. Exclusive locks are used on insert, update, and delete operations.

Setting Lock Timeout

The lock timeout setting is the time in seconds that the engine waits for a lock to be released. By default, lock timeout is set to 30 seconds. When the timeout interval is reached, SOLID *SynchroNet* terminates the timed out transaction. For example, if a user is querying a specific row in a table and the second user is inserting data into the same row, the insert will not go through until the first user's query is completed or times out. Once completed, a lock is then issued for the second user's insert transaction.

You can set the lock time out with the following SQL command:

SET LOCK TIMEOUT *timeout_in_seconds*

Setting Lock Timeout for Optimistic Tables

When you use SELECT FOR UPDATE, the selected rows are locked also for tables with optimistic concurrency control. To set the lock timeout separately for optimistic tables per connection, use the following SQL command:

SET OPTIMISTIC LOCK TIMEOUT *seconds*

Choosing Transaction Isolation Levels

Concurrency control is based on an applications requirements. Some applications need to execute as if they had the exclusive ownership of the database. Other applications can tolerate some degree of interference from other applications running simultaneously. To meet the needs of different applications, the SQL2 standard defines four levels of isolation for transactions.

- Read Uncommitted

  This isolation level allows transactions to read data modified by other transactions that have not yet committed.

Note

The Read Uncommitted mode (known also as the'dirty read' mode) is not supported in Solid databases. Its purpose has been to enhance concurrency in DBMSs that use locking, but it sacrifices the consistent view and potentially also database integrity.

- Read Committed

  This isolation level allows a transaction to read only committed data. Still, the view of the database may change in the middle of a transaction when other transactions commit their changes. Read Committed does not prevent phantom updates, but it does ensure that the results set returned by a single query is consistent by setting the read level to the latest committed transaction when the query is started.

- Repeatable Read

  This isolation level is the default isolation level for SOLID databases. It allows a transaction to read only committed data and guarantees that read data will not change until the transaction terminates. SOLID *SynchroNet* additionally ensures that the transaction sees a consistent view of the database. Conflicts between transactions are detected by using transaction write-set validation. Still, phantom updates may occur.

- Serializable

  This isolation level allows a transaction to read only committed data with a consistent view of the database. Additionally, no other transaction may change the values read by the transaction before it is committed because otherwise the execution of transactions cannot be serialized in the general case.

  SOLID *SynchroNet* can provide serializable transactions by detecting conflicts between transactions. It does this by using both write-set and read-set validations. Because no locks are used, all concurrency control anomalies are avoided, including the phantom updates.

Setting the Isolation Level

To set the isolation level, use the following SQL command:

```
SET TRANSACTION ISOLATION LEVEL
    READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

For example:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Managing Database Objects

In keeping with ANSI SQL and ISO standards, schema and catalog support are provided for Solid database objects. Catalogs allow you to logically partition databases so you can organize your data to meet the needs of your business or application.

A catalog can qualify one or more schema names. A schema is a persistent database object that provides a definition for the entire database. It represents a collection of database objects associated with a specific schema name. These objects include tables, views, indexes, stored procedures, triggers, and sequences.

The catalog name is used to qualify a database object name. They are qualified in all DML statements as:

catalog_name.schema_name.database_object

or

catalog_name.user_id.database_object

You can create a schema without having it qualified by a catalog.You can qualify a schema with one or more database objects. To use a schema name with a database object, create the schema first.

schema_name.database_object_name

or

user_id.database_object_name

By default, database objects that are created without schema names are qualified using the user ID of the database object's creator. For example:

*user_id.table_name*e

Catalog and schema contexts are set using the SET CATALOG or SET SCHEMA statement.

If a catalog context is not set using SET CATALOG, then all database object names are resolved always using the default catalog name.

Note

When creating a new database or converting an old database to a new format, the user is prompted to specify a default catalog name for the database system catalog. Users can access the default catalog name without knowing this specified default catalog name. For example, users can specify the following syntax to access the system catalog:

""._SYSTEM.*table*

SOLID *SynchroNet* translates the empty string ("") specified as a catalog name to the default catalog name. *SynchroNet also* provides for automatic resolution of _SYSTEM schema to the system catalog, even when users provide no catalog name.

The following SQL statement provide examples of creating catalogs and schemas. Refer to *Appendix E, "SOLID SQL Syntax"* for a formal definition of the SOLID SQL statements.

Examples of SQL Statements

Below are some examples of SQL commands for managing database objects.

Creating a Catalog

CREATE CATALOG *catalog_name*

Only the creator of the database or users having SYS_ADMIN_ROLE have privileges to create or delete catalogs.

The following example creates a catalog named `C` and assumes the userid is `SMITH`

```
CREATE CATALOG C;
SET CATALOG C;
CREATE TABLE T;
SELECT * FROM T;
--The name T is resolved to C.SMITH.T
```

Setting a Catalog and Schema Context

The following example sets a catalog context to `C` and the schema context to `S`.

```
SET CATALOG C;
SET SCHEMA S;
CREATE TABLE T;
SELECT * FROM T;
-- the name T is resolved to C.S.T
```

Deleting a Catalog

DROP CATALOG *catalog_name*;

The following example deletes the catalog named `C`.

```
DROP CATALOG C;
```

Creating a Schema

CREATE SCHEMA *schema_name*

Any database user can create a schema; however, the user must have permission to create the objects that are pertain to the schema (for example, CREATE PROCEDURE, CREATE TABLE, etc.).

The following example creates a schema named FINANCE and assumes the user id is SMITH:

```
CREATE SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);
-- This table is qualified to SMITH.EMPLOYEE
SET SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (ID INTEGER);
SELECT ID FROM EMPLOYEE;
-- In this case, the table is qualified to FINANCE.EMPLOYEE
```

Deleting a Schema

DROP SCHEMA *schema_name*;

The following example deletes the schema named FINANCE.

```
DROP SCHEMA FINANCE;
```

5

Using SOLID Data Management Tools

This chapter describes SOLID Data Management Tools, a set of utilities for performing various database tasks. These tools include:

- SOLID *DBConsole*, an easy-to-use graphical user interface for administration and configuration tasks, monitoring local and remote Solid servers, issuing SQL queries and statements, and executing SQL script files.
- SOLID *Remote Contro*l (teletype) and SOLID *SQL Editor* (teletype) for command line sessions at the operating system prompt.
- SOLID *SpeedLoader* for loading data from external ASCII files into a SOLID database.
- SOLID *Export* is a product for unloading data from a SOLID database to ASCII files.
- SOLID *Data Dictionary* for retrieving data definition statements from a SOLID database.

Note

Not all SOLID Tools are necessarily part of the standard product delivery, and their availability on some platforms may be limited. For information about SOLID data management tools, contact your SOLID sales representative or SOLID Online Services at the SOLID Web site:

`http://www.solidtech.com/`

SOLID *DBConsole*

SOLID *DBConsole* is a java-based, graphical user interface for managing, administering, and querying local and remote Solid servers. Designed for intuitive and efficient ease-of-use, it allows you to create and manipulate database schemas, browse data, monitor and manage both local and remote databases, and configure Solid server parameters.

With SOLID *DBConsole* you can use an Administration window, which features an intuitive interface to perform the basic administration tasks described in this manual. You can also use a Query window to issue administrative commands (equivalent to SOLID SQL ADMIN COMMANDs) and create and execute script files.

Note

To perform administration operations in SOLID *DBConsole* requires SYS_ADMIN_ROLE rights.

Starting *DBConsole*

To start SOLID *DBConsole*:

- Enter the following command at your operating system prompt:

  ```
  java DBConsole
  ```

  -or-

- When using Windows, start *DBConsole* from the icon in your Program Group.

This displays the *DBConsole* interface, where you perform database tasks by using the menubar, toolbar, or right-click mouse menus that apply to particular items in the work-space.

Note

Ensure that the SOLID database server is running before establishing a database connection. Use the Add Database dialog box to add additional databases and the Connect dialog box to connect to the databases. For details, refer to *DBConsole* Online Help available by selecting

Help on the menu bar.

You can also start *DBConsole* by including one or more of these optional command line arguments:

```
java DBConsole options
```

where *options* can be:

-M*mode*	*mode* = BATCH; specifies that *DBConsole* run in Batch Mode without showing the user interface.
-D*databasename*	Specifies a database for connection.
-U*url*	Specifies the JDBC URL required for *DBConsole* to connect to a Solid server. The format of the JDBC URL is: JDBC:SOLID://*machine_name*:*port_number* For example: `jdbc:solid://localhost:1313`
-u*userid*	Specifies the user ID for accessing the database
-p*password*	Specifies the user's password for accessing the database
-f*queryfile*	Executes the SQL statements contained in the script file.
-e*sql_strings*	Executes the SQL strings.
-o*Outputfile*	Specifies the file where resultsets are stored.
-O*Outputfile*	Specifies the file (which will be opened for writing in append mode) where resultsets are stored.
-a	All transactions are autocommitted.
-h	Help Usage

DBConsole Interface Features

DBConsole opens each new database connection with three separate windows: a Browse window, a Query window, and an Administration window. You can move from one window to another to manage different databases simultaneously.

▶ **Note**

The features of each window are described briefly in the following sections. For details on usage, refer to the *DBConsole* Online Help available by selecting **Help** on the menu bar.

Query Window

With the Query window, you can issue administrative commands (equivalent to SOLID SQL ADMIN COMMANDs), SQL queries and statements, or execute a script file that contains these items. For a list of administrative commands, see *"Administrative Commands"* on page 5-6.

A results section in the Query window displays error messages and the result set, which you can print or save to a text file. If needed, you can cancel execution of a current SQL statement and specify transaction commits and rollbacks. Settings are also available to enable autocommit and the transaction isolation level for a connection.

Administration Window

With the Administration window, you can monitor server status (including messages) and control all Solid servers in a network from a single workstation. From the Administration window, you can perform the following local and remote operations:

- Control user access to databases
- Control network protocol connections
- Generate backups and checkpoints
- Create timed commands to automate administration
- Configure a Solid server's parameters

Browse Window

With the Browse window, you can browse database objects, which include tables, columns, views, indexes, stored procedures, sequences, roles, and users. A database workspace gives you a quick view of database connections, databases, and their objects in a tree format. You can click on a node in the tree to browse an object, which is displayed in table format. For easier viewing, you can rearrange data columns by moving and resizing table headers.

SOLID *Remote Control* (teletype)

With SOLID R*emote Control* (teletype), you can execute administrative commands (equivalent to the SOLID SQL ADMIN COMMANDs), at the command line, command prompt, or by executing a script file that contains the commands. For a list of these commands, see *"Administrative Commands"* on page 5-6.

Note

The user performing the administration operation must have SYS_ADMIN_ROLE or

SYS_CONSOLE_ROLE rights, or the connection will be refused.

Starting SOLID *Remote Control* (teletype)

Start SOLID *Remote Control* by issuing the command `solcon` or `load solcon` (Novell Netware) at the operating system prompt.

You can also specify the following syntax and include these optional command line arguments:

solcon *options servername username password*

where *options* can be:

-c*dir*	Change working directory
-e*command string*	Execute the specified *Remote Control* command
-f*filename*	Execute command string from a script file
-h, **-?**	Help = Usage

Servername is the network name of a Solid server that you are connected to. Logical Data Source Names can also be used with tools; Refer to Chapter 6,“Managing Network Connections” for further information. The given network name must be enclosed in quotes.

Username is required to identify the user and to determine the user’s authorization. Without appropriate rights, execution is denied.

Password is the user’s password for accessing the database.

SOLID *Remote Control* connects to the first server specified in the `Connect` parameter in the `solid.ini` file. If you specify no arguments, you are prompted for the database administrator’s user name and password. You can give connection information at the command line to override the connect definition in `solid.ini`.

To exit *Remote Control*, enter the command `exit`.

Examples

Start up Remote Control with the servername and the administrator’s username and password:

```
solcon “tcp localhost 1313” admin iohi4y
```

Start up Remote Control to back up a specific database:

```
solcon -ebackup “ShMem SOLID” dbadmin password
```

Entering Commands in SOLID *Remote Control* (teletype)

After the connection to the server is established, the command prompt appears.

You can execute all commands at the command line with the **-e** option or in a text file with the **-f** option. You can also execute all SOLID *Remote Control* commands programmatically from an application using options of the SQL command “ADMIN COMMAND”. For example, you can start a backup with the SQL command ADMIN COMMAND ‘backup’.

When there is an error in the command line, SOLID *Remote Control* gives you a list of the possible options as a result. Please be sure to check the command line you entered.

Administrative Commands

Command	Abbreviation	Explanation
backup [*backup_directory*]	bak	Makes a backup of the database. The default backup directory is the one defined in parameter BackupDirectory in the General section of the configuration file. The backup directory may also be given as an argument. For example, **backup abc** creates backup on directory ‘abc’. All directory definitions are relative to the SOLID *Embedded Engine* working directory.
backuplist	bls	Displays a status list of last backups.
close	clo	Closes server from new connections; no new connections are allowed.
describe parameter *param*	des par	Returns description text of parmeter. The following example describes parameter [Com]Trace=y/n. `describe parameter com.trace`
errorcode *SOLID_error_code*	ec	Displays a description of an error code. Provide the error code as an argument. For example: **errorcode 10033**
exit	ex	Exits SOLID *Remote Control* or SOLID *DBConsole.*

Command	Abbrevia-tion	Explanation
info *options*	in	Returns server information. Options are one or more of the following values, each separated by a space: ▪ **Numusers** - number of current users ▪ **Maxusers** - maximum number of users ▪ **Sernum** - Server serial number ▪ **Dbsize** - database size ▪ **Logsize** - size of log files ▪ **Uptime** - server up since ▪ **Bcktime** - timestamp of last successfully completed backup ▪ **Cptime** - timestamp of last successfully completed checkpoint ▪ **tracestate** - Current trace statem ▪ **monitorstate** - Current monitor state, number of users withmonitor enabled, or -1 if all ▪ **openstate** - Current open or close state Values are returned in the same order as requested, one row for each value. Example: `ADMIN COMMAND ’info dbsize logsize’`
help	?	Displays available commands.
makecp	mcp	Makes a checkpoint.
messages [**-n**] [**warnings** | **errors**] [*count*]	mes	Displays server messages. Optional severity and message numbers can also be defined. For example: `messages warning 100` The above example displays the last 100 warnings.
monitor {**on** | **off**} **[user** *username* | *userid*]	mon	Sets server monitoring on and off. Monitoring logs user activity and SQL calls to `soltrace.out` file

Command	**Abbrevia-tion**	**Explanation**
notify {**user** *username* | *user id* | **ALL** } *message*	not	This command sends an event to a given user with event identifier NOTIFY. This identifier is used to cancle an event waiting thread when the statement timeout is not long enough for a disconnect or to change the event registration. The following example sends a notify message to a user with user id 5; the event then gets the value of the message parameter. `notify user 5 Canceled by admin`
open	ope	Opens server for new connections; new connections are allowed.
parameter [*option*][*name*[=*value*]]	par	Displays server parameter values. For example: ■ **parameter** used alone displays all parameters ■ **parameter genera**l displays all parameters from section “general.” ■ **parameter general.readonly** displays a single parameter “readonly” from section “general.” ■ **parameter com.trace=yes** sets communication trace on If **-r** is used, then only the current parameter values are returned.

Command	Abbrevia-tion	Explanation

perfmon [*options*][*subsystem prefix*]	pmon	Returns performance statistics from the server. Options are:
		■ **-c** returns all values as the counter
		■ **-d** returns short descriptions
		■ **-v** returns current values
		■ **-t** returns total values
		By default, some values are averages/second.
		The subsystem prefix is used to find matches to value names. Only those values are returned that match the subsystem prefix.
		The following example returns all information:
		`perfmon`
		The following example returns all values whose name starts with prefix **File** as counters.
		`perfmon-c`

pid	pid	Returns server process id
protocols	prot	Returns list of available communication protocols, one row for each protocol.
report *filename*	rep	Generates a report of server info to a file given as an argument.
shutdown	sd	Stops SOLID *Embedded Engine*.
status	sta	Displays server statistics.
status backup	sta bak	Displays status of the last started backup. The status can be one of the following: ■ If the last backup was successful or any back-ups have not been requested, the output is 0 SUCCESS. ■ If the backup is in process; for example, started but not ready yet, the output is 14003 ACTIVE. ■ If the last backup failed, the output is: *errorcode* ERROR where the *errcode* shows the reason for the fail-ure.

Command	Abbrevia-tion	Explanation
throwout {*username* | *userid* | **all }**	to	Exits users from SOLID *Embedded Engine*. To exit a specified user, give the user id as an argument. To throw out all users, use the keyword ALL as an argument.
trace {**on** | **off**} **sql** | **rpc** | **sync**	tra	Sets server trace on or off. This command is similar to the **monitor** command, but traces different entities and a different levels. By default, the output is written to the SOLTRACE.OUT file.
userid	uid	Returns user identification number of the current connection.
userlist [**-l**] [*name* | *id*]	ul	Displays a list of users. option -l displays more detailed output.
version	ver	Displays server version info.

SOLID *SQL Editor* (teletype)

With SOLID *SQL Editor* (teletype), SQL statements (including the SQL ADMIN COMMANDs) can be issued at the command line, command prompt, or by executing a script file that contains the SQL statements. For a formal definition of SQL statements and a list of ADMIN COMMANDs, see *"ADMIN COMMAND"* on page E-1 in this guide. To access a short description of available ADMIN COMMANDs, including short abbreviations, execute:

```
ADMIN COMMAND 'help'.
```

▶ **Note**

The user performing SQL statements must have appropriate user rights on the corresponding tables, or the connection will be refused.

Starting SOLID *SQL Editor* (teletype)

Start SOLID *SQL Editor* by issuing the command `solsql` or `load solsql` (Novell Netware) at the operating system prompt.

You can also specify the following syntax and include these optional command line arguments:

solsql *options servername username password*

where *options* can be:

-a	Auto commit every statement
-c*dir*	Change working directory
-e*sql-string*	Execute the SQL string; if used commit can only be done using -a.
-f*filename*	Execute SQL string from a script file
-h, -?	Help = Usage
-o*filename*	Write result set to this file
-O*filename*	Append result set to this file
-s*schema_name*	Use only this schema
-t	Print execution time per command
-u	Expect input in UTF-8 format
-x onlyresults	Print only rows

Note

If the user name and password are specified at the command line, the server name must also be specified. Also if the name of the SQL script file is specified at the command line (not with the -f option), the server name, user name, and password must also be specified. Remember to commit work at the end of the SQL script or before exiting *SQL Editor.*

Servername is the network name of a Solid server that you are connected to. Logical Data Source Names can also be used with tools; Refer to Chapter 6,“Managing Network Connections” for further information. The given network name must be enclosed in quotes.

Username is required to identify the user and to determine to determine the user’s authorization. Without appropriate rights, execution is denied.

Password is the user’s password for accessing the database.

SOLID *SQL Editor* connects to the first server specified in the `Connect` parameter in the solid.ini file. If you specify no arguments, you are prompted for the database administrator’s user name and password.

When there is an error in the command line, the SOLID *SQL Editor* gives you a list of the possible options as a result. Please be sure to check the command line you entered.

To exit *SQL Editor*, enter the command `exit`.

Examples

Assuming that a database connection is established, this example at the command prompt, executes the SQL statements terminated by a colon:

```
create table testtable (value integer, name varchar);
```

commit work;

Start *SQL Editor* and execute the tables.sql script:

```
solsql “tcp localhost 13113” admin iohe47 tables.sql
```

Executing SQL Statements with SOLID *SQL Editor* (teletype)

After the connection to the server has been established a command prompt appears. SOLID *SQL Editor* (teletype) executes SQL statements terminated by a semicolon.

Example:

```
create table testtable (value integer, name varchar);
commit work;

insert into testtable (value, name) values (31, 'Duffy Duck');
select value, name from testtable;
commit work;

drop table testtable;
commit work;
```

Executing a SQL Script from a File

To execute a SQL script from a file, the name of the script file must be given as a command line parameter:

solsql *servername username password filename*

All statements in the script must be terminated by a semicolon. SOLID *SQL Editor* (teletype) exits after all statements in the script file have been executed.

Example:

```
solsql "tcp localhost 1313" admin iohe4y tables.sql
```

▶ **Note**

Remember to commit work at the end of the SQL script or before exiting SOLID *SQL Editor* (teletype). If a SQL-string is executed with the option -e, commit can only be done using the -a option.

SOLID *SpeedLoader*

SOLID *SpeedLoader* is a tool for loading data from external ASCII files into a SOLID database. SOLID *SpeedLoader* can load data in a variety of formats and produce detailed information of the loading process into a log file. The format of the import file, that is, the file containing the external ASCII data, is specified in a control file.

The data is loaded into the database through the SOLID *SynchroNet* program. This enables online operation of the database during the loading. The data to be loaded does not have to reside in the server computer.

Control File

The control file provides information on the structure of the import file. It gives the following information:

- name of the import file
- format of the import file
- table and columns to be loaded

▶ **Note**

Each import file requires a separate control file. SOLID *SpeedLoader* loads data into one table at a time.

For details about control file format, read *"Control File Syntax"* on page 5-16. Please note the following:

- The table must exist in the database in order to perform data loading.
- Catalog support is available in SOLID *Speedloader.* The following syntax is supported:

  catalog_name.schema_name.table_name

Import File

The import file must be of ASCII type. The import file may contain the data either in a fixed or a delimited format:

- In fixed-length format data records have a fixed length, and the data fields inside the records have a fixed position and length.
- In delimited format data records can be of variable length. Each data field and data record is separated from the next with a delimiting character such as a comma (this is what SOLID *Export* produces). Fields containing no data are automatically set to NULL.

Data fields within a record may be in any order specified by the control file. Please note the following:

- Data in the import file must be of a suitable type. For example, numbers that are presented in a float format cannot be loaded into a field of integer or smallint type.
- Data of varbinary and long varbinary type are hexadecimal encoded in the import file.

Message Log File

During loading, SOLID *SpeedLoader* produces a log file containing the following information:

- Date and time of the loading
- Loading statistics such as the number of rows successfully loaded, the number of failed rows, and the load time if it has been specified with the option
- Any possible error messages

If the log file cannot be created, the loading process is terminated. By default the name of the log file is generated from the name of the import file by substituting the file extension of the import file with the file extension `.log`. For example, `my_table.ctr` creates the log file `my_table.log`. To specify another kind of file name, use the option -l.

Configuration File

A configuration file is not required for SOLID *SpeedLoader.* The configuration values for the server parameters are included in the SOLID *SynchroNet* configuration file `solid.ini`.

Client copies of this file can be made to provide connection information required for SOLID *Speedloader.* If no server name is specified in the command line, SOLID *SpeedLoader* will choose the server name it will connect to from the server configuration file. For example to connect to a server using the NetBIOS protocol and with the server name SOLID, the following lines should be included in the configuration file:

```
[Com]
Connect=netbios SOLID
```

Starting SOLID *SpeedLoader*

Start SOLID *SpeedLoader* with the command `solload` followed by various argument options. If you start SOLID *SpeedLoader* with no arguments, you will see a summary of the arguments with a brief description of their usage. The command line syntax is:

solload *options servername username password control_file*

where *options* can be:

-C*catalog_name*	Set the default catalog from where data is read from or written to
-s*schema_name*	Set the default schema
-b*records*	Number of records to commit in one batch
-c*dir*	Change working directory
-l*filename*	Write log entries to this file
-L*filename*	Append log entries to this file
-***n****records*	Insert array size (network version)
-t	Print load time
-x **emptytable**	Load data only if there are no rows in the table
-x **errors**:*count*	Maximum error count
-x nointegrity	No integrity checks during load (standalone version)
-x skip:*records*	Number of records to skip
--?	Help = Usage

For details on *control_file*, read following section.

Servername is the network name of a Solid server that you are connected to. Logical Data Source Names can also be used with tools; Refer to Chapter 6,“Managing Network Connections” for further information. The given network name must be enclosed in quotes.

Username is required to identify the user and to determine to determine the user’s authorization. Without appropriate rights, execution is denied.

Password is the user’s password for accessing the database.

When there is an error in the command line, the SOLID *SpeedLoader* gives you a list of the possible options as a result. Please be sure to check the command line you entered.

Control File Syntax

The control file syntax has the following characteristics:

- keywords must be given in capital letters
- comments can be included using the standard SQL double-dash (--) comment notation
- statements can continue from line to line with new lines beginning with any word

SOLID *SpeedLoader* reserved words must be enclosed in quotes if they are used as data dictionary objects, that is, table or column names. The following list contains all reserved words for the SOLID *SpeedLoader* control file:

AND	ANSI
APPEND	BINARY
BLANKS	BY
CHAR	CHARACTERSET
DATA	DATE
DECIMAL	DOUBLE
ENCLOSED	ERRORS
FIELDS	FLOAT
IBMPC	INFILE
INSERT	INTEGER
INTO	LOAD
LONG	MSWINDOWS
NOCNV	NOCONVERT
NULLIF	NULLSTR
NUMERIC	OPTIONALLY
OPTIONS	PCOEM
POSITION	PRECISION
PRESERVE	REAL
REPLACE	SCAND7BIT
SKIP	SMALLINT
TABLE	TERMINATED

TIME	TIMESTAMP
TINYINT	VARBIN
VARCHAR	WHITESPACE

The control file begins with the statement LOAD DATA followed by several statements that describe the data to be loaded. Only comments or the OPTIONS statement may optionally precede the LOAD DATA statement.

Full Syntax of the Control File

Syntax Element	**Definition**
control_file	::= [*option_part*] *load_data_part into_table_part*
option_part	::= OPTIONS (*options*)
options	::= *option* [, *option*]
option	::= [SKIP = '*int_literal*'] | [ERRORS = '*int_literal*']
load_data_part	::= LOAD [DATA] [*characterset_specification*] [DATE *date_mask*] [TIME *time_mask*] [TIMESTAMP *timestamp_mask*] [INFILE *filename*] [PRESERVE BLANKS]
characterset_specification	::= CHARACTERSET { NOCONVERT | NOCNV | ANSI | MSWINDOWS | PCOEM | IBMPC | SCAND7BIT }
into_table_part	::= INTO TABLE *tablename* [APPEND | INSERT | REPLACE] [FIELDS TERMINATED BY { WHITESPACE | *hex_literal* |'char']} [FIELDS [OPTIONALLY] ENCLOSED BY {"char"'| *hex_literal*} [AND "char" | *hex_literal*]] (*column_list*)
hex_literal	::= X'hex_byte_string'
column_list	::= *column* [, *column*]
column	::= *column_name datatype_spec* [POSITION ('*int_literal*' {: | -} '*int_literal*')] [DATE *date_mask*] [TIME *time_mask*] [TIMESTAMP *timestamp_mask*] [NULLIF BLANKS | NULLIF NULLSTR| NULLIF '*string*' | NUL-LIF (('*int_literal*' {: | -} '*int_literal*') = '*string*')]

Syntax Element	**Definition**
datatype_spec	::= {BINARY | CHAR [*length*] | DATE | DECIMAL [(*precision* [, *scale*])] | DOUBLE PRECISION | FLOAT [(precision)] | INTEGER | LONG VARBINARY | LONG VARCHAR | NUMERIC [(*precision* [, *scale*])] | REAL | SMALL-INT | TIME | TIMESTAMP [(*timestamp precision*)] | TINYINT | VARBINARY | VARCHAR [(*length*)] }

The following paragraphs explain syntax elements and their use is in detail.

CHARACTERSET

The CHARACTERSET keyword is used to define the character set used in the input file. If the CHARACTERSET keyword is not used or if it is used with the parameter NOCONVERT or NOCNV, no conversions are made. Use the parameter ANSI for the ANSI character set, MSWINDOWS for the MS Windows character set, PCOEM for the ordinary PC character set, IBMPC for the IBM PC character set, and SCAND7BIT for the 7-bit character set containing Scandinavian characters.

DATE, TIME, and TIMESTAMP

These keywords can be used in two places with different functionality:

- When one of these keywords is used as a part of the load-data-part element, it defines the format used in the import file for inserting data into any column of that type.
- When a keyword appears as a part of a column definition it specifies the format used when inserting data into that column.

Notes

1. Masks used as part of the load-data-part element must be in the following order: DATE, TIME, and TIMESTAMP. Each is optional.
2. Data must be of the same type in the import-file, the mask, and the column in the table into which the data is loaded.

Data Masks

Data Type	Available Data Masks
DATE	YYYY/YY-MM/M-DD/D
TIME	HH/H:NN/N:SS/S
TIMESTAMP	YYY/YY-MM/M-DD/D HH/H:NN/N:SS/S

In the above table, year masks are YYYY and YY, month masks MM and M, day masks DD and D, hour masks HH and H, minute masks NN and N, and second masks SS and S. Masks within a date mask may be in any order; for example, a date mask could be ‘MM-DD-YYYY’. If the date data of the import file is formatted as 1995-01-31 13:45:00, use the mask YYYY-MM-DD HH:NN:SS.

PRESERVE BLANKS

The PRESERVE BLANKS keyword is used to preserve all blanks in text fields.

into_table_part

The *into_table_part* element is used to define the name of the table and columns that the data is inserted into.

FIELDS TERMINATED BY

The FIELDS TERMINATED BY keyword is used to define the character used to distinguish where fields end in the input file.

The ENCLOSED BY keyword is used to define the character that precedes and follows data in the input file.

POSITION

The POSITION keyword is used to define a field's position in the logical record. Both start and end positions must be defined.

NULLIF

The NULLIF keyword is used to give a column a NULL value if the appropriate field has a specified value. An additional keyword specifies the value the field must have. The keyword BLANKS sets a NULL value if the field is empty; the keyword NULL sets a NULL value if the field is a string 'NULL'; the definition '*string*' sets a NULL value if the field matches the string '*string*'; the definition '((start : end) = '*string*')' sets a NULL value if a specified part of the field matches the string '*string'*.

Loading Fixed-format Records

Examples of the control file when loading data from a fixed-format import file:

```
-- EXAMPLE 1
LOAD DATA
INFILE 'EXAMP1.DAT'
INTO TABLE SUPPLIERS (
NAME       POSITION(01:19) CHAR,
ADDRESS    POSITION(20:40) VARCHAR,
ID         POSITION(41:48) INTEGER )

-- EXAMPLE 2
OPTIONS (SKIP = 10, ERRORS = 5)
-- Skip the first ten records. Stop if
-- errorcount reaches five.
LOAD DATA
INFILE 'sample.dat'
-- import file is named sample.dat
INTO TABLE TEST1 (
ID         INTEGER POSITION(1-5),
ANOTHER_ID INTEGER POSITION(8-15),
DATE1      POSITION(20:29) DATE 'YYYY-MM-DD',
DATE2      POSITION(40:49) DATE 'YYYY-MM-DD' NULLIF NULL)
```

Loading Variable-length Records

Examples of the control file when loading data from a variable-length import file:

```
-- EXAMPLE 1
LOAD DATA
INFILE 'EXAMP2.DAT'
INTO TABLE SUPPLIERS
FIELDS TERMINATED BY ','
(NAME VARCHAR, ADDRESS VARCHAR, ID INTEGER)
-- EXAMPLE 2
OPTIONS (SKIP=10, ERRORS=5)
-- Skip the first ten records. Stop if
-- errorcount reaches five.
LOAD
DATE 'YYYY-MM-DD HH:NN:SS'
-- The date format in the import file
INFILE 'sample.dat'
```

```
-- The import file
INTO TABLE TEST1
-- data is inserted into table named TEST1
FIELDS TERMINATED BY X'2C'
-- Field terminator is HEX ',' == 2C
-- This line could also be:
-- FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '[' AND ')'
-- Fields may also be enclosed
-- with '[' and ')'
(
ID INTEGER,
ANOTHER_ID DECIMAL(2),
DATE1 DATE(20) DATE 'YYYY-MM-DD HH:NN:SS',
DATE2 NULLIF NULL
)
-- ID is inserted as integer
-- ANOTHER_ID is a decimal number with 2
-- digits.
-- DATE1 is inserted using the datestring
-- given above
-- The default datestring is used for DATE2.
-- If the column for DATE2 is 'NULL' a NULL is
-- inserted.
```

Running a Sample Load Using Solload

To Run a Sample Load Using Solload

1. Start SOLID *SynchroNet*.
2. Create the table using the `sample.sql` script and your SOLID *SQL Editor*.
3. Start loading using the following command line:

   ```
   solload "shmem solid" dba dba delim.ctr
   ```

   The user name and password are assumed to be 'dba'. To use the fixed length control file, use the following command line:

   ```
   solload "shmem solid" dba dba fixed.ctr
   ```

The output of a successful loading using `delim.ctr` will be:

```
SOLID Speed Loader v.3.00.00xx
```

```
(C) Copyright Solid Information Technology Ltd 1992-2000
Load completed successfully, 19 rows loaded.
```

The output of a successful loading using `fixed.ctr` will be:

```
SOLID Speed Loader v.3.00.00xx
(C) Copyright Solid Information Technology Ltd 1992-2000
Load completed successfully, 19 rows loaded.
```

Hints to Speed up Loading

The following hints can be used to ensure that loading is done with maximum performance:

- It is faster not to load data over the network, that is, connect locally if possible.
- Increasing the number of records committed in one batch speeds up the load. By default, commit is done after each record.
- Disable transaction logging.

To disable logging the LogEnabled parameter needs to be used. The following lines in the `solid.ini` file will disable logging:

```
[Logging]
LogEnabled=no
```

After the loading has been completed, remember to enable logging again. The following line in the `solid.ini` file will enable logging:

```
[Logging]
LogEnabled=yes
```

Note

Running the server with logging disabled is strongly discouraged. If logs are not written, no recovery can be made if an error occurs due to power failure, disk error etc.

SOLID *Export*

SOLID *Export* is a product for unloading data from a SOLID database to ASCII files. SOLID *Export* produces both the import file, that is, the file containing the exported ASCII data, and the control file that specifies the format of the import file. SOLID *SpeedLoader* can directly use these files to load data into a SOLID database.

Note

The user name used for performing the export operation must have select rights on the table exported. Otherwise no data is exported.

Starting SOLID *Export*

Start SOLID *Export* with the command `solexp`. If you start `solexp` with no arguments, you'll see a summary of the arguments with a brief description. The command line syntax is:

solexp *options servername username password tablename* **|** *

where *options* argument can be:

-C*catalog_name*	Set the default catalog from where data is read from or written to
-c*dir*	Change working directory
-e*sql-string*	Execute SQL string for export
-f*filename*	Execute SQL string from file for export
-h, **-?**	Help = Usage
-l*filename*	Write log entries to this file
-L*filename*	Append log entries to this file
-o*filename*	Write exported data to this file
-s*schema_name*	Use only this schema for export

Notes

1. The symbol * can be used to export all tables with one command. However, it cannot be used as a wildcard.
2. The **-t***tablename* (Export table) option is still supported in order to keep old scripts valid.

Servername is the network name of a Solid server that you are connected to. Logical Data Source Names can also be used with tools; Refer to Chapter 6,"Managing Network Connections" for further information. The given network name must be enclosed in quotes.

Username is required to identify the user and to determine to determine the user's authorization. Without appropriate rights, execution is denied.

Password is the user's password for accessing the database.

When there is an error in the command line, the SOLID *Export* gives you a list of the possible options as a result. Please be sure to check the command line you entered.

SOLID *Data Dictionary*

SOLID *Data Dictionary* is a product for retrieving data definition statements from a SOLID database. SOLID *Data Dictionary* produces a SQL script that contains data definition statements describing the structure of the database. The generated script contains definitions for tables, views, procedures, sequences, and events.

Notes

1. User and role definitions are not listed for security reasons.
2. The user name used for performing the export operation must have select right on the tables. Otherwise the connection is refused.

Starting SOLID *Data Dictionary*

Start SOLID *Data Dictionary* with the command `soldd`. If you invoke `soldd` with no arguments, you'll see a summary of the arguments with a brief description. The command line syntax is:

soldd *options servername username password* [*tablename*]

where *options* can be:

-C*catalog_name*	Set the default catalog from where data definitions are read from or written to
-c*dir*	Change working directory
-h, -?	Help = Usage
-o*filename*	Write data definitions to this file
-O*filename*	Append data definitions to this file
-s*schema_name*	List definitions from this schema only
-x eventonly	List event definitions only
-x indexonly	List index definitions only
-x procedureonly	List procedure definitions only
-x sequenceonly	List sequence definitions only
-x tableonly	List table definitions only
-x viewonly	List view definitions only

Servername is the network name of a Solid server that you are connected to. Logical Data Source Names can also be used with tools; Refer to Chapter 6,"Managing Network Connections" for further information. The given network name must be enclosed in quotes.

Username is required to identify the user and to determine to determine the user's authorization. Without appropriate rights, execution is denied.

Password is the user's password for accessing the database.

When there is an error in the command line, the SOLID *Data Dictionary* gives you a list of the possible options as a result. Please be sure to check the command line you entered.

Examples:

```
soldd -odatabase.sql "tcp database_server 1313" dbadmin f1q32j4
```

Print the definition of procedure TEST_PROC:

```
soldd -x procedureonly " " dba dba TEST_PROC
```

Notes

1. If no table name is given, all definitions are listed to which the user has rights.
2. If the *objectname* parameter is provided with one of the **-x** options, the name is used to print only the definition of the named object.
3. The **-t***tablename* option is still supported in order to keep old scripts valid.

Tools Sample: Reloading a Database

This example demonstrates how a SOLID *Embedded Engine* database can be reloaded to a new one. At the same time the use of each SOLID tool is introduced with an example. This reload is a useful procedure since it shrinks the size of the database file `solid.db` to a minimum.

To Reload the Database:

1. Extract data definitions from the old database.
2. Extract data from the old database.
3. Replace the old database with a new one.
4. Load data definitions into a new database.
5. Load data into the new database.

Walkthrough

In this example the server name is SOLID and the protocol used for connections is Shared Memory. Therefore, the network name is “ShMem SOLID”. The database has been created with the user name “dbadmin” and the password “password”.

1. Data definitions are extracted with SOLID *Data Dictionary*. Use the following command line to extract a SQL script containing definitions for all tables, views, procedures, sequences, and events. The default for the extracted SQL file is `soldd.sql`.

   ```
   soldd "ShMem SOLID" dbadmin password
   ```

   With this command all data definitions are listed into one file, `soldd.sql` (the default name). As mentioned earlier, user and role definitions are not listed for security reasons. If the database contains users or roles, they need to be appended into this file.

2. All data is extracted with SOLID *Export*. The export results in control files (files with the extension `.ctr`) and data files (files with the extension `.dat`). The default file name is the same as the exported table name. In 16-bit environments, file names longer than eight letters are concatenated. Use the following command line to extract the control and data files for all tables.

   ```
   solexp "ShMem SOLID" dbadmin password *
   ```

   With this command data is exported from all tables. Each table’s data is written to an import file named `table_name.dat`. A separate control file `table_name.ctr` is written for each table name.

3. A new database can be created to replace the old one by deleting the `solid.db` and all `sol####.log` files from the appropriate directories. When SOLID *Embedded Engine* is started for the first time after this, a new database is created.

Note

It is recommended that a backup is created of the old database before it is deleted. This can be done using SOLID *Remote Control* (teletype).

4. Use the following command line to create a backup using SOLID *Remote Control* (teletype):

   ```
   solcon -eBACKUP "ShMem SOLID" dbadmin password
   ```

   With this command, a backup is created. The option -e precedes an administration command.

5. Load data definitions into the new database. This can be done using SOLID *SQL Editor* (teletype). Use the following command line to execute the SQL script created by SOLID *Data Dictionary***.**

```
solsql -fSOLDD.SQL "ShMem SOLID" dbadmin password
```

With this command, data definitions are loaded into the new, empty database. Definitions are retrieved with the option -f from the file `soldd.sql`. Connection parameters are the same as in the earlier examples.

The previous two steps can be performed together by starting SOLID *Embedded Engine* with the following command line. The option -x creates a new database, executes commands from a file, and exits. User name and password are defined as well.

```
solid -Udbadmin -Ppassword -x execute:soldd.sql
```

6. Load data into the new database. This is be done SOLID *Speedloader.* To load several tables into the database a batch file containing a separate command line for each table is recommended. In Unix-based operating systems, using the wildcard symbol * is possible. Use either of the following command lines to load data into the new database.

```
solload "ShMem SOLID" dbadmin password table_name.ctr
```

7. With this command, data for one table is loaded. The server is online.

Batch files that can be used are:

- Shell scripts in Unix environments
- `.com` -scripts in VMS
- `.bat` -scripts in Windows 95/98/2000 and NT

6

Managing Network Connections

As a true client/server DBMS, SOLID *SynchroNet* provides simultaneous support for multiple network protocols and connection types. Both the database server and the client applications can be simultaneously connected to multiple sites using multiple different network protocols.

This chapter describes how to set up network connections for each of the supported platforms.

▶ **Note**

Some platforms may limit the number of concurrent users to a single SOLID server process even if the SOLID license accepts higher limits. Refer to the Release Notes for details that apply to your specific operating system.

Communication between Client and Server

The database server and client transfer information between each other through the computer network using a communication protocol.

When a database server process is started, it will publish at least one network name that distinguishes it in the network. The server starts to *listen* to the network using the given network name. The network name consists of a communication protocol and a server name.

To establish a connection from a client to a server they both have to be able to use the same communication protocol. The client has to know the network name of the server and often also the location of the server in the network. The client process uses the network name to specify which server it will *connect* to.

This chapter will give you information on how to administer network names.

Managing Network Names

The network name of a server consists of a *communication protocol* and a *server name*. This combination identifies the server in the network. The network names are defined with the `Listen` parameter in the `[Com]` section of the configuration file. The `solid.ini` file should be located in a SOLID *SynchroNet* program’s working directory or in the directory set by the `SOLIDDIR` environment variable.

A server may use an unlimited number of network names. Note that all components of network names are case insensitive.

Network names are managed in the following ways:

- Using the Protocols page in the SOLID *DBConsole* accessed through the Administration window or menu.
- Using the command `protocols` in SOLID *Remote Control* (teletype) or *SOLID DBConsole*.
- Using the SOLID SQL syntax ADMIN COMMAND ’protocols’ in SOLID *SQL Editor* (teletype).
- Directly, by editing the server configuration file `solid.ini`.

An example of an entry in `solid.ini` is:

```
[Com]
Listen = tcpip 1313, nmpipe solid
```

The example contains two network names which are separated by a comma. The first one uses the protocol TCP/IP and the service port 1313, the other one uses the Named Pipes protocol with the name ‘solid’. In our example the ‘tcpip’ and ‘nmpipe’ are communication protocols while ‘1313’ and ‘solid’ are server names.

If the `Listen` parameter is not set in the `solid.ini` file, the environment dependent defaults are used.

Note

1. When a database server process is started it publishes the network names it starts to listen to. This information is also written to a file named `solmsg.out` located in the same directory as the `solid.ini` file.
2. Network names must be unique within one host computer. For example, you cannot have two database servers running, both listening to the same TCP/IP port in one host, but it is possible that the same port number is in use in different hosts. Exceptions to this

are the NetBIOS and IPX/SPX protocols, which require that used server names are unique throughout the whole network.

Viewing Supported Protocols for the Server

Because not all protocols are supported in all environments and operating systems, you can view the protocol options available for your server.

To view supported protocols for a server:

Enter the command `protocols` in SOLID *DBConsole* or SOLID *Remote Control* (teletype).

-or-

Enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

```
ADMIN COMMAND 'protocols'
```

A list of all available communication protocols is displayed. The command provides the following kind of result set, which contains one row for each supported communication protocol:

```
RC       TEXT
--       ----
0        tc TCP/IP
0        nb NetBIOS
```

Viewing Network Names for the Server

Following are ways that you can view network names for the server:

- Select the **Status** option from the SOLID *DBConsole* Administration window or menu and click the **Protocols** icon to view the network names listed in the Protocols dialog box.
- View the parameter `Listen` in the `[Com]` section in the `solid.ini` file.
- Enter the command `parameter com.listen` in SOLID *DBConsole* or SOLID *Remote Control* (teletype).

  -or-

  Enter the following SOLID SQL syntax in the SOLID *SQL Editor* (teletype):

  ```
  ADMIN COMMAND 'parameter com.listen'
  ```

A list of all network names for the server is displayed. The command provides the following kind of result set, which contains one row for each supported communication protocol:

```
RC                  TEXT
--                  ----
0                   TCP/IP 1313
0                   NetBIOS Solid
```

Adding and Modifying a Network Name for the Server

Following are ways you can add and edit network names for a server, which consists of a *communication protocol* and a *server name*; for example, `nmpipe solid.`

- Select the **Status** option from the SOLID *DBConsole* Administration window or menu and click the **Protocols** icon to add or modify the network names in the Protocols dialog box.
- To add network names for the server:

  Enter the command `parameter com.listen=`*`network_name`* in SOLID *DBConsole* or SOLID *Remote Control* (teletype).

  -or-

  Enter the following SOLID SQL syntax in SOLID *SQL Editor* (teletype):

  ```
  ADMIN COMMAND 'parameter com.listen=network_name'
  ```

  The command returns the new value as the resultset. If the network name entered is invalid, the ADMIN COMMAND statement returns an error.
- In `solid.ini`, locate the working directory of your SOLID *SynchroNet* process and add a new network name or edit an existing one as a part of the `Listen` parameter entry in the `[Com]` section.

  Use a comma (,) to separate network names. For example:

  ```
  [Com]
  Listen = tcpip 1313, nmpipe solid
  ```

  Be sure to save the changes and to restart the SOLID *SynchroNet* process to activate the changes.

To Remove a Network Name from the Server

Following are ways you can remove network names for a server, which consists of a *communication protocol* and a *server name*; for example, `nmpipe solid.`

- Select the **Status** option from the SOLID *DBConsole* Administration window or menu and click the **Protocols** icon to remove the network name in the Protocols dialog box.
- In `solid.ini`, locate the working directory of your SOLID *SynchroNet* process and remove the network name in the `Listen` parameter entry in the `[Com]` section.

  Be sure to save the changes and to restart the SOLID *SynchroNet* process to activate the changes.

Note

The modifications to network names do not become active immediately after editing the `solid.ini` file. You must restart the SOLID *SynchroNet* process.

You can disable a network name using option **-d** after the protocol name in the network name:

```
tcp -d hobbes 1313, shmem -d solid
```

Network Name for Clients

The network name of a client consists of a *communication protocol*, an optional *host computer name* and a *server name.* By this combination the client specifies the server it will establish a connection to. The communication protocol and the server name must match the ones that the server is using in its network listening name. In addition, most protocols need a specified host computer name if the client and server are running on different machines. All components of the client's network name are case insensitive.

The client's network names are defined in the configuration file `solid.ini` in the `[Com]` section with the `Connect` parameter. The `solid.ini` file should be located in the application program's working directory or in the directory set by the `SOLIDDIR` environment variable.

The following connect line in the `solid.ini` of the application workstation will connect an application (client) using the TCP/IP protocol to a SOLID server running on a host computer named ‘spiff’ and listening with the name (port number in this case) ‘1313’.

```
[Com]
```

```
Connect = tcpip spiff 1313
```

If the `Connect` parameter is not found in the configuration file `solid.ini` the client uses the environment dependent default instead. The defaults for the `Listen` and `Connect` parameters are selected so that the application (client) will always connect to a local SOLID server listening with a default network name. So the local communication (inside one machine) does not necessarily need a configuration file for establishing a connection.

▶ **Note**

1. When the connection is requested by client application using the `SQLConnect` function the network name of the server is given as a Data Source Name parameter for that function. If the given name is not an empty string, its contents are used as a network name and the `Connect` parameter in the configuration file is omitted. If an empty string is passed, the possibly existing `Connect` parameter is used.
2. In the Windows (95, 98, 2000, NT) operating system, the connection can be made by using the SOLID *ODBC Driver.* When a client program is using the SOLID *ODBC Driver*, the network name of the server can be used as the ODBC Data Source Name and the `Connect` parameter in the configuration file is not used

Communication Protocols

A client process and SOLID *SynchroNet c*ommunicate with each other by using computer networks and network protocols. A network operating system—for example, IBM LAN Server or Novell NetWare—is not necessarily needed. You only need a functioning communication protocol for both ends. Supported communication protocols depend on the type of computer and network you are using.

The following paragraphs describe the supported communication protocols and common environments that may be used and also show the required forms of network names for the various protocols.

▶ **Note**

Depending on your network protocol, there may be relevant communication parameters associated with the protocol. Be sure to use ADMIN COMMAND ’parameter’ in the SOLID *DBConsole* Query window to find the communication parameters in use. Then you can use ADMIN COMMAND ’describe parameter’ to view details on the specific communication parameter. See *"Managing Parameters"* on page 7-7 for details on these commands.

Shared Memory

Usually the fastest way two processes can exchange information is to use Shared Memory. This can be used only when the embedded engine and application processes are both running in the same computer. The Shared Memory protocol uses a shared memory location for moving data from one process to another.

To use the Shared Memory protocol in SOLID *Embedded Engine*, select `ShMem` from the list of protocols in SOLID *DBConsole* and enter server name. The server name has to be unique only in this computer.

Format Used in the solid.ini File

Server	`Listen = shmem` *`servername`*
Client	`Connect = shmem` *`servername`*

Note

Server names must be character strings less than 128 characters long.

TCP/IP

The TCP/IP protocol is typically used for communicating to a server process running under a UNIX operating system. When starting a server using the TCP/IP protocol, you must reserve a port number for it. You will find reserved port numbers in the `/etc/services` file of your system. Select a free number greater than 1024 since smaller numbers are usually reserved for the operating system.

To use the TCP/IP protocol, select `TCP/IP` in the list of protocols in SOLID *DBConsole* and enter a non-reserved port number.

Format Used in the solid.ini File

Server	`Listen = tcpip` *`server_port_number`*
Client	`Connect = tcpip [`*`host_computer_name`*`]` *`server_port_name`*

▶ Note

1. If the server is running in the same computer with the client program, the host computer name need not be specified. The client computer has to have the used host name listed in its `etc/hosts` file or it must be recognized by the DNS (Domain Name Server). You can also give the host computer's TCP/IP address in dotted decimal format (for example, 194.53.94.97) instead of its host name.
2. On Windows (95, 98, 2000, NT) and UNIX the TCP/IP protocol is usually included in the operating system. On other environments (like VAX/VMS) the TCP/IP software needs to be installed on the system. For a list of supported TCP/IP software, contact Solid Information Technology Ltd. at: `http://www.solidtech.com/`
3. The local loopback interface address, 127.0.0.1, is the default address when a client attempts to open a TCP/IP connection without specifying a hostname.
4. Using option **`-i`***`ip_address`* or **`-i`***`host_name`* SOLID *SynchroNet* listens only to the specified IP-address or host name. This is useful in multi-homed systems that support many TCP/IP interfaces (or have multiple ip-addresses). For example, a server with the following setting in `solid.ini` accepts connection requests only from inside the same machine, either referred by IP-address 127.0.0.1 or with the name 'localhost', if the DNS is correctly configured:

   ```
   [com]
   Listen = tcp -i127.0.0.1 1313
   ```

   Note that DNS entries can be used instead of IP-addresses, for example:

   ```
   [com]
   Listen = tcp -ilocalhost 1313
   ```
5. Using option -i127.0.0.1, which starts the server to listen only to a local loopback connection, allows TCP/IP listening with a desktop license. To enable TCP/IP usage with desktop licenses, all entries in `solid.ini` have to be edited to include **-i**. Note that default listening of port 1313 (without `solid.ini`) works automatically.

UNIX Pipes

The UNIX domain sockets (UNIX Pipes, Named Pipes, portals) are typically used when communicating between two processes running in the same UNIX machine. UNIX Pipes usually have a very good throughput. They are also more secure than TCP/IP since Pipes can only be accessed from applications that run on the computer where the server executes.

When starting a server using UNIX Pipes, you must reserve a unique listening name (inside that machine) for the server, for instance, 'solid'. Because UNIX Pipes handle the UNIX domain sockets as standard file system entries, there is always a corresponding file created for every listened pipe. In SOLID*SynchroNet's* case, the entries are created under the path '/tmp'. Our example listening name 'solid' creates the directory '/tmp/solunp_SOLID' and shared files into that directory. The '/tmp/solunp_' is a constant prefix for all created objects while the latter part ('SOLID' in this case) is the listening name in upper case format.

Format Used in the solid.ini File

Server	`Listen = upipe` *`server_name`*
Client	`Connect = upipe` *`server_name`*

Note

1. Server and client processes must run in the same machine in order to use UNIX Pipes for communication.
2. The server process must have a “write” permission to the directory '/tmp'.
3. The client accessing UNIX Pipes must have an “execute” permission to the directory '/tmp'.
4. The directory '/tmp' must exist.
5. UNIX Pipes cannot be used in SCO UNIX.

NetBIOS

The NetBIOS protocol is commonly used in the Windows (95, 98, 2000, NT) operating systems.

To use NetBIOS protocol, select `NetBIOS` in the list of available protocols in SOLID *DBConsole* and enter a non-reserved server name.

Format Used in the solid.ini File

Server	`Listen = netbios [a`*`LANA_NUMBER`*`]` *`server_name`*
Client	`Connect = netbios [a`*`LANA_NUMBER`*`]`*`server_name`*

▶ **Note**

1. The server name must be a character string at most 16 characters long. It may not begin with an asterisk (*).
2. In the above format the optional -aLANA_NUMBER is used to override the default value of the LANA number.
3. In Windows NT the available LANA numbers can be checked using the Network Setup found in the Control Panel. The default value 0 may not be generally very good. You should choose the one(s) where the protocol stack matches the other computers you are using. The LANA number (Network Route: Nbf->Elnk3->Elnk31) that uses NetBEUI as a transport usually functions quite smoothly when used for SOLID communication.
4. The server names have to be unique in the whole network. Establishing a connection or starting the listener using the NetBIOS protocol may be somewhat slow because of the checks needed for uniqueness.
5. SOLID *SynchroNet* and SOLID Client versions 2.2 and newer use all available LANA numbers by default. This makes it unnecessary to specify explicitly which LANA number the application or embedded engine should use. For backward compatibility the parameter '-aLANA_NUMBER' remains available.

Named Pipes

Named Pipes is a protocol commonly used in the Windows (95, 98, 2000, NT) operating systems.

Windows (95/98) support Named Pipes only in client end communication. Windows NT and 2000 supports Named Pipes both in server and client communication.

Format Used in the solid.ini File

Server	`Listen = nmpipe` *`server_name`*
Client	`Connect = nmpipe [`*`host_computer_name`*`]` *`server_name`*

▶ **Note**

1. The server names must be character strings at most 50 characters long.
2. If the server is running in the same computer with the application program, the host computer name should not be specified.
3. In order to connect to the SOLID *SynchroNet* for Windows NT through Named Pipes, the user must have at least the same rights as the user who started the server. For example if an administrator starts the server only, users with administrator's rights are able to connect to the server through Named Pipes. Similarly if a user with normal user's rights starts the server all users with greater rights are able to connect the server through Named Pipes. If a user doesn't have proper rights, SOLID Communication Error 21306 message will be given.
4. It is not recommended to use the Named Pipes communication from SOLID *Remote Control*. The asynchronous nature of SOLID *Remote Control* communication may cause problems with Named Pipes.

DECnet

The DECnet protocol is used to connect to an embedded engine running on an OpenVMS system. To use this protocol in Windows (95, 98, 2000, NT) you need to have PATHWORKS 32 installed on your client computer.

To use the DECnet protocol, select `DECnet` in the list of protocols in SOLID *DBConsole* and enter a non-reserved server name.

Format Used in the solid.ini File

Server	`Listen = decnet` *`server_name`*
Client	`Connect = decnet` *`node_name server_name`*

▶ **Note**

To establish a connection the DECnet node name of the server machine is configured to your node database. The node name can be given either as a node number such as ‘1.1’ or as a node name such as ‘VAX1’.

IPX/SPX

The IPX/SPX protocol is used to communicate with SOLID *SynchroNet* for Novell Netware.

SOLID *SynchroNet* for Novell Netware starts listening with the default listening name `SOLID` if no listening name is specified in the configuration file `solid.ini`. When SOLID *SynchroNet* starts, it prints out the network and node information of the server machine.

The SOLID server listening name can be given as a character string or as a socket number. If the given network name is a valid socket number, that is, hex number with exactly 4 characters (for example, 400F) SOLID *SynchroNet* starts listening in the given port. If the network name could not be interpreted as a socket number it is treated as a server name character string and is published using Novell NetWare SAP (Service Advertising Protocol).

Connecting to a SOLID *SynchroNet* using SAP requires that you specify only the correct server name in the `Connect` parameter. If the server is listening using some given port, the full NLM server info (see comment below) has to be given.

To use the IPX/SPX protocol, select `IPX/SPX` in the list of protocols in SOLID *Remote Control* and enter a non-reserved server name.

Format Used in the solid.ini File

Server	`Listen = spx {`*`servername`*` | `*`socket_number`*`}`
Client	`Connect = spx {NLM `*`server_info`*` | `*`server_name`*`}`

Note

1. The server names must be less than 48 characters long.
2. In the above format, NLM *server_info* stands for a string containing the network number, the node number, and the socket number separated by colons. For example, NLM *server_info* for network 1, node 1, socket number 1313 is 00000001:000000000001:1313. You can abbreviate the information by removing the leading zeros. The previous *SynchroNet* information could thus also be written as 1:1:1313. The *server_name* stands for an alphanumeric string.
3. The possibility to use socket numbers as the listening name is supported mainly for historical reasons. SAPing is intended to be the primary method.
4. After removing a network name or shutting down SOLID *Embedded Engine* using SOLID *DBConsole* or SOLID teletype tools, the server name used may still remain reserved for up to one minute although everything completes successfully. The error 'network name in use' is displayed if SOLID *Embedded Engine* is restarted immediately. This is a 'normal' NetWare SAP feature and happens more often if your network consists of more than one NetWare server. Propagating the SAP cancellation packets to every network node may take a while.

A Summary of Protocols

The following tables summarize the possible operating systems and required forms for network names for the various communication protocols.

Note

The following tables contain the protocols and operating systems that were available when this guide was printed. For an updated list, refer to the Solid Website at: `http://www.solidtech.com/`.

***SynchroNet* Protocols and Network Names**

Protocol	Server OS	Network name in solid.ini file
Shared Memory	Windows 95 Windows 98 Windows 2000 Windows NT	Listen = shmem *server*
NetBIOS	Windows 95 Windows 98 Windows 2000 Windows NT	Listen = netbios *server*
Named Pipes	Windows 2000 Windows NT	Listen = nmpipe *server*

IPX/SPX	Novell Netware	Listen = spx *server*
		Listen = spx *socket number*

TCP/IP	Windows 95 Windows 98 Windows 2000 Windows NT UNIX	Listen = tcpip *port*
UNIX Pipes	UNIX	Listen = upipe *server*

Application Protocols and Network Names

Protocol	Server OS	Network name in solid.ini file
Shared Memory	Windows 95 Windows 98 Windows 2000 Windows NT	Connect = shmem *server*
NetBIOS	Windows 95 Windows 98 Windows 2000 Windows NT	Connect = netbios *server*
Named Pipes	Windows 95 Windows 98 Windows 2000 Windows NT	Connect = nmpipe [host] *server*

IPX/SPX	Novell Netware
	Windows 95 [1]
	Windows 98 [1]
	Windows 2000 [1]
	Windows NT [1]
		Connect = spx *server*
		Connect = spx NLM *server_nfo*

TCP/IP	Windows 95 Windows 98 Windows 2000 Windows NT UNIX	Connect = tcpip [host] *port*
UNIX Pipes	UNIX	Connect = upipe *server*
DECnet	Windows 95 [2] Windows 98 [2] Windows 2000[2] Windows NT [2]	Connect = decnet *host server*

1) requires Novell's Netware Client for Windows 95/98/2000/NT.

2) requires Digital PATHWORKS 32 for Windows 95/98/2000/NT.

Logical Data Source Names

SOLID Clients support Logical Data Source Names. These names can be used for giving a database a descriptive name. This name can be mapped to a network name in three ways:

1. Using the parameter settings in the application’s `solid.ini` file.

2. Using the Windows operating systems registry settings.
3. Using settings in a solid.ini file located in the Windows directory.

This feature is available on all supported platforms. However, on non-Windows platforms, only the first method is available.

A SOLID Client attempts to open the file `solid.ini` first from the directory set by the `SOLIDDIR` environment variable. If the file is not found from the path specified by this variable or if the variable is not set, an attempt is made to open the file from the current working directory.

To define a Logical Data Source Name using the `solid.ini` file, you need to create a `solid.ini` file containing the section `[Data Sources]`. In that section you need to enter the 'logical name' and 'network name' pairs that you want to define. The syntax of the parameters is the following:

```
[Data Sources]
```

```
logical_name = network_name, Description
```

In the description field, you may enter comments on the purpose of this logical name.

If, for example, you want to define a logical name for the application 'My_application', and the database is located in a UNIX server that you want to connect to by using TCP/IP. You should include the following lines to the solid.ini file, which you need to place in the working directory of your application:

```
[Data Sources]

My_application = tcpip irix 1313, Sample data source
```

When your application now calls the Data Source 'My_application', the SOLID Client maps this to a call to 'tcpip irix 1313'.

On Windows platforms (95, 98, 2000, NT), the registry can be used to map Data Sources. These follow the standards of mapping ODBC Data Sources on a system.

In Windows 95, 98, 2000, and NT, a Data Source may be defined in the Windows Registry. The entry is searched from the path "software\odbc\odbc.ini"

1. first under the root HKEY_CURRENT_USER and if not found,
2. under the root HKEY_LOCAL_MACHINE.

The order of resolving a Data Source name in Windows systems is the following:

1. Look for the Data Source Name from the `solid.ini` file in the current working directory, under the section `[Data Source]`

2. Look for the Data Source Name from the following registry path `HKEY_CURRENT_USER\software\odbc\odbc.ini\DSN`
3. Look for the Data Source Name from the following registry path `HKEY_LOCAL_MACHINE\software\odbc\odbc.ini\DSN`

In case an application uses normal ODBC Data Sources, the network name is mapped normally using the methods that are provided in the ODBC Driver Manager.

7 Configuring SOLID *SynchroNet*

This chapter describes how to configure SOLID *SynchroNet* to meet your environment, performance, and operation needs. It includes the most important parameters and their settings. The section *"Managing Parameters"* on page 7-7 in this chapter gives you step-by-step instructions on how to view and set the parameter values in SOLID *DBConsole*, SOLID *Remote Control* (teletype), or *SQL Editor* (teletype).

Configuration File and Default Settings

When SOLID *SynchroNet* is started, it attempts to open the configuration file `solid.ini` first from the directory set by `SOLIDDIR` environment variable. If the file is not found from the path specified by this variable or if the variable is not set, an attempt is made to open the file from the current working directory.

The configuration file contains settings for the SOLID *SynchroNet* parameters. If the file does not exist, SOLID *SynchroNet* uses default settings for the parameters. Also, if a value for a specific parameter is not set in the `solid.ini` file, SOLID *SynchroNet* will use a default value for the parameter. The default values depend on the operating system you are using.

Generally, default settings offer good performance and operability, but in some cases modifying some parameter values can improve performance.

Most Important Parameters

This section describes the most important SOLID *SynchroNet* parameters and their default settings. Parameters are grouped according to section categories in the configuration file. See *Appendix C, "Configuration Parameters"* of this manual for a quick overview of the section categories and all available parameters.

Defining Network Names (Com section)

When a server is started, it will start listening to one or more protocols with network names that distinguish it in the network. A client application uses a similar network name to specify which protocol to use and which server to connect to.

Connect parameter

The `Connect` parameter in the `[Com]` section defines the network name for a client; this is the protocol and name that a client application uses for a server connection. Its default is Operating System dependent. Refer to *Chapter 6, "Managing Network Connections"* for details on the parameter format.

When an application program is using a SOLID *ODBC Driver* the ODBC Data Source Name is used and the `Connect` parameter has no effect. The `solid.ini` file, which includes the `Connect` parameter, must be located in the application program's working directory or in the directory set by `SOLIDDIR` environment variable.

The following connect line will connect a client program using the TCP/IP protocol to a SOLID server running in a computer named 'spiff' and server port number '1313'.

```
connect = tcpip spiff 1313
```

Listen parameter

The `Listen` parameter in the `[Com]` section defines the network name for the server; this is the protocol and name that a SOLID server uses when it starts to listen to the network. Its default is Operating System dependent. Refer to *Chapter 6, "Managing Network Connections"* for details on the parameter format.

Managing Database Files and Caching (IndexFile section)

In SOLID *SynchroNet* data and indexes are stored in the same logical files. The term 'index file' is used as a synonym for the term 'database file'. You also control the caching-related parameters in this section.

FileSpec_[1-N] parameter

The `FileSpec` parameter describes the location and the maximum size of the index file (database file). To define the location and maximum value the database file may reach, the `FileSpec` parameter accepts the following three arguments:

- database file name
- max filesize
- device number (optional)

You can also use the `FileSpec` parameter to divide the database file into multiple files and onto multiple disks. To do this, specify another `FileSpec` parameter identified by the number `2`. The index file will be written to the second file if it grows over the maximum value of the first `FileSpec` parameter. The default value for this parameter is solid.db, 2147483647 (which equals 2 GB expressed in bytes):

```
FileSpec_1=SOLID.DB 2147483647
```

In the following example, the parameters divide the database file on the disks C:, D: and E: to be split after growing larger than 1 GB (=1073741824 bytes).

```
FileSpec_1=c:\soldb\solid.1 1073741824  1
FileSpec_2=D:\soldb\solid.2 1073741824  2
FileSpec_3=G:\soldb\solid.3 1073741824  3
```

Note

The index file locations entered must be valid path names in the server’s operating system. For example, if the server runs on a UNIX operating system, path separators must be slashes instead of backslashes.

Although the database files reside in different directories, the file names must be unique. In the above example, it is assumed that C:, D: and E: partitions reside on separate physical disks.

Splitting the database file on multiple disks will increase the performance of the server because multiple disk heads will provide parallel access to the data in your database. There is no limit to the number of database files you may use.

If the database file is split into multiple physical disks, then multithreaded SOLID *Synchro-Net* is capable of assigning a separate disk I/O thread for each device. This way the server can perform database file I/O in a parallel manner. Read *"Dedicated Threads"* on page 1-8 for more details.

CacheSize

The `CacheSize` parameter defines the amount of main memory the server allocates for the cache. The default value depends on the server operating system; the minimum size is 512 KB. Although SOLID *SynchroNet* is able to run with a small cache size, a larger cache size speeds up the server. The cache size needed depends on the size of the index file, the number of connected users, and the nature of the operations executed against the server. For example:

```
CacheSize=512
```

ExtendIncrement

The `ExtendIncrement` parameter defines the number of blocks that is allocated at one time when SOLID *SynchroNet* needs to allocate more space for the database file. The default is 50 blocks. For example:

```
ExtendIncrement=50
```

Specifying the Backup Directory (General section)

Backups of the database, log files and the configuration file `solid.ini` are copied to the backup directory. If you are not using the default 'backup' directory, the backup directory must exist and it must have enough disk space for the backup files. It can be set to any existing directory except the database file directory, the log file directory or the working directory.

BackupDirectory parameter

The `BackupDirectory` parameter in the `[General]` section defines a name and location for your backup directory. Note that default 'backup' is a directory relative to your SOLID directory. For example if the parameter is:

```
BackupDirectory=backup
```

then the backup will be written to a directory that is a sub-directory of the SOLID directory.

▶ **Note**

The backup directory entered must be a valid path name in the server's operating system. For example if the server runs on a UNIX operating system, path separators must be slashes instead of backslashes.

Specifying the Transaction Log Files Directory (Logging section)

At commit time, transaction results are written immediately to a specified directory. This file must be stored to a local drive using local disk names to avoid problems with network I/O and to achieve better performance. The default log file directory is the SOLID working directory.

FileNameTemplate

The parameter `FileNameTemplate` in the `Logging` section defines a filename structure for the transaction log files. For example, the following setting

```
FileNameTemplate = d:\logdir\sol#####.log
```

instructs SOLID *SynchroNet* to create log files to directory `d:\logdir` and to name them sequentially starting from `sol00001.log`.

Specifying a Directory for the External Sorter Algorithm (Sorter section)

The external sorter algorithm is used for sorting processes that do not fit in main memory. When the `TmpDir_[1...N]` is specified in the configuration file, the external sorter algorithm is enabled. All temporary files used by the external sort are created in a specified directory (or directories) and are automatically deleted.

TmpDir_[1...N]

The `TmpDir[1...N]` parameter in the `Sorter` section defines the directory (or directories) that can be used for the external sorter algorithm. There is no default setting. For example:

```
TmpDir_1=c:\soldb\temp.1
TmpDir_2=d:\soldb\temp.2
TmpDir_3=g:\soldb\temp.3
```

ReadAhead

When the I/O manager is handling a long sequential search, it enters a read ahead operation mode. This ensures that the next file blocks of the search in question are read in the cache in advance. This naturally improves the overall performance of sequential searches. The `ReadAhead` sets the number of prefetched index leafs during long sequential searches. The default is 4. For example:

```
ReadAhead=4
```

PreFlushPercent

The preflush operations prepare the cache for the allocation of new blocks. The blocks are written onto the disk from the tail of the cache based on a Least Recently Used (LRC) algorithm. Therefore, when the new cache blocks are needed, they can be taken immediately without writing the old contents onto the disk. The `PreFlushPercent` parameter defines the percentage of page buffer which is kept clean by the preflush thread. The default is 5. For example:

```
PreFlushPercent=5
```

Setting Threads for Processing (Srv section)

In addition to the communication, I/O, and log manager threads, SOLID *SynchroNet* can start general purpose threads to execute tasks from the server's tasking system. Read *"Multithread Processing"* on page 1-8 for more details.

The optimum number of threads depends on the number of processors the system has installed. Usually it is most efficient to have between two and eight threads per processor.

Finding the value that provides the best performance requires experimentation. A good formula to start with is:

threads= (2 x number of processors) + 1

Threads

The `Threads` parameter in the `[Srv]` section defines the amount of general purpose threads used by SOLID *SynchroNet*. The default value is two threads for SOLID *SynchroNet* use. For example:

```
Threads=8
```

Setting SQL Trace Level (SQL section)

The SQL Info facility lets you specify a tracing level on the SQL Parser and Optimizer. For details on each level, read *"SQL Info Facility"* on page 9-1.

Info

The SQL Info facility is turned on by setting the `Info` parameter to a non-zero value in the `[SQL]` section of the configuration file. The output is written to a file named `sol-trace.out` in the SOLID directory.

Specifying Network Communication Tracing (Com section)

The communication tracing facility is necessary, for instance, if the network hardware is not functioning properly. By turning the tracing on, the communication layer is capable of logging even the system specific errors and may help in diagnosing the real problem in the network. The following parameters control the outputting of network trace information.

Trace

By changing `Trace` parameter default setting No to the value **Yes**, SOLID *SynchroNet* starts logging trace information on network messages on established network connection to the default trace file or to the file specified in the `TraceFile` parameters.

TraceFile

If `Trace` parameter is set to yes, trace information on network messages is written to a file specified with this parameter. The default if no file is specified is `soltrace.out`, written to the current working directory of the server or client depending on which end the tracing is started.

Specifying the Character Set for an Application (Client section)

By default, the `CharacterSet` parameter is set to the standard ISO Latin-1 (8859-1). If you are running an old SOLID client version (prior to version 3.0), the setting for the character set may be obsolete. In this case you need to include the `CharacterSet` parameter in the `solid.ini` file and set it to ISO 8859-1 or a setting suitable to your application.

Managing Parameters

You can view and modify SOLID *SynchroNet* parameters and their values in the following ways:

- Using the Configuration page in SOLID *DBConsole.*

  The *DBConsole* Configuration page lets you display a parameters listing in a tree node format and change configuration settings through a dialog box. For details, refer to *DBConsole* Online Help available by selecting **Help** on the menu bar.

- Entering the command `parameter` in SOLID *DBConsole* or SOLID *Remote Control* (teletype).

- Entering the SQL extension ADMIN COMMAND '*parameter*' in SOLID *SQL Editor.*
- Directly, by editing the `solid.ini` file in the SOLID directory.

Instructions for using managing parameters with ADMIN COMMAND and in `solid.ini` is described in the following sections.

▶ **Note**

For details on viewing and setting server communication protocol parameters only, read *"Managing Network Connections"* on page 6-1.

Viewing and Setting Parameters with ADMIN COMMAND

With ADMIN COMMAND, you can change the parameters remotely through a Solid server without restarting it. All parameters are accessible even if they are not present in the `solid.ini` configuration file. If the parameter is not present, the default value is used.

Viewing All Parameters

To view all parameters, enter the following command in SOLID *DBConsole* or SOLID *Remote Control* (teletype):

```
parameter
```

or in SOLID *SQL Editor*:

```
ADMIN COMMAND 'parameter';
```

A list of all parameters with current and default values is returned. If you need to view communication protocol settings, see *"Managing Network Connections"* on page 5-1.

If desired, you can also qualify this command with a **-r** option to display only the current values. For example:

```
parameter -r
```

The command provides each parameter in one row. Note that there are four parts to the resultset:

- **Section name** identifies the parameter category.
- **Parameter name** is the name of the parameter.
- **Value** displays the parameters current value from the solid.ini file if it is present; if the parameter is not present, it displays the system default value.

- **System** default displays the default value.

Viewing the Value of a Specific Parameter

To view the value of a specific parameter, enter the following command using SOLID *DBConsole* or SOLID *Remote Control* (teletype):

parameter -r *section_name.parameter_name*

or in SOLID *SQL Editor*:

ADMIN COMMAND ’parameter -r *section_name.parameter_name*’;

where:

section_name is the category name where the parameter is located in `solid.ini`

Example:

- see if communication trace is on

```
parameter -r com.trace
```

-or-

```
ADMIN COMMAND 'parameter -r com.trace';
```

If you need to view a communication protocol settings, see *“Managing Network Connections”* on page 6-1.

Viewing the Description of a Specific Parameter

You can also view a description of a specific parameter, which includes valid parameter types and access modes. This is useful information, especially because parameters may need to be handled dynamically; parameter support may vary between products, platforms, or releases.

To view a parameter’s description, enter the following command using SOLID *DBConsole* or SOLID *Remote Control* (teletype):

describe parameter *section_name.parameter_name*

or in SOLID *SQL Editor*:

ADMIN COMMAND ’describe parameter *section_name.parameter_name*’;

where:

section_name is the category name where the parameter is located in `solid.ini`

Example:

```
describe parameter com.trace
```

-or-

```
ADMIN COMMAND 'describe parameter com.trace';
```

The command provides a resultset with the following four rows:

- Row 1 is a short description of the parameter
- Row 2 shows the valid types for the parameter
- Row 3 show the parameter access modes, which can be one of the following:
  - READ - the parameter is read only and its value cannot be altered.
  - WRITE - the parameter value can be altered and the new value is used.
  - CREATE - the parameter value is used only when a new database is created and cannot be altered.
  - STARTUP - the parameter value can be altered and the new value is used only when the server is started.
- Row 4 shows the current value of the parameter.

Setting a Parameter Value

To set a value for a specific parameter, enter the following command using SOLID *DBConsole* or SOLID *Remote Control* (teletype):

parameter *section_name.parameter_name=value*

or in SOLID *SQL Editor*:

ADMIN COMMAND 'parameter *section_name.parameter_name=value*';

where:

section_name is the category name where the parameter is located in `solid.ini`

`value` is a valid parameter value. If no value is specified, this sets the parameter with a default (or unset) value.

Example:

-set communication trace is on

```
parameter com.trace=yes
```

-or-

```
ADMIN COMMAND 'parameter com.trace=yes';
```

Note

Parameter management operations are not transactional and cannot be rolled back.

The commands return the new value as the resultset. If the parameter’s access mode is READ (read-only) or the value entered is invalid, the ADMIN COMMAND statement returns an error.

Viewing and Setting Parameters in SOLID.INI

1. Open the `solid.ini` file located in the working directory of your SOLID *SynchroNet* process.
2. View the value of the parameter.
3. If necessary add the section, parameter and parameter's value.
4. Save the changes.

You need to restart the SOLID *SynchroNet* process to activate the changes.

The parameters displayed are the parameters currently active in the server. If you have not set a parameter value, the displayed value is the default value for the parameter. The default values are set at start-up and depend on the operating system SOLID *SynchroNet* runs on.

Note

To force a parameter value change to take effect you must shut down and restart the SOLID *SynchroNet* process.

Caution

The new parameter values are not checked by the server. Setting an unreasonable value for a parameter may result in an operation failure the next time the server process is started. Do not set a parameter to a random value unless you know what you are doing. Use the default parameter values as an indication on the value range.

Constant Parameter Values

The parameter access mode for the `Blocksize` parameter in both the `IndexFile` and `Logging` sections of the configuration file are CREATE. This means that the parameter is set when the database is created and cannot be modified afterwards.

If you want to use a different constant value, you have to create a new database. Before creating a new database, set the new parameter constant value by editing the `solid.ini` file in the SOLID directory.

The following example sets a new block size for the index file by adding the following lines to the `solid.ini` file:

```
[Indexfile]
Blocksize=4096
```

After editing and saving the `solid.ini` file, move or delete the old database and log files, and start SOLID *SynchroNet*.

▶ **Note**

If you have changed the constant value for the log file block size only, you are only required to move or delete the log files, not the database file.

The server program will create a new database with the new constant value from the `solid.ini` file.

8

Performance Tuning

This chapter discusses techniques that you can use to improve the performance of SOLID *SynchroNet*. The topics included in this chapter are:

- Tuning SQL statements and applications
- Using indexes to improve query performance
- Optimizing batch inserts and updates
- Tuning memory allocation
- Tuning CPU concurrency load
- Tuning I/O
- Tuning checkpoints
- Using optimizer hints for performance

Tuning SQL Statements and Applications

Tuning the SQL statements, especially in applications where complex queries are involved, is generally the most efficient means of improving the database performance.

Be sure to tune your application *before* tuning the RDBMS because:

- during application design you have control over the SQL statements and data to be processed
- you can improve performance even if you are unfamiliar with the internal working of the RDBMS you are going to use
- if your application is not tuned well, it will not run well even on a well-tuned RDBMS

You should know what data your application processes, what are the SQL statements used, and what operations the application performs on the data. For example, you can improve query performance when you keep SELECT statements simple, avoiding unnecessary clauses and predicates.

Evaluating Application Performance

To isolate areas where performance is lacking in your application, SOLID *SynchroNet* provides the following diagnostic tools for observing database performance:

- SQL info facility
- EXPLAIN PLAN statement

These tools are helpful in tuning your application and identifying any insufficient SQL statements in it. Read Chapter 9, "Diagnostics and Troubleshooting" for additional information on how to use these tools.

Using Indexes to Improve Query Performance

You can use indexes to improve the performance of queries. A query that references an indexed column in its WHERE clause can use the index. If the query selects only the indexed column, the query can read the indexed column value directly from the index, rather than from the table.

If a table has a primary key, SOLID *SynchroNet* orders the rows on disk in the order of the values of the primary key. Otherwise the rows are ordered using the ROWID, that is, the rows are stored on disk in the order they are inserted into the database. Read *"Primary Keys"* on page 4-9.

Searches with row value constructor constraints are optimized to use an index (if one is available). For efficiency, SOLID *SynchroNet* uses an index to resolve constraints of the form (A, B, C) >= (1, 2, 3). The constraints <, <=, >= and > can use the index. Note that if several row value constraints are defined for one table, only the first one is optimized to use an index.

Indexes improve the performance of queries that select a small percentage of rows from a table. You should consider using indexes for queries that select less than 15% of table rows.

Full table scan

If a query does not use an index, SOLID *SynchroNet* must perform a full table scan to execute the query. This involves reading all rows of a table sequentially. Each row is examined to determine whether it meets the criteria of the query's WHERE clause. Finding a single row with an indexed query can be substantially faster than finding the row with a full table scan. On the other hand, a query that selects more than 15% of a table's rows may be performed faster by a full table scan than by an indexed query.

To perform a full table scan, every block in the table is read. For each block, every row stored in the block is read. To perform an indexed query the rows are read in the order in which they appear in the index, regardless of which blocks contain them. If a block contains more than one selected row it may be read more than once. So, there are cases when a full table scan requires less I/O than an indexed query.

Concatenated indexes

An index can be made up of more than one column. Such an index is called a concatenated index. It is recommended to use concatenated indexes when possible.

Whether or not a SQL statement uses a concatenated index is determined by the columns contained in the WHERE clause of the SQL statement. A query can use a concatenated index if it references a leading portion of the index in the WHERE clause. A leading portion of an index refers to the first column or columns specified in the CREATE INDEX statement.

Example:

```
create index job_sal_deptno on emp(job, sal, deptno);
```

This index can be used by these queries:

```
select * from emp where job = ‘clerk’ and sal =
   800 and deptno = 20;
select * from emp where sal = 1250 and job = salesman;
select job, sal from emp where job = ‘manager’ ;
```

The following query does not contain the first column of the index in its WHERE clause and cannot use the index:

```
select * from emp where sal = 6000;
```

Choosing columns to index

The following list gives guidelines in choosing columns to index:

- index columns that are used frequently in WHERE clauses
- index columns that are used frequently to join tables
- index columns that are used frequently in ORDER BY clauses
- index columns that have few of the same values or unique values in the table.
- do not index small tables (tables that use only a few blocks) because a full table scan may be faster than an indexed query

- if possible choose a primary key that orders the rows in the most appropriate order
- if only one column of the concatenated index is used frequently in WHERE clauses, place that column first in the CREATE INDEX statement
- if more than one column in concatenated index is used frequently in WHERE clauses, place the most selective column first in the CREATE INDEX statement.

Optimizing Batch Inserts and Updates

You can optimize the speed for large batch inserts and updates to SOLID *SynchroNet*. Following are guidelines for increasing speed:

1. Check that you are running the application with the AUTOCOMMIT mode set off.

   SOLID ODBC Driver's default setting is AUTOCOMMIT. This is the standard setting according to the ODBC specification. To set your application with AUTOCOMMIT off, call the SQLSetConnectOption function as in the following example:

```
rc = SQLSetConnectOption
(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
```

2. Do not use large transactions.

   It is recommended to COMMIT transactions after every 100-200 rows inserted.

3. Use stored procedures to further increase the speed of inserts.

   The use of stored procedures provides an additional 10-20% speed increase.

4. Lower the `MergeInterval` parameter.

   The `MergeInterval` parameter determines the frequency of writing data from SOLID *SynchroNet*'s cache to disk. For large batch operations, set the value to a lower level. The parameter should be set to approximately 100 when running large inserts.

Number 1 and 2 of these guidelines are the most important actions you can take to increase the speed of batch inserts. The actual rate of insertions also depends on your hardware, on the amount of data per row and on the existing indices for the table.

Tuning Memory Allocation

Main memory is allocated dynamically according to system usage and the operating system environment. The basic element of the memory management system is a pool of central memory buffers of equal size. You can configure the amount and size of memory buffers to meet the demands of different application environments.

Tuning Your Operating System

Your operating system may store information in:

- real memory
- virtual memory
- expanded storage
- disk

Your operating system may also move information from one location to another. Depending on your operating system, this movement is called paging or swapping. Many operating systems page and swap to accommodate large amounts of information that do not fit into real memory. However, this takes time. Excessive paging or swapping can reduce the performance of your operating system and indicates that your system's total memory may not be large enough to hold everything for which you have allocated memory. You should either increase the amount of total memory or decrease the amount of database cache memory allocated.

Database Cache

The information used by SOLID *SynchroNet* is stored either in memory or on disk. Since memory access is faster than disk access, it is desirable for data requests to be satisfied by access to memory rather than access to disk.

Database cache uses available memory to store information that is read from the hard disk. When an application next time requests this information, the data is read from memory instead of from the hard disk. The default value of cache depends on the platform used and can be changed by changing the `CacheSize` parameter. Increasing the value is recommended when there are several concurrent users.

The following values can be used as a starting point:

- a dedicated server with 16 MB RAM: Cachesize 4 MB
- a dedicated server with 32 MB RAM: Cachesize 10 MB
- a dedicated server with 64 MB RAM: Cachesize 30 MB

▶ **Note**

You should increase the value of `Cachesize` carefully. If a value is too large, it leads to poor performance.

Sorting

SOLID *SynchroNet* does all sorting by default in memory. The amount of memory used for sorting is determined by the parameter `SORTARRAYSIZE` in the `[SQL]` section. If the amount of data to be sorted does not fit into the allocated memory, you may want to increase the value of the parameter `SORTARRAYSIZE`. If there is not enough memory to increase the value of `SORTARRAYSIZE` you should activate external sort that stores intermediate information to disk.

The external disk sort is activated by adding the following section and parameters in the configuration file `solid.ini`:

```
[sorter]
TmpDir_1 = c:\tmp
```

Additional sort directories are added with similar definitions:

```
[sorter]
TmpDir_1 = c:\tmp
TmpDir_2 = d:\tmp
TmpDir_3 = e:\tmp
```

Defining more than one sorter temporary directory on separate physical disks significantly improves sort performance by balancing the I/O load to multiple disks.

Tuning CPU Concurrency Load

SOLID *SynchroNet* contains shared data structures. Some actions on them are protected by mutual exclusion semaphores (mutexes) which allow only one thread access to a resource at a time. In a multi-CPU machine, it is possible that at some configuration and workload, one or more mutexed sections can become a bottleneck that limits the maximum CPU load.

To improve concurrency you can do the following:

- Run the server with more general purpose (worker) threads (for example, 40-80 threads may be reasonable on an NT machine running 400 clients). For details on setting threads, read *"Setting Threads for Processing (Srv section)"* on page 7-6.
- Increase the log file size from the 2048 default. A large blocksize for a log file speeds up logging considerably (especially under NT and UW-SCSI disks). A recommended setting is:

  ```
  [logging]
  BlockSize=32728
  ```

Note

Be sure to shut down the server and delete all old log files before changing the blocksize value.

Tuning I/O

The performance of many software systems is inherently limited by disk I/O. Often CPU activity must be suspended while I/O activity completes.

Distributing I/O

Disk contention occurs when multiple processes try to access the same disk simultaneously. To avoid this, move files from heavily accessed disks to less active disks until they all have roughly the same amount of I/O.

Follow these guidelines:

- use a separate disk for log files
- divide your database into several files and place each of these database files on a separate disk. Read *"Managing Database Files and Caching (IndexFile section)"* on page 7-2.
- consider using a separate disk for the external sorter

Setting the MergeInterval Parameter

SOLID *SynchroNet*'s indexing system consists of two storage structures, the Bonsai Tree which stores new data in central memory and the storage server which stores more stable data. As the Bonsai Tree performs concurrency control, storing delete, insert, and update operations, as well as key values, it merges new commit data to the storage server as a highly-optimized batch insert. This offers significant I/O optimization and load balancing.

You can adjust the number of index inserts made in the database that causes the merge process to start by setting the following parameter in the General section of the `solid.ini` file. For example:

```
MergeInterval = 100
```

Normally the recommended setting is the default value, which is cache size dependent. The default is calculated dynamically from the cache size, so that only part of the cache is used for the Bonsai Tree. If you change the merge interval, be sure that the cache is large enough to accommodate the Bonsai Tree.

▶ **Note**

If the merge interval setting does not allow the Bonsai Tree to fit into cache, then it is flushed partially to the disk; this has an adverse affect on performance.

Tuning Checkpoints

Checkpoints are used to store a consistent state of the database quickly onto the disk.

Checkpoints affect:

- recovery time performance
- runtime performance

Frequent checkpoints can reduce the recovery time in the event of a system failure. If the checkpoint interval is small, then relatively few changes to the database are made between checkpoints and relatively few changes must be recovered.

Checkpoints cause SOLID *SynchroNet* to perform I/O, so they momentarily reduce the runtime performance. This overhead is usually small.

Using Optimizer Hints

Due to various conditions with the data, user query, and database, the SQL Optimizer is not always able to choose the best possible execution plan. For more efficiency, you may want to force a merge join because you know, unlike the Optimizer, that your data is already sorted.

Or sometimes specific predicates in queries cause performance problems that the Optimizer cannot eliminate. The Optimizer may be using an index that you know is not optimal. In this case, you may want to force the Optimizer to use one that produces faster results.

Optimizer hints is a way to have better control over response times to meet your performance needs. Within a query, you can specify directives or *hints* to the Optimizer, which it then uses to determine its query execution plan. Hints are detected through a pseudo comment syntax from SQL2.

Hints are available for:

- Selecting merge or nested loop join

- Using a fixed join order as given in the from list
- Selecting internal or external sort
- Selecting a particular index
- Selecting a table scan over an index scan
- Selecting sorting before or after grouping

You can place a hint(s) in a SQL statement as a static string, just after a SELECT, UPDATE, or DELETE keyword. Hints are not allowed after the INSERT keyword. The hint always follows the SQL statement that applies to it.

Table name resolution in optimizer hints is the same as in any table name in a SQL statement. When there is an error in a hint specification, then the whole SQL statement fails with an error message.

Hints are enabled and disabled using the following configuration parameter in `solid.ini`:

```
[Hints]
EnableHints=YES | NO
```

`The default is set to` **`YES.`**

Optimizer Hints Syntax

The syntax to specify an optimizer hint is:

`--(* vendor (SOLID), product (Engine), option(hint)`

`--`*`hint`* **`*)--`**

hint:=

```
   [MERGE JOIN |
   LOOP JOIN |
   JOIN ORDER FIXED |
   INTERNAL SORT |
   EXTERNAL SORT |
   INDEX [REVERSE] table_name.index_name |
   PRIMARY KEY [REVERSE] table_name
   FULL SCAN table_name |
  [NO] SORT BEFORE GROUP BY]
```

Following is a description of the keywords and clauses used in the syntax:

Pseudo comment identifier

The pseudo comment prefix is followed by identifying information. You must specify the vendor as SOLID, product as Engine, and the option, which is the pseudo comment class name, as hint.

Hint

Hints always follow the SELECT, UPDATE, or DELETE keyword that applies to it.

Note

Hints are not allowed after the INSERT keyword.

Each subselect requires its own hint; for example, the following are valid uses of hints syntax:

INSERT INTO ... SELECT *hint* FROM ...

UPDATE *hint* TABLE ... WHERE *column* = (SELECT *hint* ... FROM ...)

DELETE *hint* TABLE ... WHERE *column* = (SELECT hint ... FROM ...)

Be sure to specify multiple hints in one pseudo comment separated by commas as shown in the following examples:

Example 1

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
*
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

Example 2

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--INDEX TAB1.INDEX1
--INDEX TAB1.INDEX1 FULL SCAN TAB2 *)--
```

```
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

Hint is a specific semantic, corresponding to a specific behavior. Following is a list of possible hints:

Hint	**Definition**
MERGE JOIN	Directs the Optimizer to choose the merge join access plan in a select query for all tables listed in the FROM clause. Use this hint when the data is sorted by a join key and the nested loop join performance is not adequate. Note that when data is not sorted before performing the merge operation, the SOLID query executor sorts the data. When considering the usage of this hint, keep in mind that the merge join with a sort is more resource intensive than the merge join without the sort.
LOOP JOIN	Directs the Optimizer to pick the nested loop join in a select query for all tables listed in the FROM clause. By default, the Optimizer does not pick the nested loop join. Using the loop join when tables are small and fit in memory may offer greater efficiency than using other join algorithms.
JOIN ORDER FIXED	Specifies that the Optimizer use tables in a join in the order listed in the FROM clause of the query. This means that the Optimizer does not attempt to rearrange any join order and does not try to find alternate access paths to complete the join. Before using this hint, be sure to run the EXPLAIN PLAN to view the associated plan. This gives you an idea on the access plan used for executing the query with this join order.
INTERNAL SORT	Specifies that the query executor use the internal sort. Use this hint if the expected result set is small (100s of rows as opposed to 1000s of rows); for example, if you are performing some aggregates, ORDER BY with small result sets, or GROUP BY with small result sets, etc. This hint avoids the use of the more expensive external sort.

Hint	Definition
EXTERNAL SORT	Specifies that the query executor use the external sort. Use this hint when the expected result set is large and does not fit in memory; for example, if the expected result set has 1000s of rows. In addition, specify the SORT working directory in the `solid.ini` before using the external sort hint. If a working directory is not specified, you will receive a run-time error.
INDEX {REVERSE} *table_name*.*index_name*	Forces a given index scan for a given table. In this case, the Optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. The optional keyword **REVERSE** returns the rows in the reverse order. In this case, the query executor begins with the last page of the index and starts returning the rows in the descending (reverse) key order of the index. Note that in *tablename*.*indexname*, the *tablename* is a fully qualified table name which includes the *catalogname* and *schemaname*.
PRIMARY KEY [REVERSE] *tablename*	Forces a primary key scan for a given table. The optional keyword **REVERSE** returns the rows in the reverse order. If the primary KEY is not available for the given table, then you will receive a run-time error.
FULL SCAN *table_name*	Forces a table scan for a given table. In this case, the optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. In this FULL SCAN, the query executor tries to use the PRIMARY KEY, if one is available. If not, then it uses the SYSTEM KEY.

Hint	Definition
[NO] SORT BEFORE GROUP BY	Indicates whether the SORT operation occurs before the result set is grouped by the GROUP BY columns. If the grouped items are few (100s of rows) then use NO SORT BEFORE. On the other hand, if the grouped items are large (1000s of rows), then use SORT BEFORE.

Optimizer Hint Examples

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- JOIN ORDER FIXED *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- LOOP JOIN *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX REVERSE MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- SORT BEFORE GROUP BY *)--
AVG(I) FROM TAB1 WHERE I > 10 GROUP BY I2

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INTERNAL SORT *)--
* FROM TAB1 WHERE I > 10 ORDER BY I2
```

9 Diagnostics and Troubleshooting

This chapter provides information on the following SOLID *SynchroNet* diagnostic tools:

- SQL info facility and the EXPLAIN PLAN statement used to tune your application and identify inefficient SQL statements in your application.
- Network trace facility used to trace the server communication
- Ping facility used to trace client communication

You can use these facilities to observe performance, troubleshooting, and produce high quality problem reports. These reports lets you pinpoint the source of your problems by isolating them under product categories (such as SOLID *ODBC API*, SOLID *ODBC Driver*, SOLID *JDBC Driver*, etc.).

Observing Performance

You can use the SQL Info facility to provide information on a SQL statement and the SQL statement EXPLAIN PLAN to show the execution graph that the SQL optimizer selected for a given SQL statement. Typically, if you need to contact SOLID Technical support, you will be asked to provide the SQL statement, EXPLAIN PLAN output, and SQL Info output from the EXPLAIN PLAN run with info level 8 for more extensive trace output.

SQL Info Facility

Run your application with the SQL Info facility enabled. The SQL Info facility generates information for each SQL statement processed by SOLID *SynchroNet*.

The `Info` parameter in the `[SQL]` section specifies the tracing level on the SQL parser and optimizer as an integer between 0 (no tracing) and 8 (SOLID info from every fetched row). Trace information will be output to the file named `soltrace.out` in the SOLID directory.

Example:

```
[SQL]
info = 1
```

SQL Info levels

Info value	Information
0	no output
1	table, index, and view info in SQL format
2	SQL execution graphs (for SOLID technical support use only)
3	some SQL estimate info, SOLID selected key name
4	all SQL estimate info, SOLID selected key info
5	SOLID info also from discarded keys
6	SOLID table level info
7	SQL info from every fetched row
8	SOLID info from every fetched row

The SQL Info facility can also be turned on with the following SQL statement (this sets SQL Info on only for the client that executes the statement):

```
SET SQL INFO ON LEVEL info_value FILE file_name
```

and turned off with the following SQL statement:

```
SET SQL INFO OFF
```

Example:

```
SET SQL INFO ON LEVEL 1 FILE ‘my_query.txt’
```

The EXPLAIN PLAN Statement

The syntax of the EXPLAIN PLAN statement is:

```
EXPLAIN PLAN FOR sql_statement
```

The EXPLAIN PLAN statement is used to show the execution plan that the SQL optimizer has selected for a given SQL statement. An execution plan is a series of primitive operations, and an ordering of these operations, that SOLID *SynchroNet* performs to execute the statement. Each operation in the execution plan is called a unit.

Unit	Description
JOIN UNIT*	Join unit joins two or more tables. The join can be done by using loop join or merge join.
TABLE UNIT	Table unit is used to fetch the data rows from a table. Table unit is always the last unit in the chain, since it is responsible for fetching the actual data from the index or table.
ORDER UNIT	Order unit is used to order rows for grouping or to satisfy ORDER BY. The ordering can be done in memory or using an external disk sorter.
GROUP UNIT	Group unit is used to do grouping and aggregate calculation (SUM, MIN, etc.).
UNION UNIT*	Union unit performs the UNION operation. The unit can be done by using loop join or merge join.
INTERSECT UNIT*	Intersect unit performs the INTERSECT operation. The unit can be done by using loop join or merge join.
EXCEPT UNIT*	Except unit performs the EXCEPT operation. The unit can be done by using loop join or merge join.

*This unit is generated also for queries that reference only a single table. In that case no join is executed in the unit; it simply passes the rows without manipulating them.

Explain Plan Table Columns

The table returned by the EXPLAIN PLAN statement contains the following columns.

Column Name	Description
ID	The output row number, used only to guarantee that the rows are unique.
UNIT_ID	This is the internal unit id in the SQL interpreter. Each unit has a different id. The unit id is a sparse sequence of numbers, because the SQL interpreter generates unit ids also for those units that are removed during the optimization phase. If more than one row has the same unit id it means that those rows belong to the same unit. For formatting reasons the info from one unit may be divided into several different rows.
PAR_ID	Parent unit id for the unit. The parent id number refers to the id in the UNIT_ID column.

Column Name	Description
JOIN_PATH	For join, union, intersect, and except units there is a join path which specifies which tables are joined in the unit and the join order for tables. The join path number refers to the unit id in the UNIT_ID column. It means that the input to the unit comes from that unit. The order in which the tables are joined is the order in which the join path is listed. The first listed table is the outer-most table in a loop join.
UNIT_TYPE	Unit type is the execution graph unit type.
INFO	Info column gives additional info. It may contain, for example, index usage, the database table name and constraints used in the database engine to select rows. Note that the constraints listed here may not match those constraints given in the SQL statement.

The following texts may exist in the INFO column for different types of units.

Unit type	Text in Info column	Description
TABLE UNIT	*tablename*	The table unit refers to table *tablename*.
TABLE UNIT	*constraints*	The constraints that are passed to the database engine are listed. If for example in joins the constraint value is not known in advance, the constraint value is displayed as NULL.
TABLE UNIT	SCAN TABLE	Full table scan is used to search for rows.
TABLE UNIT	SCAN *indexname*	Index *indexname* is used to search for rows. If all selected columns are found from an index, sometimes it is faster to scan the index instead of the entire table because the index has fewer disk blocks.
TABLE UNIT	PRIMARY KEY	The primary key is used to search rows. This differs from SCAN in that the whole table is not scanned because there is a limiting constraint to the primary key attributes.
TABLE UNIT	INDEX *indexname*	Index *indexname* is used to search for rows. For every matching index row, the actual data row is fetched separately.

Unit type	Text in Info column	Description
TABLE UNIT	INDEX ONLY *indexname*	Index *indexname* is used to search for rows. All selected columns are found from the index, so the actual data rows are not fetched separately
JOIN UNIT	MERGE JOIN	Merge join is used to join the tables.
JOIN UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
JOIN UNIT	LOOP JOIN	Loop join is used to join the tables.
ORDER UNIT	NO ORDERING REQUIRED	No ordering is required, the rows are retrieved in correct order from the data-base engine.
ORDER UNIT	EXTERNAL SORT	External sorter is used to sort the rows. To enable external sorter, the temporary direc-tory name must be specified in the Sorter section of the configuration file.
ORDER UNIT	FIELD *n* USED AS PARTIAL ORDER	For distinct result sets, an internal sorter (in-memory sorter) is used for sorting and the rows retrieved from the database engine are partially sorted with column number *n*. The partial ordering helps the internal sorter avoid multiple passes over the data.
ORDER UNIT	*n* FIELDS USED FOR PAR-TIAL SORT	An internal sorter (in-memory sorter) is used for sorting and the rows retrieved from the database engine are partially sorted with n fields. The partial ordering helps the internal sorter to avoid multiple passes over the data.
ORDER UNIT	NO PARTIAL SORT	Internal sorter is used for sorting and the rows are retrieved in random order from the database engine.
UNION UNIT	MERGE JOIN	Merge join is used to join the tables.
UNION UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
UNION UNIT	LOOP JOIN	Loop join is used to join the tables.
INTERSECT UNIT	MERGE JOIN	Merge join is used to join the tables.
INTERSECT UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.

Unit type	Text in Info column	Description
INTERSECT UNIT	LOOP JOIN	Loop join is used to join the tables.
EXCEPT UNIT	MERGE JOIN	Merge join is used to join the tables.
EXCEPT UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
EXCEPT UNIT	LOOP JOIN	Loop join is used to join the tables.

Example 1

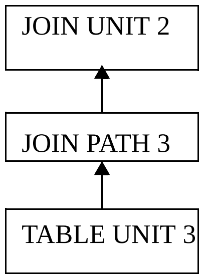
```
EXPLAIN PLAN FOR SELECT * FROM TENKTUP1 WHERE UNIQUE2_NI BETWEEN 0 AND
99;
```

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
1	2	1	3	JOIN UNIT	
2	3	2	0	TABLE UNIT	TENKTUP1
3	3	2	0		FULL SCAN
4	3	2	0		UNIQUE2_NI <= 99
5	3	2	0		UNIQUE2_NI >= 0
6	3	2	0		

Execution graph:

JOIN UNIT 2 gets input from TABLE UNIT 3

TABLE UNIT 3 for table TENKTUP1 does a full table scan with constraints UNIQUE2_NI <= 99 and UNIQUE2_NI >= 0



Example 1. Execution graph

Example 2

```
EXPLAIN PLAN FOR SELECT * FROM TENKTUP1, TENKTUP2 WHERE TENKTUP1.UNIQUE2
> 4000 AND TENKTUP1.UNIQUE2 < 4500 AND TENKTUP1.UNIQUE2 =
TENKTUP2.UNIQUE2;
```

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
1	6	1	9	JOIN UNIT	MERGE JOIN
2	6	1	10		
3	9	6	0	ORDER UNIT	NO ORDER-ING REQUIRED
4	8	9	0	TABLE UNIT	TENKTUP2
5	8	9	0		PRIMARY KEY
6	8	9	0		UNIQUE2 < 4500
7	8	9	0		UNIQUE2 > 4000
8	8	9	0		
9	10	6	0	ORDER UNIT	NO ORDER-ING REQUIRED
10	7	10	0	TABLE UNIT	TENKTUP1
11	7	10	0		PRIMARY KEY
12	7	10	0		UNIQUE2 < 4500
13	7	10	0		UNIQUE2 > 4000
14	7	10	0		

Execution graph:

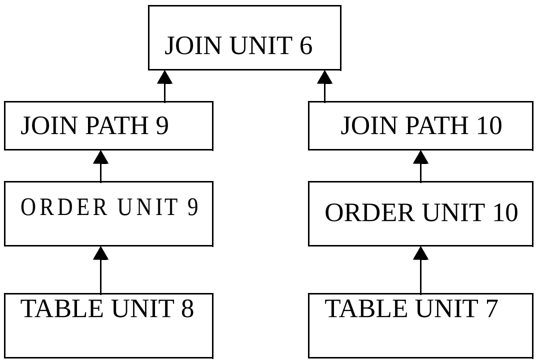
JOIN UNIT 6 the input from order units 9 and 10 are joined using merge join algorithm

ORDER UNIT 9 orders the input from TABLE UNIT 8. Since the data is retrieved in correct order, no real ordering is needed

ORDER UNIT 10 orders the input from TABLE UNIT 7. Since the data is retrieved in correct order, no real ordering is needed

TABLE UNIT 8: rows are fetched from table TENKTUP2 using primary key. Constraints UNIQUE2 < 4500 and UNIQUE2 > 4000 are used to select the rows

TABLE UNIT 7: rows are fetched from table TENKTUP1 using primary key. Constraints UNIQUE2 < 4500 and UNIQUE2 > 4000 are used to select the rows



Example 2. Execution graph

Tracing Communication between Client and Server

SOLID*SynchroNet* provides following tools for observing the communication between an application and a database server:

- the Network Trace facility
- the Ping facility

You can use these tools to analyze the functionality of the networking between an application and an embedded engine. The network trace facility should be used when you want to know why a connection is not established to an embedded engine. The ping facility is used to determine how fast packets are transferred between an application and an a database server.

The Network Trace Facility

Network tracing can be done on the *SynchroNet* computer, on the application computer or on both computers concurrently. The trace information is written to the default trace file or file specified in the `TraceFile` parameter.

The default name of the output file is `soltrace.out`. This file will be written to the current working directory of the server or client depending on which end the tracing is started.

The file contains information about:

- loaded DLLs
- network addresses
- possible errors

The Network Trace facility is turned on by editing the configuration file:

```
[Com]
Trace ={Yes|No}
; default No
TraceFile = file_name
; default soltrace.out
```

or by using the environment variables `SOLTRACE` and `SOLTRACEFILE` to override the definitions in the configuration file. Setting of `SOLTRACE` and `SOLTRACEFILE` environment variables have the same effect as the parameters `Trace` and `TraceFile` in the configuration file.

Note

Defining the `TraceFile` configuration parameter or the `SOLTRACEFILE` environment variable automatically turns on the Network trace facility.

A third alternative to turn on the Network trace facility is to use the option **-t** and/or **-o***filename* as a part of the network name. The option -t turns on the Network trace facility. The option **-o** turns on the facility and defines the name of the trace output file.

Example 1. Defining Parameter Trace in the Configuration File

```
[Com]
Connect = nmp SOLID
```

```
Listen = nmp SOLID
Trace = Yes
```

Example 2. Defining Environment Variables

```
set SOLTRACE = Yes
```

or

```
set SOLTRACEFILE = trace.out
```

Example 3. Using Network Name Options

```
[Com]
Connect  = nmp -t solid
Listen = nmp -t solid
```

or

```
[Com]
Connect = nmp -oclient.out solid
Listen = nmp -oserver.out solid
```

The Ping Facility

The Ping facility can be used to test the performance and functionality of the networking. The Ping facility is built in all SOLID client applications and is turned on with the network name option **-p***level*.

The output file will be written to the current working directory of the computer where the parameter is given. The default name of the output file is `soltrace.out`.

Clients can always use the Ping facility at level 1. Levels 2, 3, 4 or 5 may only be used if the server is set to use the Ping facility at least at the same level.

The Ping facility levels are:

Setting	Function	Description
0	no operation	do nothing, default
1	check that server is alive	exchange one 100 byte message
2	basic functional test	exchange messages of sizes 0.1K, 1K, 2K..30K, increment 1K
3	basic speed test	exchange 100 messages of sizes 0.1K, 1K, 8K and display each sub-result and total time

Setting	Function	Description
4	heavy speed test	exchange 100 messages of sizes 0.1K, 1K, 2K, 4K, 8K, 16K and display each sub-result and total time
5	heavy functional test	exchange messages of sizes 1..30K, increment 1 byte

Note

If a SOLID client does not have an existing server connection, you can use the **SQLConnect**() function with the connect string **-p1** option (ping test, level 1) to check if SOLID is listening in a certain address. Without logging into SOLID, **SQLConnect**() can then check the network layer and ensure SOLID is listening. When used in this manner, **SQLConnect**() generates error code 21507, which means the server is alive.

Example 1

The client turns on the Ping facility by using the following network name:

```
nmp -p1 -oping.out SOLID
```

This runs the Ping facility at the level 1 into a file named SOLTRACE.OUT. This test checks if the server is alive and exchanges one 100 byte message to the server.

After the Ping facility has been run, the client exits with the following message:

```
SOLID Communication return code xxx: Ping test successful/failed,
results are in file FFF.XX
```

Example 2

If the server is using the following listen parameter, applications can run the Ping facility at levels 1,2 and 3, but not 4 and 5.

```
[Com]
Listen = nmp -p3 SOLID
```

Note

Ping clients running at level greater than 3 may cause heavy network traffic and may cause slowness of application using the network. They will also slow down ordinary SQL clients connected to the same SOLID *SynchroNet*.

Problem Reporting

SOLID *SynchroNet* offers sophisticated diagnostic tools and methods for producing high quality problem reports with very limited effort. Use the diagnostic tools to capture all the relevant information about the problem.

All problem reports should contain the following files and information:

- solid.ini
- license number
- solmsg.out
- solerror.out
- soltrace.out
- problem description
- steps to reproduce the problem
- all error messages and codes
- contact information, preferably email address of the contact person

Problem Categories

Most problems can be divided into the following categories:

- SOLID *ODBC API*
- SOLID *ODBC* or *JDBC Driver*
- UNIFACE driver for SOLID *SynchroNet*
- Communication problems between the application and SOLID *SynchroNet*

The following pages include a detailed instructions to produce proper problem report for each problem type. Please follow the guidelines carefully.

SOLID *ODBC API* Problems

If the problem concerns the performance of a specific SOLID *ODBC API* or SQL statement, you should run SQL info facility at level 4 and include the generated `soltrace.out` file into your problem report. This file contains the following information:

- create table statements
- create view statements
- create index statements
- SQL statement(s)

SOLID *ODBC Driver* Problems

If the problem concerns the performance of SOLID *ODBC Driver*, please include the following information:

- SOLID *ODBC Driver* name, version, and size
- ODBC Driver Manager version and size

If the problem concerns the cooperation of SOLID *SynchroNet* and any third party standard software package, please include the following information:

- Full name of the software
- Version and language
- Manufacturer
- Error messages from the third party software package

Use ODBC trace option to get a log of the ODBC statements and include it to your problem report.

SOLID *JDBC Driver* Problems

If the problem is related to the SOLID *JDBC Driver*, please include the following information into your problem report:

- Exact version of JDK or JRK used
- Size and date of the SOLIDDriver class package
- Contents of DriverManager.setLogStream(someOutputStream) output, if available
- Call stack (that is, Exception.printStackTract() output) of the application, if an Exception has occurred n the application

UNIFACE Driver for SOLID *SynchroNet* Problems

If the problem concerns the performance of SOLID UNIFACE Driver, please include following information:

- SOLID *UNIFACE Driver* version and size
- UNIFACE version and platform
- Contents of the UNIFACE message frame
- Error codes from the driver, $STATUS, $ERROR
- All necessary files to reproduce the problem (TRXs, SQL scripts, USYS.ASN etc.)

Communication between a Client and Server

If the problem concerns the performance of the communication between a client and server use the Network trace facility and include the generated trace files into your problem report. Please include the following information:

- SOLID communication DLLs used: version and size
- other communication DLLs used: version and size
- description of the network configuration

Part III

SOLID *SynchroNet* Data Synchronization

Part III shows you how to administer, maintain, configure and tune SOLID *SynchroNet*. It includes basic server administration. Part III contains the following topics:

10

Getting Started with Data Synchronization

This chapter gives detailed instructions on executing sample scripts that demonstrate basic SOLID *SynchroNet* operations. This chapter assumes familiarity of SOLID basic administration functions described in Part II of this manual. Knowledge in the following areas is assumed:

- Basic administration of SOLID (installing, starting, shutting down, configuring network names, etc.). Read relevant chapters in this guide for details.
- Working with SOLID (connecting the database, running SQL statements)
- SOLID SQL procedure programming
- Fundamentals of SOLID *SynchroNet* architecture. Be sure to read *"SOLID SynchroNet Data Synchronization Architecture"* on page 2-1 before you use this chapter.

Before You Begin

SOLID *SynchroNet* is implemented as SQL statements. The data synchronization functions described in this chapter are available through all SOLID APIs (*ODBC, JDBC, Light Client*), SOLID *DBConsole* or *SQL Editor* (teletype). For details on passing SQL to SOLID *SynchroNet* using an API, please refer to the documentation for that particular API.

Before you begin to use the sample scripts, you need to prepare your SOLID *SynchroNet* environment for executing them. Instructions for setting up a SOLID *SynchroNet* evaluation environment are provided in a separate document named **SynchroNet_eval_setup.txt**. This document is located in the SOLID SDK installation root directory.

SynchroNet Implementation

This chapter contains a brief step-by-step example on setting up, configuring and using SOLID *SynchroNet*. Once you have completed all the steps in the order described, you will have a basic SOLID *SynchroNet* environment.

Step	Procedure	Description
1	Getting started	Contains instructions on installing and starting two SOLID *SynchroNet* databases. One database is configured as a master and the other one is configured as a replica. Also defines a sample publication in the master database.
2	Designing a database table	Defines a table both in the master and replica databases that meet the SOLID *SynchroNet* design requirements for implementation.
3	Providing transactions	Contains two transactions implemented as SOLID stored procedures in both the master and replica databases.
4	Implementing SOLID *SynchroNet* functionality	Registers the sample publication, updates the replica database, and saves the transactions for propagation to the master database.
5	Implementing a *SynchroNet* message	Contains SOLID *SynchroNet* SQL commands to be run from the console in order to synchronize the databases.

About the Scripts

This chapter contains instructions on running basic SOLID *SynchroNet* 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 SOLID *SynchroNet* architecture.

All the SQL scripts included in this chapter are also available in the following directory of the SDK package: **`samples/SynchroNet/eval_setup`**

Getting Started

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

- Installed the SOLID Development Kit successfully. For details, read the readme.txt contained in the Development Kit.
- Prepared the environment setup as described in the **SynchroNet_eval_setup.txt.**

Setting Up the Environment

To set up the SOLID *SynchroNet* environment, complete the following steps:

1. Start SOLID *SynchroNet* for the master and replica databases.

Instructions for starting the master and replica databases are found in the **SynchroNet_eval_setup.txt**.

2. Start up the SOLID *SQL Editor* (**solsql.exe**) to execute the SQL scripts in the master database.
3. Move to the SOLID installation root directory and enter the following command:

```
./platform/bin/solsql -O eval.out “tcp 1315” dba dba ./samples/
SynchroNet/eval_setup/master1.sql
```

where:

-O eval.out is an optional parameter that defines the output file for results

dba dba are username and password respectively

master1.sql is the executed SQL script.

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

Note

When you execute SOLID *SynchroNet* scripts you must set the autocommit mode OFF. If you are using SOLID *DBConsole*, the autocommit mode is set ON by default so be sure to set it OFF. If you are using the *SQLEditor* (teletype), the autocommit mode is set OFF by default.

Initializing the master database

MASTER1.SQL

```
-- ********************************************
-- master1.sql
-- Execute in the MASTER database
-- Initializes the master with node name MASTER
-- ********************************************
SET SYNC NODE "MASTER";
COMMIT WORK;

SET SYNC MASTER YES;
COMMIT WORK;
```

4. To define the database, execute the CREATEDB.SQL script described in *"Designing a Database Table"* on page 10-7.

5. To define a publication in the master database, execute the following script:

   MASTER2.SQL

```
-- ***********************************************************
-- master2.sql
-- Creates a publication for table Syncdemo
-- Execute in the MASTER database
-- ***********************************************************


"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 table *syncdemo*. The subscriptions to this publication are incremental since the SYNCHISTORY property is set for this table in both databases.

6. Move to the SOLID installation root directory and enter the following command:

```
./platform/bin/solsql -O eval.out “tcp 1316” dba dba ./samples/sync/
eval_setup/replica1.sql
```

   where:

   -O eval.out is an optional parameter that defines the output files for results

   dba dba are username and password respectively

   replica1.sql is the executed SQL script.

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

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

   - Set the user id and password for connecting the master database for the first time using the following command:

     ```
     SET SYNC USER uid IDENTIFIED BY pwd
     ```

   - Set the master database's connection string for *both* configuration messages in the script using the following command:

     ```
     MESSAGE msg FORWARD TO master_connect
                 TIMEOUT FOREVER
     ```

Note

When you execute SOLID *SynchroNet* scripts you must set the autocommit mode OFF. If you are using SOLID *DBConsole*, the autocommit mode is set ON by default so be sure to set it OFF. If you are using the *SQLEditor* (teletype), the autocommit mode is set OFF by default.

Initializing the replica database

REPLICA1.SQL

```
-- **********************************************************
-- replica1.sql
-- Initialize a replica database with node name "replica1"
-- Each replica must have a unique node name
--
-- Execute in the REPLICA database
--
-- NOTE: AUTOCOMMIT must be set off for MESSAGE handling!
-- **********************************************************
SET SYNC NODE REPLICA1;
COMMIT WORK;

SET SYNC REPLICA YES;
COMMIT WORK;

-- Use temporarily the user ID "DBA" for synchronization
SET SYNC USER DBA IDENTIFIED BY DBA;
COMMIT WORK;
```

```
-- register the replica with the master by sending a
registration message
MESSAGE CFG1 BEGIN;
MESSAGE CFG1 APPEND REGISTER REPLICA;
MESSAGE CFG1 END;
COMMIT WORK;
MESSAGE CFG1 FORWARD TO 'tcp 1315' TIMEOUT FOREVER;
COMMIT WORK;

-- subscribe user information from the master in a separate
message
MESSAGE CFG2 BEGIN;
MESSAGE CFG2 APPEND SYNC_CONFIG('%');
MESSAGE CFG2 END;
COMMIT WORK;
MESSAGE CFG2 FORWARD TO 'tcp 1315' TIMEOUT FOREVER;
COMMIT WORK;

SET SYNC USER NONE;
COMMIT WORK;
```

Designing a Database Table

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

- Generated unique keys to ensure global uniqueness of rows
- Row status column to provide a means for handling update conflicts.
- Time of last update or version number to allow detection of update conflicts.

Read *"Defining a Database Table"* on page 11-10 for a detailed discussion about these requirements. The following SQL creates a table that meets these criteria.

CREATEDB.SQL

```
-- **********************************************************
-- Createdb.sql
-- Creates the sample table to be used in Sync Eval exercise
--
-- Execute in both the MASTER and REPLICA databases
-- **********************************************************
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;
```

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 unique id 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.
- SYNCHISTORY property creates a shadow table for the SYNCDEMO main table. Old versions of updated and deleted rows are moved to this table.

Setting the SYNCHISTORY property on for the table enables incremental publication, which means only modified rows are transferred from the master to the replica when synchronizing the table data. SYNCHISTORY must be active both in master and replica databases.

Define the table both **in the master and replica databases.**

Providing Transactions

Typically, write operations in synchronized architecture are implemented using stored procedures. This allows you to implement business logic into the transactions to handle possible conflicts without violating the application's data integrity and business rules.

The procedure logic in the following scripts are designed to be as simple as possible. The only logic for handling conflicts is implemented in the update procedure. If the update fails 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 following scripts **proced1.sql** and **proced2.sql** at *both* the replica and master databases.

Transaction for inserting rows

The procedure below inserts a row into table *syncdemo*. For simplicity's sake there is no processing for errors. In a real application there should be 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: NO VALIDATION RULES. DUPLICATES HANDLED BY IGNORING
-- the DUPLICATE INSERT
```

continued

```
   -- *********************************************************

   "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;

   END";
   COMMIT WORK;
```

Intelligent transaction for updating

The following procedure updates a row in the table *syncdemo*. 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 infinite 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 TIMESTAMP HAS CHANGED 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
   -- *********************************************************

   "CREATE  PROCEDURE SYNCDEMO_UPDATE
   (
    MACHINEID INTEGER,
    ID INTEGER,
    INTDATA INTEGER,
    TEXTDATA CHAR(20),
```

continued

```
     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 = ? ;
```

continued

```
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,DUMMY) ;
    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 the **SOLID Programmer Guide**.

Using SOLID *SynchroNet* Functionality

You are ready to run the following scripts, which demonstrate basic SOLID *SynchroNet* functionality. This includes:

- Registering a publication to the replica so that once it subscribes to the publication, the replica can receive it. Registering publications allows publication parameters to be validated. This prevents users from accidently requesting subscriptions they do not want, or requesting ad-hoc subscriptions
- 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
 -- *********************************************************


 COMMIT WORK ;
 SELECT *  FROM SYNCDEMO ;
```

Publication registration and inserts at replica

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
      MESSAGE REG_PUBL BEGIN;
```

continued

```
    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.

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** to propagate all local transactions to the master and subscribe the changes of **pub_demo** publication. Finally, the script sends the message to the master database and waits for a reply.

Subscribing a publication with deleted rows

REPLICA3.SQL

```
-- ********************************************************
-- replica3.sql
-- creates a new message with name 'my_msg'
-- append tasks to message my_msg: propagate transactions
--          : subscribe publications
--
-- Execute in the REPLICA database.
--
-- NOTE: AUTOCOMMIT must be off!
-- ********************************************************

MESSAGE my_msg BEGIN ;
MESSAGE my_msg APPEND PROPAGATE TRANSACTIONS ;
MESSAGE my_msg APPEND SUBSCRIBE 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:

Updates from replica

MASTER3.SQL

```
-- ************************************************
-- master3.sql
-- Deletes a row from the sample table
-- Execute in the MASTER database
--*************************************************
DELETE FROM SYNCDEMO WHERE ID =  2;
COMMIT WORK ;
```

Use **select.sql** to verify that replica contains two rows and master 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:**

A conflicting update from 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
-- ******************************************************

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
-- *********************************************************


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 **master4.sql** in the **master:**

   MASTER4.SQL

```
-- *********************************************************
-- master4.sql
-- This script updates one row in the MASTER database
--
-- Execute in the MASTER database
-- *********************************************************
CALL SYNCDEMO_UPDATE (1,3,203,'Row 3 masterchange','1998-05-15
```

continued

```
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
-- *********************************************************


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.

11

Planning and Designing for SOLID *SynchroNet* Applications

This chapter describes the design and planning issues you need to consider before installing and implementing an application that uses SOLID *SynchroNet* data synchronization technology. *Chapter 10,"Getting Started with Data Synchronization"* provided you with a quick overview of S*ynchroNet* functionality. Now you can begin planning for SOLID *SynchroNet* application development and customizing it to meet your own unique business needs. This chapter shows you how to plan and design your multidatabase system to do this. It pinpoints the various areas of the application and database where design issues apply.

Planning for *SynchroNet* Installation

Before installing *SynchroNet*, 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, schedule synchronization, create your infrastructure, and allocate computer and network resources.

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.

Tailoring the Synchronization Process

A distributed SOLID *SynchroNet* system can utilize the off-peak hours of the system. The SOLID *SynchroNet* 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.

A compromise between the overall performance of the database and the data timeliness is often needed. The higher the timeliness requirement of the data, the less scalability you have in the overall system.

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 transactions created in the system are eventually committed in the master database. Similarly, publication data is subscribed 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 subscriptions. 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.

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 multidatabase 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 subscriptions to full publications use more resources, enable publications to be incremental by setting the synchistory property for the tables of the publication. This allows only the master data that has changed in the publication to be sent to the replica database, rather than a full publication. Read *"Creating Publications"* on page 12-22 for details.
- Optimize the frequency of synchronization. Too frequent subscriptions of publications generate unnecessary overhead to the system.
- 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 databases

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
- Subscriptions to publications which include:
    - all inserted and updated rows from the master database
    - primary keys of rows that are deleted from the master database

Designing and Preparing Databases for Synchronization

After you install *SynchroNet* on each machine as instructed in the readme.txt on the SOLID Web site, 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 *SynchroNet* 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.
- Provide user access required for synchronization.
- Set up backups of the master database and large replicas.
- Implement the databases for synchronization.

Each of these topics is described in the following sections.

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 command. You can use SOLID *DBConsole* (interactive or batch mode) or SOLID *SQL Editor* (teletype) to enter the *SynchroNet* command required for set up.

To specify a database 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 database 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 database 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.

Creating the Database Schema

In a multidatabase system, the usage of databases can vary a lot. Therefore you must consider the way databases in your *SynchroNet* 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 *SynchroNet* 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; the *SynchroNet* 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 schemas 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. Read *Appendix E, “SOLID SQL Syntax”*for details.

Guidelines for multi-tier topology

A multi-tier topology contains multiple 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.

Multi-tier topologies are useful in systems that have a wide, geographic distribution and a potentially large number of replica databases that have also local 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

SOLID *SynchroNet* allows multiple, independent databases inside one physical database. These databases are implemented as database catalogs. Each of these catalogs can act as an independent master or replica database. For example, it is possible to create two or more independent replica databases into one physical local database. It is also possible to have one or multiple catalogs in the database that each contain a master database.

Multi-master topologies are useful in environments where a SOLID *SynchroNet* 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-based calling application.

Note that you can combine multi-tier and multi-master topologies.

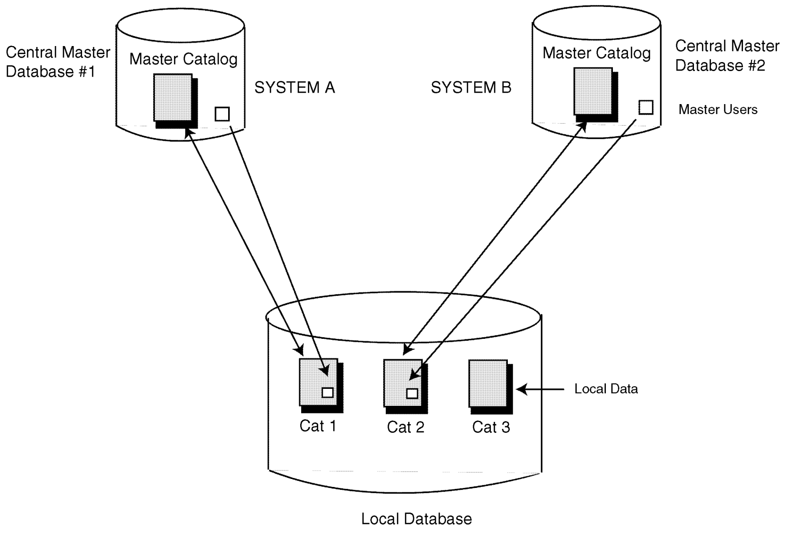
Creating Catalogs

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

- One database can contain any number of catalogs.
- Each catalog of a database can be either a master, replica, or both.
- A catalog can contain multiple schemas. Transactions can access database objects in any catalog.
- A catalog can contain tables from a master and tables from the local replica; thus, transactions can span 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 11–1 below illustrates these guidelines.

Figure 11–1 Multi-Master Model



To create catalogs on masters and replicas, use the CREATE CATALOG command on each master/replica pair. Refer to the CREATE CATALOG command in *Appendix E, "SOLID SQL Syntax"* for details on creating catalogs. Note that the catalog name does not need to be the same in a master and replica.

On Master:

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

On Replica:

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

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 subscribing data to replica.
- Foreign key constraints cannot be used because subscribed publications do not necessarily contain all the data in referenced table.
- Apply *SynchroNet*'s own ALTER TABLE command to set up incremental publications on the master and replica tables. Otherwise full publications (which use more resources) are sent to replicas, rather than just the replica data that has changed from the previous subscription.

  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 replica updates for incremental publication. For details on the ALTER TABLE syntax, read *"Creating Incremental Publications" on page 12-23.*

Design the Logical Database

The data modeling of a multidatabase 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 multidatabase 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 ORDERS
(DB_ID VARCHAR NOT NULL, ID INTEGER, NOT NULL,
... other columns of the table,
(PRIMARY KEY (DB_ID, ID));
```

Detecting update conflicts

The SOLID *SynchroNet* architecture leaves responsibility of all transaction validation to the application developer. This includes the detection of update conflicts. The easiest method is for update conflicts to be detected using an "updatetime" column in each table of the system. Whenever a row is updated, the current 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 already widely used in centralized multiuser systems and it is known as "optimistic locking."

Reporting synchronization errors

It is always possible that an application or system level error can occur during synchronization. Unhandled synchronization errors halt the synchronization message in the master database and require user administration. Therefore, these synchronization errors require easy detection.

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 *"Creating a Synchronization Error Log Table for an Application"* on page 12-34.

Design the Physical Database

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. The *SynchroNet* database engine treats 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 subscriptions to 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;
```

Write load caused by transactions

The write load of the master database sets the practical limits to the scalability of a *SynchroNet* system. In a SOLID *SynchroNet 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.

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 *SynchroNet* 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 can use the ALTER TABLE *SynchroNet* extension, 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 *"Setting Up Databases for Synchronization"* on page 12-20 for details.

```
CREATE TABLE 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 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. 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 unofficial ones. 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.

Handling Concurrency Conflict in Synchronized Tables

SOLID *SynchroNet* 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:

- A replica creates and executes a synchronization script that makes a subscription to a publication.
- Simultaneously, a user updates a row in the replica that will be refreshed by the subscription.
- Before the user commits the transaction, the reply message of the synchronization message arrives at the replica and the engine starts applying the subscription data to the database.
- The user commits the on-line transaction.
- The subscription attempts to modify the same row that the on-line user already modified.

- 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.

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. In this mode, all other concurrent access to the table affected is blocked until the synchronization message has completed.	The command for executing the message in pessimistic mode is: MESSAGE *msgname* EXECUTE PESSIMISTIC
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
SOLID 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

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 *“Implementing Security through Access Rights and Roles”* on page 12-9 for details on implementing synchronization access.

Creating Backups for Fault Tolerance

The critical component of SOLID *SynchroNet* is 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 *SynchroNet* databases. For details, read *“Performing Backup and Recovery”* on page 3-19.

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

Replica databases can be reconstructed from the master database by subscribing 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 *SynchroNet* system is fully implemented and is sending and receiving transactions and messages, be sure to perform periodic backups to the system. Read *“Backing up and Restoring the Master and Replica Databases”* on page 3-19.

Designing the Application for Synchronization

What sets SOLID *SynchroNet* 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.

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.

Providing a User Interface to Manage Synchronization

In a stand-alone replica database, the data contents of the database can be very dynamic. 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 subscribing new (and refreshing existing) data as well as deleting unnecessary local data by dropping the corresponding subscriptions.

Managing the synchronization process

The synchronization process of SOLID *SynchroNet* architecture needs to be implemented at the application level. The synchronization process management contains the following tasks:

- Define the contents of the synchronization process, that is, define which transactions are propagated to the master and which publications are subscribed 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 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 architecture. For instance, the network may be down when synchronization has been 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 SOLID *SynchroNet*.

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 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 *SynchroNet*’s easy-to-use model. Read *“Designing and Implementing Transactions”* on page 12-29 for details on creating transactions with SOLID *Intelligent Transaction*.

12 Implementing a SOLID *SynchroNet* Application

This chapter describes the basic tasks required to implement synchronization. These tasks, demonstrated briefly in *Chapter 10,"Getting Started with Data Synchronization"* are described in more detail in this chapter. You are also introduced to *SynchroNet* SOLID SQL, which are extensions that allow you to set up and define your *SynchroNet* installation.

The topics in this chapter include:

- Using *SynchroNet* commands
- Implementing messages for the synchronization process
- Implementing synchronization access rights and roles
- Specifying databases for synchronization
- Creating publications and subscriptions
- Implementing SOLID *Intelligent Transaction*

Using *SynchroNet* Commands

SynchroNet commands are extensions to SOLID SQL. They allow you to manage distributed data by letting you specify synchronization operations. You can execute *SynchroNet* commands using the SOLID *DBConsole* in batch or interactive mode, or SOLID *SQL Editor* (teletype). Read *Chapter 5,"Using SOLID Data Management Tools"*for more details.

Types of *SynchroNet* Commands

SynchroNet commands let you perform such implementation tasks as registering replica databases, implementing access rights, and creating publications. In addition, you administer *SynchroNet* using many of these commands. 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 *SynchroNet* commands are grouped into the following categories and their usage is described in this chapter. Refer to *Chapter 13,"SOLID SynchroNet SQL Reference"*for an alphabetical list of commands.

Database Configuration Commands

These commands are used to set up and configure the *SynchroNet* databases.

DROP MASTER

DROP REPLICA

REGISTER REPLICA

SET SYNC *master_or_replica*

SET SYNC CONNECT

SET SYNC NODE

SET SYNC PARAMETER

SET SYNC USER

Security Commands

These commands are used to set up security for multi-level and multi-master environments.

ALTER USER SET MASTER

ALTER USER SET PUBLIC | PRIVATE

Publication Commands

These commands are used to create, maintain, and subscribe to publications.

ALTER TABLE SET SYNCHISTORY | NOSYNCHISTORY

CREATE SYNC BOOKMARK

CREATE PUBLICATION

DROP PUBLICATION

DROP PUBLICATION REGISTRATION

DROP SUBSCRIPTION

DROP SYNC BOOKMARK

EXPORT SUBSCRIPTION

IMPORT

MESSAGE APPEND REGISTER | UNREGISTER PUBLICATION

MESSAGE APPEND SUBSCRIBE

GRANT SUBSCRIBE ON

REVOKE SUBSCRIBE ON

Intelligent Transaction Control

These commands are used for executing and saving transactions in the replica database for propagation to the master, defining parameters, and locking tables for concurrency control.

GET_PARAM ( )

PUT_PARAM ( )

SAVE

SAVE PROPERTY

Message Commands

These commands are used to send specific information between the master and replica databases and to recover from failed messages.

MESSAGE APPEND PROPAGATE TRANSACTIONS

MESSAGE APPEND SUBSCRIBE

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

Building Messages for the Synchronization Process

The “implementation vehicle” of the synchronization process is the *SynchroNet* messaging architecture. It is a store and forward messaging architecture that is built inside *SynchroNet*. It is capable of transferring messages reliably between a replica and the master database.

The synchronization process of SOLID *SynchroNet* architecture consists of two different tasks:

- **propagating transactions** to the master database
- **subscribing publications** to a replica

A combination of these tasks (containing the propagation command, subscriptions, 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 subscribing data from that table. Both of these actions are permitted in the same synchronization message as shown in Example 12–1:

Example 12–1 Synchronization Messaging

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

Beginning messages

You must explictly 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 server. registering a replica to a database system catalog other than the master. Be sure to also set the autocommit to off.

Note

You use the optional TO clause if you sending a message that contains the REGISTER REPLICA command and are registering a replica to a database system catalog other than the base catalog. For details on registration, read *"Registering Replicas with the Master Database"* on page 12-20.

Propagating Transactions from Replica to Master

The MESSAGE APPEND PROPAGATE TRANSACTIONS command lets you propagate local transactions from the replica to the master database. The syntax is:

```
MESSAGE unique_message_name APPEND
        [PROPAGATE TRANSACTIONS [WHERE property_name (=|<|<=|>|>=|<>)
        'value_string' | ALL)]
```

You use the WHERE clause to propagate only those transactions where the *property_name* meets specific criteria. The transactions property can be set with the SAVE PROPERTY command. Use keyword ALL to propagate statements that do not contain any 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. For details on saved properties, read *"Using the Transaction Bulletin Board for Parameter Passing"* on page 12-31.

Subscribing Publication Data from Master to Replica

The MESSAGE APPEND SUBSCRIBE command lets you subscribe to a publication from the master database. The syntax is:

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

Using this command, you can provide parameters to the publication (if they are used in the publication) to narrow the scope of the replica's subscription.

Ending Messages

It's good practice to explictly end each synchronization message with the MESSAGE END command. This command makes the message persistent in the replica by saving it before it is sent to the master database. Note that after ending the message, you must be sure to commit the message by providing the COMMIT WORK command. The syntax for ending the message is:

MESSAGE *unique_messsage_name* END

Forwarding Messages to the Master Database

After a message has ended, with the MESSAGE END command, you send it to the master database using the MESSAGE FORWARD command. Each sent message is issued a reply message from the master database, which is fetched from the replica database. 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 command described in the following section.

Requesting a Reply Message from the Master Database

If the TIMEOUT in the MESSAGE FORWARD command is not defined, the message is forwarded to the master and the replica does not fetch 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*]

Configuring *SynchroNet* 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. SOLID *SynchroNet* 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 length of messages sent from a replica database to the master, while SYS_R_MAXBYTES_IN sets the 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 set, 2GB is used.

To set these parameters, use the SET SYNC PARAMETER command 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 Size of the Commit Block

If the reply of a sent message will contain subscriptions 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 commands. The syntax 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
```

Note that setting the size of the commitblock has a positive impact on the performance of the replica databases. However, data integrity cannot be guaranteed if concurrent online usage occurs simultaneously. Therefore, we recommend that you disconnect all online users from the replica database when you use the COMMITBLOCK option.

Executing a Synchronization Process

The synchronization process is always initiated and largely also controlled from the replica database.

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

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

   ```
   EXEC SEQUENCE SYNCMSGNUMBER.NEXT INTO MSGNUMBER;
   ```

```
MSGNAME := ’MSG’ + CONVERT_CHAR(MSGNUMBER);
```

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

2. Append the synchronization tasks (propagate transactions and subscribe 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

The following sample script summarizes how to execute a typical synchronization process.

```
-- create a new message with name 'my_msg'
-- AUTOCOMMIT must be off!
MESSAGE my_msg BEGIN ;
-- append tasks to message: propagate transactions
MESSAGE my_msg APPEND
PROPAGATE TRANSACTIONS ;
-- append tasks to message: subscribe publications
MESSAGE my_msg APPEND
SUBSCRIBE ORDERS_BY_SALESPERSON ('1') ;
MESSAGE my_msg APPEND
SUBSCRIBE SYS_PARAMETERS ;
--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 max. 100 seconds
MESSAGE my_msg GET REPLY TIMEOUT 100 ;
COMMIT WORK ;
```

Implementing Security through Access Rights and Roles

Implementing access rights and roles is the method that SOLID *SynchroNet* uses to enforce security throughout the system. This section describes basic principles of *SynchroNet* security and how to use *SynchroNet* commands to set it up.

How *SynchroNet* Security Works

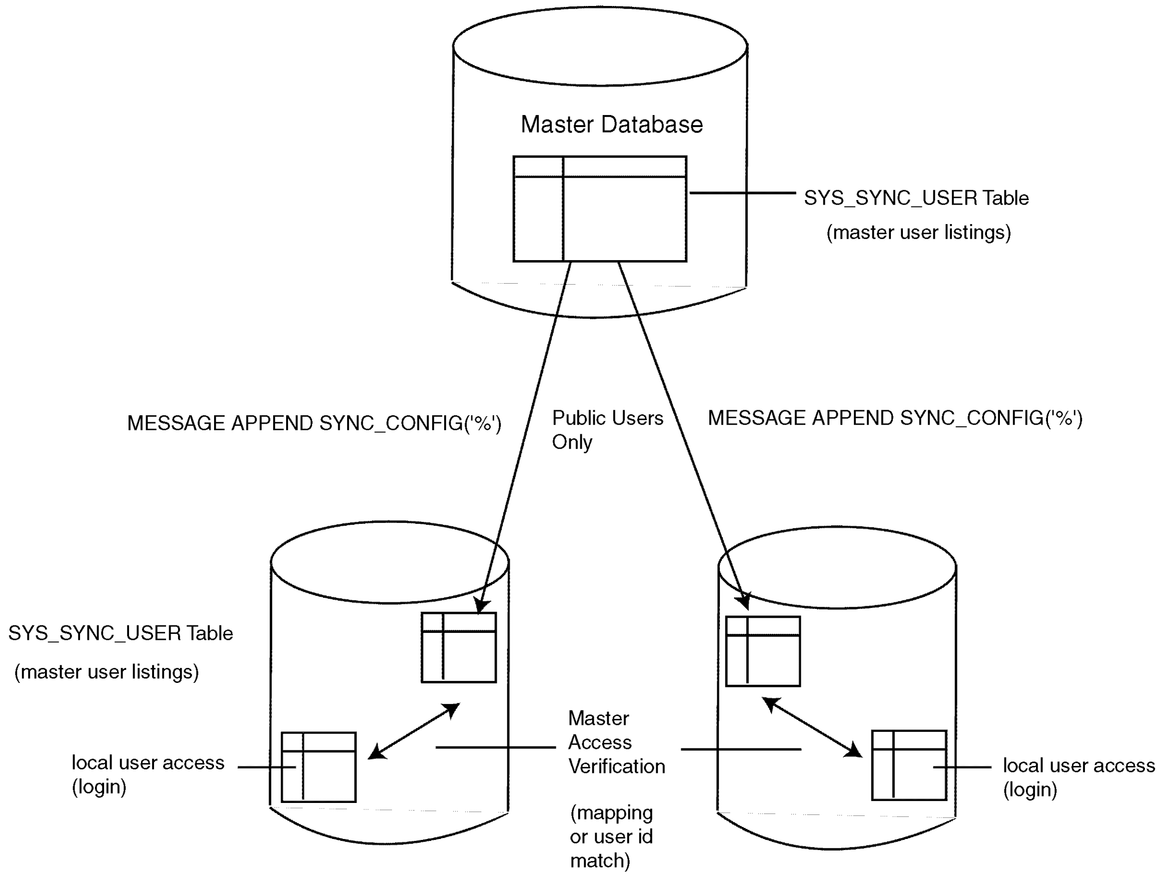
The SOLID *SynchroNet* security model is based on the following principles:

- Both local users and master users exist in the replica database of a *SynchroNet* 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 functions such as SAVE *sql_statement* or MESSAGE commands.
- 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 the *"Access Rights Summary"* on page 12-16.

Figure 12–1 SynchroNet User Access Rights


Master Database
SYS_SYNC_USER Table
(master user listings)
MESSAGE APPEND SYNC_CONFIG('%')
Public Users Only
MESSAGE APPEND SYNC_CONFIG('%')
SYS_SYNC_USER Table
(master user listings)
local user access (login)
Master Access Verification
(mapping or user id match)
local user access (login)

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

Managing Master Users

Master users control all access to the data synchronization functions of SOLID *SynchroNet*. 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 the local user id with the master user id.

Mapping Replica User ID with Master User ID

To specify the mapping for a replica user id with a master user id, you use the ALTER USER SET MASTER command. When you use the ALTER USER SET MASTER command you provide the master user names and passwords for those users you want to designate as master users. The same local user can be mapped to multiple masters.

After you have completed the mapping, when a user logs into a replica database, *SynchroNet* checks to see which master user id is mapped to the currently used local user id. If no mapping is specified, by default, *SynchroNet* 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 12–1, "SynchroNet User Access Rights"* which illustrates this concept.

Setting Public and Private Users

Administrators can alter users in the SYS_SYNC_USERS table (with the ALTER SYNC_CONFIG USER PRIVATE | PUBLIC command) to designate them as *private* or *public*. If the PRIVATE option is set for a user, this user's information is 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_SYNC_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 *"ALTER USER"* on page 13-3.

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 *SynchroNet* system.

Setting a user as PRIVATE allows *SynchroNet* Administrators to configure a *SynchroNet* system where no user accounts with administrative rights are ever sent to the replica. This provides an extra measure of security by preventing a DBA's password from ever becoming public. Should the DBA password become exposed, all replicas of the system would need to be dropped and recreated after the password is changed in the master database.

DBAs or *SynchroNet* Administrators can also set those users (especially if there is a large number of them) who 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 propagating transactions) fails with a security error.

Determining Access Rights of Subscriptions and Transactions

Synchronization messages are labeled with the master username of the creator of the message. *SynchroNet* 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.

Setting Up Access Rights

The following sections identify the access rights required to implement a *SynchroNet* 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, the procedure creator must be a master user.

Grant master users with appropriate access rights in the master database to the tables they use for 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, they take effect only if the user:

- accesses the table or object on which the rights are set for the first time
-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 using the ALTER USER SET MASTER command.

When setting up access rights, you can use the following *SynchroNet* 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 *"Using SOLID SQL for Data Management"* on page 4-1 for details.

To grant users access to publications, use:

GRANT SUBSCRIBE ON *publication_name* TO *username*

Read *"Granting SUBSCRIBE Access"* on page 3-38 for details.

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

ALTER USER *replica_user* SET MASTER *master_name* USER *user_specification*

Read *"ALTER USER"* on page 13-3 for details.

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 user's username. When the transaction is executed in the master database, it uses the access rights defined for this username.

When the *SynchroNet* 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 different masters and then change the active catalog using the SET CATALOG command so that the user can access the master of that catalog. For example, in a replica database assume there is one local Database

Administrator with a user id of DBA and a password of DBA. This user is mapped to the CALENDARUSER master user of the calendar application and the NEWSUSER master user of the news application, etc. The SET CATALOG command is used to set the current catalog to either CALENDAR or NEW. If CALENDAR catalog is set, then the current master user is automatically set to the CALENDARUSER master user. Similarly, if NEW is set, the current master user is set to NEWSUSER.

If mapping is not defined, the first *SynchroNet* data synchronization operation (for example, SAVE or MESSAGE command) must return the "no active master user" error.

Creating User rights to Publications and Tables

Master users who define publications must have read and write access rights to tables referenced by the publications. This also allows them to drop the publication.

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

Master users must have subscribe access rights to a publication in order use it, as well as the right to make modifications to the actual subscription tables in the replica database. When subscription rows are inserted (or deleted) in a replica, *SynchroNet* verifies that the subscriber has rights to modify the table.

Read *"Changing Access Rights to Publications"* on page 3-38 for details on granting publications access rights.

Creating the Replica Registration User

When a new SOLID *SynchroNet* 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 with 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 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 command:

SET USER SYNC NONE

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

```
User defintion 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 in a replica database. If you want to designate a different master user for a replica, you must map the replica ID on replica database with the master ID on the master database. For details, read *"Mapping Replica User ID with Master User ID"* on page 12-11.

For details on how the master users control user access of data synchronization functions, read "Managing Master Users" on page 12-11.

Implementing Special *SynchroNet* Roles

SOLID *SynchroNet's* two special roles for performing synchronization operations are:

- SYS_SYNC_ADMIN_ROLE

  The is an administration role for performing *SynchroNet* administrative operations, such as deleting messages. Anyone with this access 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.

Access Rights Summary

Following is a comprehensive summary of the access rights required to execute each *SynchroNet* 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.

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
MESSAGE APPEND SUBSCRIBE	Subscribe to a publication	Valid master user
MESSAGE APPEND PROPAGATE TRANSACTIONS	Propagate transactions	Valid master user

Command	Task	Access rights requirements
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 subscribe to the publication.	Subscribe 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 master database as part of registration or to remove a node name when dropping a synchronized database and removing registration.	SYS_SYNC_ADMIN_ROLE
SET SYNC PARAMETER	Set database parameters in a synchronized database	SYS_SYNC_ADMIN_ROLE
SET SYNC REPLICA | MASTER YES | NO	Designates the database as a replica 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 *SynchroNet* commands in master database.

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
CREATE PUBLICATION	Define a publication in the master database	Valid master user who has full access to the tables of the publication.

Command	Task	Access right requirements
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 SUBSCRIBE ON	Grant access rights on a publication to a user or role defined in the master database.	Owner of the table, or DBA
REVOKE SUBSCRIBE ON	Revoke access rights on a publication to a user or role defined in the master database	Owner of the table, or DBA
DROP PUBLICATION	Drop a publication in the master database	Valid master user
EXPORT SUBSCRIPTION	Export master data for export 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
MESSAGE DELETE CURRENT TRANSACTION	Delete current (failed) transaction of a synchronization message to recover from an error	SYS_SYNC_ADMIN_ROLE
MESSAGE FROM REPLICA EXECUTE	Execute a failed message from the replica in the master database	SYS_SYNC_ADMIN_ROLE
DROP SUBSCRIPTION REPLICA	Drop a replica's subscription to a publication in the master	SYS_SYNC_ADMIN_ROLE
DROP REPLICA	Drop a replica database from the master database	SYS_SYNC_ADMIN_ROLE
SET SYNC MASTER | REPLICA YES | NO	Designate the database as a master	SYS_SYNC_ADMIN_ROLE
SET SYNC USER NONE	Makes current master user inactive in the current database connection	SYS_SYNC_ADMIN_ROLE
SET SYNC PARAMETER	Set database parameters in the master database	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 to remove a node name when dropping a synchronized database and removing registration.	SYS_SYNC_ADMIN_ROLE

Setting Up Databases for Synchronization

After databases are 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 *SynchroNet* database. You can use the SOLID *DBConsole* to enter the *SynchroNet* commands required for set up.

Configuring the Master Database(s)

Before you begin, be sure you have defined your master database(s). For details see *"Defining Master and Replica Databases"* on page 11-4.

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

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

Registering Replicas with the Master Database

Before you begin, be sure to SET AUTOCOMMIT off for MESSAGE handling and that you have defined a master username and password for registering replicas to the master database, and that you know the name of the catalogs in your environment. Also be sure you have defined your replica databases; for details see *"Defining Master and Replica Databases"* on page 11-4.

In each replica database, perform the following steps:

1. If a local database contains more than one replica, create a catalog for the new replica. For example:

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

2. Provide a node name that is unique within the domain. For example:

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

Note

In a large environment, we recommend that you name replicas with logical names that are derived, for example, from the server's logical name or location.

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

3. When you have entered the master user 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 typically has no other rights except registration. If the registration user is still active, this prevents other master users from performing synchronization operations since the following error message would be returned:

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

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;
   ```

Note

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

MESSAGE CFG1 BEGIN TO *master_node_name*

The MESSAGE FORWARD command sends the message to the master database after it 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 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;
```

Creating Publications

A *SynchroNet* **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 *SynchroNet* instead sends a snapshot of the data changed to the replicas.

SynchroNet 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 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 and an incremental publication sends only data that has changed since the previous subscription.

▶ **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 *Creating Incremental Publications* in the following section.

Creating Incremental Publications

In order to make a publication incremental, you configure the tables of the publication to gather the synchronization history data using the following command:

ALTER TABLE *table_name* SET SYNCHISTORY

By default, this property is set to NOSYNCHISTORY. When this command is set to SYNCHISTORY, after the first subscription, subsequent subscriptions to a specific publication send the replica database only new and modified rows when the data in the table is synchronized. If the *table_name* specified in the command is referenced by an existing publication, you will receive an error message.

You need to use this command in the synchronized tables. For example:

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

This commands 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 subscriptions, the unnecessary data of the shadow table is automatically deleted.

Using the Create Publication Command

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 ] ;
            [ result_set_definition...]
    END
```

The CREATE PUBLICATION command 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 retrieved from the master database in one transaction. Similarly, the publication data is written to the replica database in one transaction. This guarantees that the data content of a publication is always internally consistent.

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 different tables. These tables can be either independent or there can 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.

Here is a typical publication:

```
CREATE PUBLICATION ORDERS_BY_SALESPERSON
(SALESPERSON_ID VARCHAR)
BEGIN
   RESULT SET FOR ORDER
   BEGIN
        SELECT * FROM ORDER
        WHERE SM_ID = :SALESPERSON_ID AND STATUS = 'ACT';
        RESULT SET FOR ORDER_LINE
        BEGIN
             SELECT * FROM ORDER_LINE
             WHERE ORDER_ID = ORDER.ID;
        END
        DISTINCT RESULT SET FOR CUSTOMER
        BEGIN
             SELECT * FROM CUSTOMER
             WHERE ID = 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 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 ORDER and ORDER_LINE is 1-N. The data is linked together using the ID column of 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 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 ORDER table and ID column of the CUSTOMER table. The keyword DISTINCT ensures that the same customer information is brought to 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, an 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.

For better performance, avoid nesting result sets whenever possible. In the following examples, better performance is achieved by rewriting an equivalent unnested version of CREATE PUBLICATION statement:

Nested version:

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

Unnested version:

```
CREATE PUBLICATION UNNESTED (IN_ORDER_ID INTEGER)
BEGIN
RESULT SET FOR ORDER
   BEGIN
   SELECT * FROM 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;
```

Publications are fully independent from each other. This means you cannot define dependencies between publications. Avoid, especially, the use of overlapping publication definitions. This includes publication definitions with overlapping tables and WHERE conditions. Publication definitions overlap when two publications access the same table even when different rows are retrieved if WHERE conditions overlap.

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 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.

Note that publication definitions overlap when two publications access the same table even when different rows are retrieved.

After using CREATE PUBLICATION, it is important to commit the transaction. If a transaction that defines a publication is uncommitted at the master, the system issues an error message stating that the publication does not exist when an attempt is made to subscribe the publication from the master database.

Publication data is requested from the master database using the MESSAGE APPEND SUBSCRIBE *publication_name* command. For details read *“Subscribing to Publications”* on page 12-27.

Subscribing to Publications

Replica databases use *subscriptions* to request publication data from the master. Subscriptions depend on the publication definition in the master database. There is no need to define the publications in the replica database, but you must be sure to register publications in the replica so they can be subscribed to. Users are unable to subscribe to publications that are not registered in the replica. Registering publications allows publication parameters to be validated. This prevents users from accidently requesting subscriptions they do not want, or requesting ad-hoc subscriptions.

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

MESSAGE APPEND REGISTER PUBLICATION *publication_name*

For example:

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

Also be sure users have access rights to the master tables in a publication that they are using. Use the GRANT SUBSCRIBE ON command to provide user access rights to publication. For details, read *"Managing Access Rights"* on page 3-37.

▶ **Note**

If the CREATE PUBLICATION command is executed in the replica database, the actual publication definition is stored in the replica, but it not used in subscriptions.

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
    [SUBSCRIBE *publication_name*[(*publication_parameters*)]
    [FULL]]

For example:

```
MESSAGE my_msg APPEND
          SUBSCRIBE ORDERS_BY_SALESMAN ('1') ;
```

The initial subscription is always a full publication and all data contained in the publication is sent to the replica database. Subsequent subscriptions for the same publication can contain only the data that has been changed since the prior subscription. 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 *"Creating Incremental Publications"* on page 12-23 for details on setting tables to track modifications for incremental publication.

When you use the keyword FULL with SUBSCRIBE 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 very large transaction. In such cases, the size of a single transaction of a synchronization message can be limited with the COMMITBLOCK option.

```
MESSAGE my_msg GET REPLY COMMITBLOCK 1000 ;
```

When COMMITBLOCK is used, no other users should access the replica database during the synchronization because transactional integrity cannot be guaranteed.

Dropping Subscriptions and Excluding Subscription Data

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. This command also serves the purpose of deleting unnecessary data from the replica database. For details, refer to *"Modifying Publications and Subscriptions"* on page 3-37.

It is also possible to have rows for local use only at a replica by subscribing publications with a 'where' constraint excluding the local rows.

Unregistering or Dropping Publication Registrations

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

MESSAGE APPEND UNREGISTER PUBLICATION *publication_name*

For example:

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

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

DROP PUBLICATION REGISTRATION *publication_name* FROM

    MASTER *master_name*

For example,

```
DROP PUBLICATION REGISTRATION publ_customer FROM Master1;
```

Designing and Implementing Transactions

Traditionally, a transaction is an atomic set of database operations that changes a database from one valid state to another valid state. The valid state of a database means that no integrity and 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 permits the creation of an order if the customer's credit limit in the database is not exceeded. A "create order" transaction may consist of inserting a row to the ORDER table and inserting another row to the INVOICE table. If inserting an order fails because of a customer credit limit is exceeded, no invoice about the order should be inserted either. The INSERT_ORDER procedure informs 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:

The SOLID *Intelligent Transaction* refers to 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 since 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 must be done by the transaction itself.
- They are alone responsible for the consistency of the master database.

Example 12–2 creates a "create order" transaction in a simple order entry application. The following sections use this example to illustrate how to implement SOLID *Intelligent Transaction*.

Example 12–2 Create Order Transaction

```
-- Make changes to local database
CALL INSERT_ORDER(…) ;
CALL UPDATE_CUSTOMER_CREDIT(…) ;
-- Save a parameter 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;
```

Updating Local Data

In Example 12–2, 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 SOLID *SynchroNet* architecture, local changes remain local unless the statements and parameters of a transaction are explicitly **saved** for later propagation.

Saving the Transaction for Later Propagation

It is also possible to just save the statements for later propagation and not update the local database at all. In this excerpt from Example 12–2, 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*

Using the Transaction Bulletin Board for Parameter Passing

It is also possible to save properties to the transaction. Statements (procedure calls) of the transaction communicate with each other using a parameter bulletin board. The bulletin board is visible to all statements of the transaction. Each catalog has a separate set of bulletin board values that are not visible to other catalogs. What this means logically is that different replicas and masters each have their own set of bulletin board values.

The statements of a transaction can leave data on the bulletin board as well as read data from there.

In this excerpt from Example 12–2, the transaction has one saved property with the name ‘priority’ and value ‘1’.

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

You assign parameters to a transaction in the form of “saved properties.” The parameter appears to the parameter bulletin board of the transaction when it is executed in the master database.

Note that this parameter can also be used, for example, as a search criteria 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’;

The syntax for saving a property to a transaction is:

SAVE PROPERTY *property_name* VALUE *property_value*

Note

When implementing transactions that are intellingent for conflict resolution, be sure to set autocommit OFF to prevent losing bulletin board parameter values. The bulletin board provides a means for separate statements (procedure calls) to communicate with other statements in a transaction using the bulletin board parameters. The life cycle of a bulletin board parameter is one transaction; that is, it is visible only in that transaction that has set the value. If autocommit is ON, then each statement is a separate transaction and bulletin board values are lost.

Other SQL Extensions related to parameter bulletin board are PUT_PARAM(), GET_PARAM() and SET SYNC PARAMETER. Refer to *Chapter 6*, *SynchroNet Command Reference* for details on these commands.

Creating Stored Procedures

The logic needed to ensure the physical and logical database consistency must be written in the stored procedures using the SOLID SQL stored procedure language. These procedures can use the parameter bulletin board for communicating with each other. In Example 12–3, 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 the previous Example 12–2.

Example 12–3 INSERT_ORDER Stored Procedure Example

```
"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 checkings should go here...
-- ...
-- End of validation

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

-- Insert the order row to the database.
-- The STATUS value can be either 'OK' or 'FAIL'
EXEC SQL PREPARE INS_ORD
   INSERT INTO ORDER (ORD_ID, CUST_ID, STATUS, ...)
   VALUES (?,?,? ...);
EXEC SQL EXECUTE INS_ORD
   USING (ORD_ID, CUST_ID, STATUS...);
END";
```

The following procedure updates the balance of a given account:

```
"CREATE PROCEDURE UPDATE_CUST_CREDIT
```

```
(ACC_NO 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_NO = ?;
EXEC SQL EXECUTE UPD_CREDIT
   USING (AMOUNT, ACC_NO);
EXEC SQL CLOSE UPD_CREDIT;
END";
```

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 row.

Validating *Intelligent Transaction*

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 versions of the same data. When a replica transaction that 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 case of an update conflict, select the “losing” operation.
- 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 usually take into consideration all imaginable transaction validation errors, those 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. Quite often, 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.

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 to ORDER table
- insert multiple rows to ORDER_LINE table
- update USED_CREDIT information of CUSTOMER table
- insert an entry to the ACCOUNT_TRANSACTION table of the bookkeeping part of the application
- update the STOCK_BALANCE column of multiple rows of the PRODUCT table to update the warehouse balance of ordered products

In this example, numerous things can go wrong when the transaction is propagated to the master database. For instance:

- The customer does not have sufficient credit for the order. Therefore the entire order is invalid.
- 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 to 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
```

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, that is, resolve 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.

Specifying Automatic Recovery from Fatal System 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 thus 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.

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
```

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, issuing directly the ROLLBACK WORK or COMMIT WORK command inside the procedure is never allowed.

13 SOLID *SynchroNet* SQL Reference

This chapter is a function reference to the SOLID *SynchroNet* SQL commands, which are extensions to SOLID SQL. These *SynchroNet* extensions support database synchronization functionality.

ALTER TABLE SET SYNCHISTORY | NOSYNCHISTORY

ALTER TABLE *table_name*

SET SYNCHISTORY **|** NOSYNCHISTORY

Usage

This command specifies whether to use the incremental publications mechanism of SOLID *SynchroNet* architecture. By default, SYNCHISTORY is not on. When this command is set to SYNCHISTORY for a specified table, a shadow table is automatically created to gather old versions of updates or deleted rows of the main table. The versioned data contains rows that were modified or deleted from the main table. Versioned data is automatically deleted from the database when there are no longer any replicas that need the data to fulfill subscriptions.

You can use this command even if data currently exists in a table; however ALTER TABLE SET SYNCHISTORY can only be used if the specified table is not referenced by an existing publication.

Note that by altering existing database tables for versioning, you can convert existing unsynchronized databases for synchronization. Once versioning is implemented using ALTER TABLE SET SYNCHISTORY, the tables can be updated through subscriptions based on an incremental publication.

SET SYNCHISTORY must be specified in the tables of both master and replica databases.

Usage in Master

Use this command in the master to enable incremental publications on a table.

Usage in Replica

Use this command in the replica to enable incremental subscriptions on a table.

Example

```
ALTER TABLE myLargeTable SET SYNCHISTORY ;
ALTER TABLE myVerySmallTable SET NOSYNCHISTORY ;
```

Return values

Error code	Description
13047	No privilege for operation
13100	Illegal table mode combination
13134	Table is not a base table
25038	Table is referenced in publication *publication_name*, drop or alter operations are not allowed
25039	Table is referenced in subscription to publication *publication_name*, drop or alter operations are not allowed.

ALTER USER

ALTER USER *replicauser* SET MASTER *mastername* USER *user_specification*

where:

user_specification ::= *masteruser* IDENTIFIED BY *masterpassword* **|** NONE

ALTER USER *username* SET PUBLIC **|** PRIVATE

Usage

The following command is used to map replica user ids to specified master user ids.

ALTER USER *replicauser* SET MASTER *mastername* USER *user_specification*

Mapping user ids is useful for implementing security in a multi-master or multi-tier synchronization environment. In such environments, it is difficult to maintain the same username and passwords in separate, geographically dispersed databases. For this reason mapping is effective.

Only those with SYS_SYNC_ADMIN_ROLE can map users. To implement mapping, an administrator must know the master user name and password. Note that it is always a replica user id that is mapped to a master user id. If NONE is specified, the mapping is removed.

All replica databases are responsible for subscribing to the SYNC_CONFIG system publication to update user information. Public master user names and passwords are downloaded, during this process, to a replica database using the MESSAGE APPEND SYNC_CONFIG command. Through mapping of the replica user id with the master user id, the system determines the currently active master user based on the local user id that is logged to the replica database. Note that if during SYNC_CONFIG loading, the system does not detect mapping, it determines the currently active master user through the matching user id and password in the master and the replica.

For more details on using mapping for security, read *"Implementing Security through Access Rights and Roles"* on page 12-9.

It is also possible to limit what master users are downloaded to the replica during SYNC_CONFIG loading. This is done by altering users as private or public with the following command:

```
ALTER USER username SET PRIVATE | PUBLIC
```

Note that the default is PUBLIC. If the PRIVATE option is set for the user, that user's information is not included in a SYNC_CONFIG subscription, even if they are specified in a SYNC_CONFIG request. Only those with DBA authority or SYS_SYNC_ADMIN_ROLE can alter other users publicly.

This allows administrators to ensure no user ids with administration rights are sent to a replica. For security reasons, administrators may want to ensure that DBA passwords are never public, or there may be a large number of users who require no access to the replica.

Usage in Master

You set user ids to PUBLIC or PRIVATE in a master database.

Usage in Replica

You map a replica user id to a master user id in a replica database.

Example

The following example maps a replica user id *smith_1* to a master user id *dba* with a password of dba.

```
ALTER USER SMITH_1 SET MASTER MASTER_1 USER DBA IDENTIFIED BY DBA
```

The following example shows how users are set to PRIVATE and PUBLIC.

```
-- this user should not be downloaded to any replica
ALTER USER dba SET PRIVATE;

-- this user should be downloaded every replica
ALTER USER salesman SET PUBLIC;
```

Return values

Error code	**Description**
13047	No privilege for operation
13060	User name *xxx* not found
25020	Database is not a master database
25062	User *user_id* is not mapped to master *user_id*
25063	User *user_id* is already mapped to master *user_id*

CREATE PUBLICATION

```
"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 ] ;
        [ result_set_definition...]
        [ DISTINCT]
    END
```

NOTE: *Search_condition* can reference *parameter_definitions* and/or columns of replica tables defined on previous (higher) levels.

Usage

Publications define the subsets of data that can be subscribed from the master to the replica database. A publication is always internally consistent, that is, its data has been read from the master database in one transaction and the data is written to the replica database in one transaction.

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 different tables. The tables of the publication can be either independent or there can be relations between the tables. If there is a relation between tables, the result sets must be nested. 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.

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.

The *replica_table_name* can be different form the *master_table_name*. The publication definition provides the mapping between the master and replica tables.

Note that the initial subscription is always a full publication and all data contained in the publication is sent to the replica database. Subsequent subscriptions for the same publication can contain only the data that has been changed since the prior subscription. This is known as an *incremental publication*. To enable usage of incremental publications, SYNCHISTORY has to be set ON for tables included in the publication in both the master and replica databases. For details, read *"DROP PUBLICATION REGISTRATION" on page 13-11*.

Usage in Master

You define the publication in the master database to enable the replicas to subscribe to it.

Usage in Replica

There is no need to define the publications in the replicas. Publication subscription functionality depends on the definitions only at the master database. If this command is executed in a replica, it will store the publication definition to the replica.

Example

The following sample publication retrieves data from customer table using the area code of the customer as search criterion. For each customer, the orders and invoices of the customer (1-N relation) as well as the dedicated salesman of the customer (N-1 relation) are also retrieved.

```
"CREATE PUBLICATION PUB_CUSTOMERS_BY_AREA
      (IN_AREA_CODE VARCHAR)
BEGIN
      RESULT SET FOR CUSTOMER
      BEGIN
         SELECT * FROM CUSTOMER
            WHERE AREA_CODE = :IN_AREA_CODE;
          RESULT SET FOR ORDER
          BEGIN
            SELECT * FROM ORDER
            WHERE CUSTOMER_ID = CUSTOMER.ID;
          END
          RESULT SET FOR INVOICE
         BEGIN
            SELECT * FROM INVOICE
            WHERE CUSTOMER_ID = CUSTOMER.ID;
         END
         DISTINCT RESULT SET FOR SALESMAN
         BEGIN
            SELECT * FROM SALESMAN
            WHERE ID = CUSTOMER.SALESMAN_ID;
         END
      END
END";
```

Return values

Error Code	Description
13120	Too long name for publication
25015	Syntax error: *error_message*, line *line_number*
25033	Publication *publication_name* already exists
25049	Referenced table *table_name* not found in subscription hierarchy
25061	Where condition for table *table_name* must refer to an outer table of the publication

CREATE SYNC BOOKMARK

CREATE SYNC BOOKMARK *bookmark_name*

Usage

This command creates a bookmark in a master database. Bookmarks represent a user-defined version of the database. It is a persistent snapshot of a Solid database, which provides a reference for performing specific synchronization tasks. Bookmarks are used typically to export data from a master for import into a replica using the EXPORT SUBSCRIPTION command. Exporting and importing data allows you to create a replica from a master more efficiently if you have databases larger than 2GB.

To create a bookmark, you must have administrative (DBA) or *SynchroNet* Administrative privileges (SYS_SYNC_ADMIN_ROLE). There is no limit to the number of bookmarks you can create in a database. A bookmark is created only in a master database. The system issues an error if you attempt to create a bookmark in a replica database.

If a table is set up for synchronization history with the ALTER TABLE SET SYNCHISTORY command, a bookmark retains history information for the table. For this reason, use the DROP SYNC BOOKMARK command to drop bookmarks when they are not longer needed. Otherwise, extra history versions will increase disk space usage.

When you create a new bookmark, the system associates other attributes, such as creator of the bookmark, creation data and time, and a unique bookmark ID. This metadata is maintained in the system table SYS_SYNC_BOOKMARKS. For a description of this table, refer to *"SYS_SYNC_BOOKMARKS"* on page A-5.

Usage in Master

Use the CREATE SYNC BOOKMARK command to create a bookmark in a master database.

Usage in Replica

The CREATE SYNC BOOKMARK command cannot be used in a replica database.

Example

```
CREATE SYNC BOOKMARK BOOKMARK_AFTER_DATALOAD;
```

Return values

Error Code	Description
25066	Bookmark already exists
13047	No privilege for operation

DROP MASTER

DROP MASTER *master_name*

Usage

This command drops the master database definitions from a replica database. After this operation, the replica cannot synchronize with the master database.

▶ **Note**

1. Unregistering the replica is the preferred way to quit using a master database. The DROP MASTER command is used only when the MESSAGE APPEND UNREGISTER REPLICA command is unable to be executed. For details on unregistering a replica, see *"MESSAGE APPEND"* on page 13-23.
2. SOLID *SynchroNet* requires that autocommit be set OFF when using the DROP MASTER command.
3. If *master_name* is a reserved word, it must be enclosed in double quotes.

Usage in Master

This command cannot be used in master.

Usage in Replica

This command is used in replica to drop a master from replica.

Example

```
DROP MASTER "MASTER";
```

Return values

Error code	Description
13047	No privilege for operation
25007	Master *master_name* not found
25019	Database is not a replica database
25056	Autocommit not allowed
25065	Unfinished message *message_name* found for master *master_name*

DROP PUBLICATION

DROP PUBLICATION *publication_name*

Usage

This command drops a publication definition in the master database. All subscriptions to the dropped publication are automatically dropped as well.

Usage in Master

Dropping a publication from master will remove it and replicas will not be able to subscribe to it.

Usage in Replica

Using this command in a replica will drop the publication definition from the replica. It is not necessary to define publications in replica databases.

Example

```
DROP PUBLICATION customers_by_area;
```

Return values

Error Code	Description
25010	Publication *publication_name* not found
13111	Ambiguous entity name *name*.

DROP PUBLICATION REGISTRATION

DROP PUBLICATION *publication_name* REGISTRATION FROM MASTER *master_name*

Usage

This command drops a registration for a publication in the replica database. The publication definition remains on the master database, but a user will be unable to subscribe to the publication. All subscriptions to the registered publication are automatically dropped as well.

You must specify the *master_name* since a replica database can be registered with more than one master with publications that have identical names.

Usage in Master

This command is not used in a master database.

Usage in Replica

Using this command in a replica will drop the registration for the publication in the replica. All subscriptions and their data to this publication are dropped automatically.

Example

```
DROP PUBLICATION customers_by_area REGISTRATION FROM MASTER MASTER1;
```

Return values

Error Code	**Description**
13047	No privilege for operation
25019	Database is not a replica database
25025	Node name not defined
25071	Not registered to publication *publication_name*

DROP REPLICA

DROP REPLICA *replica_name*

Usage

This command drops a replica database from the master database. After this operation, the dropped replica cannot synchronize with the master database.

Note

1. Unregistering the replica is the preferred way to quit using a replica database. The DROP REPLICA command is used only when the MESSAGE APPEND UNREGISTER REPLICA command is unable to be executed. For details on unregistering a replica, see *“MESSAGE APPEND”* on page 13-23.
2. SOLID *SynchroNet* requires that autocommit be set OFF when using the DROP REPLICA command.
3. If *replica_name* is a reserved word, it should be enclosed in double quotes.

Usage in Master

Use this command in the master to drop a replica from master.

Usage in Replica

This command cannot be used in replica.

Example

```
DROP REPLICA salesman_smith ;
DROP REPLICA "REPLICA";
```

Return values

Error code	Description
13047	No privilege for operation
25009	Replica *replica_name* not found
25020	Database is not a master database
25056	Autocommit not allowed
25064	Unfinished message *message_name* found for replica *replica_name*

DROP SUBSCRIPTION

In replica:

DROP SUBSCRIPTION *publication_name* [(*parameter_list*) **|** ALL]
    [COMMITBLOCK *number_of_rows*] [OPTIMISTIC **|** PESSIMISTIC]

In master:

DROP SUBSCRIPTION *publication_name* [(*parameter_list*) **|** ALL]
    REPLICA *replica_name*

Usage

Data that is no longer needed in a replica database can be deleted from the replica database by dropping the subscription that was used to retrieve the data from the master database.

▶ **Note**

SOLID *SynchroNet* requires that autocommit be set OFF when dropping subscriptions.

By default, the data of a subscription is deleted in one transaction. If the amount of data is large, for example, tens of thousands of rows, it is recommended that the COMMITBLOCK be defined. When using the COMMITBLOCK option the data is deleted in more than one transaction. This ensures good performance for the operation.

In a replica, you can define the DROP SUBSCRIPTION command to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the import has completed. Otherwise if the optimistic mode is used, the DROP SUBSCRIPTION may fail due to a concurrency conflict.

A subscription can be dropped also from the master database. In this case, the replica name is included in the command. Names of all replicates that have been registered in the master database can be found in the SYS_SYNC_REPLICAS table. This operation deletes only the internal information about the subscription for this replica.

Dropping a subscription from the master is useful when a replica is no longer using the subscription and the replica has not dropped the subscription itself. Dropping old subscriptions releases old history data from the database. This history data is automatically deleted from the master databases after dropping the subscription.

If a replica subscribes to a publication that has been dropped in the master database, the replica will receive the full data in the next subscription.

Also if the parameter `NeedPublicationRegistration` is set to **No** in the `solid.ini` file, this means that all publications are automatically registered for subscription.

If a subscription is dropped in this case, DROP SUBSCRIPTION also drops the publication registration if the last subscription to the publication was dropped. Otherwise, registration must be dropped explicitly using the DROP PUBLICATION REGISTRATION command or MESSAGE APPEND UNREGISTER PUBLICATION.

You can define the DROP SUBSCRIPTION command to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the import has completed. Otherwise if the optimistic mode is used, the DROP SUBSCRIPTION may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the TableLockWaitTimeout parameter setting in the General section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. See *Appendix C, "Configuration Parameters"* for details on this parameter.

Usage in Master

Use this command to drop a subscription for a specified replica.

Usage in Replica

Use this command to drop a subscription from this replica.

Example

Drop subscription from a master database

```
DROP SUBSCRIPTION customers_by_area('south')
      FROM REPLICA salesman_joe
```

Return Values

Error Code	Description
13047	No privileges for operation
25004	Dynamic parameters are not supported
25009	Replica *replica_name* not found
25010	Publication *publication_name* not found
25019	Database is not a replica database
25020	Database is not a master database
25041	Subscription to publication *publication_name* not found
25056	Autocommit not allowed

DROP SYNC BOOKMARK

DROP SYNC BOOKMARK *bookmark_name*

Usage

This command drops a bookmark defined on a master database. To drop a bookmark, you must have Database Administrator (DBA) or *SynchroNet* Administrator (SYS_SYNC_ADMIN_ROLE) privileges. Bookmarks are typically used when exporting data to a file. After a file is successfully imported to a replica from the master database, it is recommended that you drop the bookmark that you used to export the data to a file.

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 subscriptions.

If you do not drop bookmarks, the history information takes up disk space and unwanted disk IO is incurred, as well, for each bookmark registered in the master database. This may result in unwanted performance degradation.

Note that bookmarks should only be dropped after the exported data is imported into *all* intended replicas. Be sure to drop a bookmark only when you no longer have replicas to import and those replicas have re-subscribed once to the publication after the import.

When dropping bookmarks, *SynchroNet* uses the following rules to delete history records:

- Finds the oldest subscription delivered to any replica on that table
- Finds the oldest bookmark

- Determines which is older, the oldest subscription or oldest bookmark
- Deletes all rows from history up to what it determines is older, the oldest subscription or oldest bookmark.

Usage in Master

Use the DROP SYNC BOOKMARK command to drop a bookmark in a master database.

Usage in Replica

The DROP SYNC BOOKMARK command cannot be used in a replica database.

Example

```
DROP SYNC BOOKMARK new_database;
DROP SYNC BOOKMARK database_after_dataload;
```

Return values

Error Code	Description
25067	Synchronizer bookmark *bookmark_name* not found
13047	No privilege for operation

EXPORT SUBSCRIPTION

EXPORT SUBSCRIPTION *publication_name* [(*publication_parameters*)]
    TO '*filename*'
    USING BOOKMARK *bookmark_name*;
    [WITH [NO] DATA];

Usage

This EXPORT SUBSCRIPTION command allows you export a version of the data from a master database to a file. You can then use the IMPORT command to import the data in the file into a replica database.

There are several uses for the EXPORT SUBSCRIPTION command. Among them are:

- Creating a large replica database (greater than 2GB) 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 *"Creating a Replica By*

Exporting a Subscription with Data" on page 3-32 or *"Creating a Replica By Exporting a Subscription Without Data"* on page 3-35.

- Exporting specific versions of the data to a replica.

  For performance reasons, you may choose to "export" the data rather then to use the MESSAGE APPEND SUBSCRIBE to send the data to a replica.

- Export metadata information only without the actual row data.

  You may want to create a replica that already contains existing data and only needs the schema and version information associated with a publication.

Unlike the MESSAGE APPEND SUBSCRIBE command where a subscription is requested from a replica, you request an export directly from the master database. The export output is saved to a user-specified file rather than a *SynchroNet* message (in the case of subscription output).

Keywords and Clauses

The *publication_name* and *bookmark_name* are identifiers that must exist in the database. For details on creating a publication, read *"CREATE PUBLICATION"* on page 13-5. For details on creating bookmarks, see *"CREATE SYNC BOOKMARK"* on page 13-8. The filename represents a literal value enclosed in single quotes. You can export several publications to a single file by specifying the same file name.

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

The EXPORT SUBSCRIPTION command is based on a given bookmark, which means that export data is consistent up to this bookmark. When you export data, the EXPORT SUBSCRIPTION command includes all rows as in a full publication up to the bookmark. However, since export is based on a given bookmark, the subsequent subscriptions are incremental.

If a bookmark is created in a master for the purpose of exporting and importing data, then the bookmark must exist when:

- The EXPORT SUBSCRIPTION command is executed on the master database.

  If the bookmark does not exist at this point, error message 25057 is generated, indicating the bookmark cannot be found.

- The IMPORT command is executed on all intended replicas and replicas receive their first subscriptions.

  During a file import, a connection to a master database is not needed and the existence of the bookmark is not checked. However, if the bookmark does not exist at the time a replica receives its first subscription, the subscription fails with error message 25067 and the import data is unusable. The remedy is to create a new bookmark on the master, re-export the data, and re-import the data.

An export file can contain more than one publication. You can export subscriptions using either the WITH DATA and WITH NO DATA options:

- Use the WITH DATA option to create a replica when data is exported to an existing database that does not contain master data and requires a subset of data. For details, read *“Creating a Replica By Exporting a Subscription with Data”* on page 3-32.
- Use the WITH NO DATA option to create a replica when a subscription is imported to a database that already contains the data (for example, using a backup copy of an existing master). For details, read *“Creating a Replica By Exporting a Subscription Without Data”* on page 3-35.

By default, the export file is created using the WITH DATA option and includes all rows. If there is more than one publication specified, the exported file can have a combination of "WITH DATA" and "WITH NO DATA."

Usage Rules

Note the following rules when using the EXPORT SUBSCRIPTION command:

- Only one file per subscription is allowed for export. You can use the same file name to include multiple subscriptions on the same file.
- The file size of an export file is dependent upon the underlying operating system. If a respective platform (such as SUN, or HP) allows more than 2GB, you can write files greater than 2GB. This means that replica (recipient) should also have a compatible platform and file system. Otherwise, the replica is not able to accept the export file. If both the operating system on a master and replica support file sizes greater than 2GB, then export files greater than 2GB are permitted.
- An export file can contain more than one subscription. Subscriptions can be exported using either the "WITH DATA" or WITH NO DATA options. An exported file with multiple subscriptions can have a combination of WITH DATA and WITH NO DATA included.
- When a subscription is exported to a file using the WITH NO DATA option, only meta-data (that is, schema and version information corresponding to that publication) is exported to the file.
- SOLID *SynchroNet* requires that autocommit be set OFF when using the EXPORT SUBSCRIPTION command.

Usage in Master

Use this command to request master data for export to a file.

Usage in Replica

This command is not available in a replica database.

Example

```
EXPORT SUBSCRIPTION FINANCE_PUBLICATION TO 'FINANCE.EXP' USING BOOKMARK
BOOKMARK_FOR_FINANCE_DB WITH NO DATA;
```

Return values

Error Code	Description
25056	Autocommit not allowed
25057	Bookmark is not found.
25068	Export file *file_name* open failure.
25010	Publication *name* not found.

GET_PARAM ()

get_param ('*param_name*')

Usage

The get_param() function retrieves a parameter that was placed on the bulletin board using the put_param() function or with commands SAVE PROPERTY, SAVE DEFAULT PROPERTY, and SET SYNC PARAMETER. The parameter that is retrieved is specific to a catalog and each catalog has a different set of parameters. This function returns the VARCHAR value of the parameter or NULL, if the parameter does not exist in the bulletin board.

Because get_param() is a SQL function, it can be used only in a procedure or as part of a SELECT statement.

The parameter name must be enclosed in single quotes.

Usage in Master

Use the get_param() function in the master for retrieving parameter values.

Usage in Replica

Use the get_param() function in replicas for retrieving parameter values.

Example

```
SELECT put_param('myparam', '123abc') ;
SELECT get_param('myparam');
```

Return values

Error Code	Description
13086	Invalid data type in a parameter.

When executed successfully get_param() returns the value of the assigned parameter.

GRANT SUBSCRIBE

GRANT SUBSCRIBE ON *publication_name* TO {PUBLIC **|** *user_name*,

[*user_name*] ... **|** *role_name*, [*role_name*] ... }

Usage

This command grants access rights on a publication to a user or role defined in the master database.

Usage in Master

Use this command to grant user or role access rights to a publication.

Usage in Replica

This command is not available in a replica database.

Example

```
GRANT SUBSCRIBE ON customers_by_area TO salesman_jones;
GRANT SUBSCRIBE ON customers_by_area TO all_salesmen;
```

Return values

Error Code	Description
13137	Illegal grant/revoke mode
13048	No grant option privilege
25010	Publication *name* not found

IMPORT

IMPORT ’*filename*’ [COMMITBLOCK *number_of_rows*]
    [OPTIMISTIC | PESSIMISTIC]

Usage

This IMPORT command allows you import data to a replica from a data file created by the EXPORT SUBSCRIPTION command.

The *filename* represents a literal value enclosed in single quotes. The import command can accept a single filename only. Therefore, all the data to be imported to a replica must be contained in one file.

The COMMITBLOCK option indicates the number of rows processed before the data is committed. The *#rows* is an integer value used with the optional COMMITBLOCK clause to indicate the commitblock size. The use of the COMMITBLOCK option improves the performance of the import and releases the internal transaction resources frequently.

The optimal value for the COMMITBLOCK size varies depending on various resources at the server. A good example a COMMITBLOCK size of 1000 for 10,000 rows. If you do not specify the COMMITBLOCK option, the IMPORT command uses all rows in the publication as one transaction. This may work well for a small number of rows, but is problematic for 1000s and millions of rows.

You can define the import to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the import has completed. Otherwise if the optimistic mode is used, the IMPORT may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the TableLockWaitTimeout parameter setting in the General section of the `solid.ini` configuration file determines the transaction’s wait period until the exclusive or shared lock is released. See *Appendix C, “Configuration Parameters”* for details on this parameter.

Imported data is not valid in a replica until it is re-subscribed once after the import. At the time a replica makes its first subscription, the bookmark used to export the file must exist in the master database. If it does not exist, then the subscription fails. This means, you are required to create a new bookmark on the master database, re-export the data, and re-import the data on the replica.

Usage Rules

Note the following rules when using the IMPORT command:

- Only one file per subscription is allowed for import.
- The file size of an export file is dependent upon the underlying operating system. If a respective platform (such as SUN, or HP) allows more than 2GB, you can write files greater than 2GB. This means that a replica (recipient) should also have a compatible platform and file system. Otherwise, the replica is not able to accept the export file. If both the operating system on a master and replica support file sizes greater than 2GB, then export files greater than 2GB are permitted.
- Backup replica databases before using the IMPORT command. If a COMMITBLOCK option is used and fails, then the imported data is only partially committed; you will need to restore the replica with a backup file.
- SOLID *SynchroNet* requires that autocommit be set OFF when using the DROP REPLICA command.

Usage in Master

This command is not available in a master database.

Usage in Replica

Use this command in a replica to import data from a data file created by the EXPORT SUBSCRIPTION command in a master database.

Example

```
IMPORT 'FINANCE.EXP';
```

Return values

Error Code	Description
25007	Master *master_name* not found.
25019	Database is not a replica database.
25069	Import file *file_name* open failure.

Error Code	Description
13XXX	Table level error
13124	User id *num* not found
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
21XXX	Communication error
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25036	Publication *publication_name* not found or publication version mis-match
25040	User id *user_id* is not found
25041	Subscription to publication *publication_name* not found
25048	Publication *publication_name* request info not found
25054	Table *table_name* is not set for synchronization history
25056	Autocommit not allowed
25060	Column *column_name* does not exist on publication *publication_name* resultset in table *table_name*

MESSAGE APPEND

MESSAGE *unique_message_name* APPEND

[PROPAGATE TRANSACTIONS [WHERE *property_name* (=|<|<=|>|>=|<>)

'*value_string*' | *ALL*]]

[SUBSCRIBE *publication_name*[(*publication_parameters*)

[FULL]]

[REGISTER PUBLICATION *publication_name*]

[UNREGISTER PUBLICATION *publication_name*]

[REGISTER REPLICA]

[UNREGISTER REPLICA]

[SYNC_CONFIG ('*sync_config_arg*')]

Usage

Once a message has been created in the replica database with the MESSAGE BEGIN command, you can append the following tasks to that message:

- Propagate transactions to the master database
- Subscribe a publication from the master database
- Register or unregister a publication for replica subscription
- Register or unregister a replica database to the master
- Download master user information (list of user names and passwords) from the master database

The PROPAGATE TRANSACTIONS task may contain a WHERE clause that is used to propagate only those transactions where a transaction property defined with the SAVE PROPERTY statement meets specific criteria. Usage of keyword ALL overrides any default propagation condition set earlier with the statement SAVE DEFAULT PROPAGATE PROPERTY WHERE *property_name* (=|<|<=|>|>=|<>) '*value*'. This enables you to propagate transactions that do not contain any properties.

The REGISTER REPLICA task adds a new replica database to the list of replicas in the master database. Replicas must be registered with the master database before any other synchronization functions can be performed in the replica database.

Synchronizing each master database to the replica in a multi-master environment requires registration of a replica to each master database by setting up catalogs. One replica catalog can register only to one master catalog. This command performs the actual registration once catalogs are created in a synchronization environment. For synchronization to the replica, a new catalog for each master database is required. Read *"Guidelines for multi-master topology"* on page 11-6 for details on catalogs.

If a *catalog_name* is not specified, the default name **NULL** is used.

Note

A single-master environment does not require the use of catalogs. By default, when catalogs are not used, registration of the replica occurs automatically with a base catalog that maps to a master base catalog, whose name is given when the database is created. Therefore, no backward compatibility issues exist for versions prior to *SynchroNet* 2.0, which supported only the single-master architecture.

The UNREGISTER REPLICA option removes an existing replica database from the list of replicas in the master database.

The SUBSCRIBE task may contain parameters to the publication (if used in the publication). Using keyword FULL with SUBSCRIBE forces fetching full subscription data to the replica. The publication requested must be registered.

The REGISTER PUBLICATION task registers a publication in the replica so that it can be subscribed to. Users can only subscribe to publications that are registered. In this way, publication parameters are validated, preventing users from accidently subscribing to unwanted subscriptions or requesting ad hoc subscriptions. All tables that the registered publication refers to must exist in the replica.

The UNREGISTER PUBLICATION option removes an existing registered publication from the list of registered publications in the master database.

▶ **Note**

You can set the `NeedPublicationRegistration` parameter in the solid.ini file to "No" which disables exploit registration. This means that publications are automatically registered for user subscriptions when they are created. Disabling registration is used for backward compatibility with SynchroNet version prior to 2.0.

The input argument of the SYNC_CONFIG task defines the search pattern of the user names that are returned from the master database to the replica. a SQL wildcard '%' can be used in this argument.

Usage in Master

The MESSAGE APPEND command cannot be used in a master database.

Usage in Replica

Use MESSAGE APPEND in replicas to append tasks to a message that has been created with MESSAGE BEGIN.

Example

```
MESSAGE MyMsg0001 APPEND PROPAGATE TRANSACTIONS;
MESSAGE MyMsg0001 APPEND SUBSCRIBE PUB_CUSTOMERS_BY_AREA('SOUTH');
MESSAGE MyMsg0001 APPEND REGISTER REPLICA;
MESSAGE MyMsg0001 APPEND SYNC_CONFIG 'S%'
MESSAGE MyMsg0001 APPEND REGISTER PUBLICATION publ_customer;
```

Return values

Error Code	Description
13133	Not a valid license for this product
25004	Dynamic parameters are not supported
25005	Message *message_name* is already active
25006	Message *message_name* not active
25015	Syntax error: *error_message*, line *line_number*
25018	Illegal message state
25024	Master not defined
25025	Node name not defined
25026	Not a valid master user
25028	Message *message_name* can include only one system subscription
25035	Message *message_name* is in use
25044	SYNC_CONFIG system publication takes only character arguments
25056	Autocommit not allowed
25071	Not registered to publication *publication_name*
25072	Already registered to publication *publication_name*

MESSAGE BEGIN

MESSAGE *unique_message_name* BEGIN [TO *master_node_name*]

Usage

Each message that is sent from a replica to the master database must explicitly begin with the MESSAGE BEGIN command. Each message must have a name that is unique within the server.

When registering a replica to a master catalog, other than the master system catalog, you must provide the master node in the MESSAGE BEGIN command. The master node name is used to resolve the correct catalog at the master database. Note that specifying a master node name only applies when using the REGISTER REPLICA command.

Note

When working with messages, be sure the autocommit mode is always switched off.

Usage in Master

The MESSAGE BEGIN command cannot be used in a master database.

Usage in Replica

Use MESSAGE BEGIN to define new messages in replicas

Example

```
MESSAGE MyMsg0001 BEGIN ;
```

Return values from Replica

Error Code	Description
25005	Message *message_name* is already active.
25035	Message *message_name* is in use.
25056	Autocommit not allowed

Return values from Master

Error Code	Description
25019	Database is not a replica database.
25025	Node name not defined.
25056	Autocommit not allowed

MESSAGE DELETE

MESSAGE *message_name* [FROM REPLICA *replica_name*] DELETE

Usage

If the execution of a message is terminated because of an error, this command lets you explicitly delete the message from the database to recover from the error. Note that all transactions that were propagated to the master in this message are permanently lost when the message is deleted. To use this command, you must have SYS_SYNC_ADMIN_ROLE access.

▶ **Note**

As an alternative, the MESSAGE DELETE CURRENT TRANSACTION command provides better recovery since it lets you delete only the offending transaction.

If the message needs to be deleted from the master database, then the name of the replica database that forwarded the message needs to also be provided.

When deleting messages, be sure the autocommit mode is always switched off.

Usage in Master

Use this command in the master to delete a failed message. Be sure to specify the replica in the syntax: 'FROM REPLICA *replica_name*'.

Usage in Replica

This command is used in the replica to delete a message.

Example

```
MESSAGE MyMsg0000 DELETE ;
MESSAGE MyMsg0001 FROM REPLICA bills_laptop DELETE ;
```

Return values from replica

Error code	Description
25005	Message *message_name* is already active
25013	Message *message_name* not found
25035	Message *message_name* is in use
25056	Autocommit not allowed

Return values from master

Error code	Description
13047	No privilege for operation
25009	Replica *replica_name* not found
25013	Message *message_name* not found
25020	Database is not a master database
25035	Message *message_name* is in use
25056	Autocommit not allowed

Result set master

Column Name	Description
Partno	Message part number
Type	Equal to 6, for 'message delete'.
Masterid	Master ID
Msgid	Message ID
Errcode	Message error code. Zero if successful.
Errstr	Message error string. NULL is successful.
Insertcount	Number of inserted rows.
Deletecount	Number of rows deleted from replica.
Bytecount	Size of message in bytes.

Column Name	Description
Info	NULL

MESSAGE DELETE CURRENT TRANSACTION

MESSAGE *message_name* FROM REPLICA *replica_name*

DELETE CURRENT TRANSACTION

Usage

This command deletes the current transaction from a given message. To use this command requires SYS_SYNC_ADMIN_ROLE privilege.

The execution of a message stops, if a DBMS level error such as a duplicate insert occurs during the execution. This kind of error can be resolved by deleting the offending transaction from the message. Once deleted with the MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION, an administrator can proceed with the synchronization process.

When deleting the current transaction, be sure the autocommit mode is always switched off.

This command is used only when the message is in an error state; if used otherwise, an error message is returned. This command is a transactional operation and must be committed before message execution may continue. To restart the message after the deletion is committed, used the following statement:

MESSAGE *msgname* FROM REPLICA *replicaname* EXECUTE

Note

Delete a transaction only as a last resort; normally transactions should be written to prevent unresolved conflicts in a master database. MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION is intended for use in the development phase, when unresolved conflicts occur more frequently.

Use caution when deleting a transaction. Because subsequent transactions may be dependent on the results of a deleted transaction, the risk incurred is more transaction errors.

Usage in Master

Use this command in the master to delete a failed transaction.

Usage in Replica

This command is not available in the replica.

Example

```
MESSAGE somefailures FROM REPLICA laptop1 DELETE
CURRENT TRANSACTION;
COMMIT WORK;
MESSAGE somefailures from REPLICA laptop1 EXECUTE;
```

Return values

Error code	Description
13047	No privilege for operation
25009	Replica *replica_name* not found
25013	Message name *message_name* not found
25018	Illegal message state
25056	Autocommit not allowed

MESSAGE END

MESSAGE *unique_message_name* END

Usage

A message must be persistent before it can be sent to the master database. Ending the message with the MESSAGE END command (which is optional) saves the message, which makes it persistent. Another way is committing the transaction, which ends any currently open message.

▶ **Note**

When working with messages, be sure the autocommit mode is always switched off.

Usage in Master

The MESSAGE END command cannot be used in a master database.

Usage in Replica

Use the MESSAGE END command in replicas to end a message.

Example

```
MESSAGE MyMsg001 END ;
COMMIT WORK ;
```

The following example shows a complete message that propagates transactions and subscribes to publication PUB_CUSTOMERS_BY_AREA.

```
MESSAGE MyMsg0001 BEGIN ;
MESSAGE MyMsg0001 APPEND PROPAGATE TRANSACTIONS;
MESSAGE MyMsg0001 APPEND SUBSCRIBE PUB_CUSTOMERS_BY_AREA('SOUTH');
MESSAGE MyMsg001 END ;
COMMIT WORK ;
```

Return values from Replica

Error Code	Description
13133	Not a valid license for this product
25005	Message *message_name* is already active
25013	Message *message_name* not found
25018	Illegal message state
25026	Not a valid master user
25035	Message *message_name* is in use
25056	Autocommit not allowed

Return values from Master

Error Code	Description
25019	Database is not a replica database
25056	Autocommit not allowed

MESSAGE EXECUTE

MESSAGE *message_name* EXECUTE [OPTIMISTIC | PESSIMISTIC]

Usage

This command allows a message to be re-executed if the execution of a reply message fails in a replica. This can occur, for example, if the database server detects a concurrency conflict between a subscription and an ongoing user transaction.

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 the PESSIMISTIC option for table-level locking; this ensures the message execution is successful.

In this mode, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise if the optimistic mode is used, the MESSAGE EXECUTE command may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the TableLockWaitTimeout parameter setting in the General section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. See *Appendix C, "Configuration Parameters"* for details on this parameter.

▶ **Note**

When working with messages, be sure the autocommit mode is always switched off.

Usage in Master

This command is not available in the master.

Usage in Replica

Use this command in the replica to re-execute a failed message execution in the replica.

Example

```
MESSAGE MyMsg0002 EXECUTE;
```

Return values

Error code	**Description**
13XXX	Table level error
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
25005	Message *message-name* is already active
25013	Message name *message_name* not found
25018	Illegal message state
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	Message *message_name* is in use
25040	User id *user-id* is not found
25041	Subscription to publication *publication_name* not found
25048	Publication *publication_name* request info not found
25056	Autocommit not allowed

Result set

MESSAGE EXECUTE returns a result set. The returned result set is the same as with command MESSAGE GET REPLY.

MESSAGE FORWARD

MESSAGE *unique_message_name* FORWARD

[TO *master_listen_name*]

[TIMEOUT FOREVER | *seconds*]

[COMMITBLOCK *block_size_in_rows*]

[OPTIMISTIC | PESSIMISTIC]

Usage

After a message has been completed and made persistent with the MESSAGE END command, it can be sent to the master database using the MESSAGE FORWARD command.

It is only necessary to specify the recipient of the message with keyword TOwhen a new replica is being registered with the master database; that is, when the first message from a replica to the master server is sent.

Each sent message has a reply message. The TIMEOUT property defines how long the replica server will wait for the reply message.

If a TIMEOUT is not defined, the message is forwarded to the master and the replica does not fetch the reply. In this case the reply can be retrieved with a separate MESSAGE GET REPLY call.

If the reply of the sent message contains subscriptions of large publications, the size of the subscription's commit block, that is, the number of rows that are committed in one transaction, can be defined using the COMMITBLOCK property. This has a positive impact on the performance of the replica database. It is recommended that there are no on-line users accessing the database when the COMMITBLOCK property is being used.

As part of the MESSAGE FORWARD operation, you can specify table-level pessimistic locking when the reply message is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise if the optimistic mode is used, the MESSAGE FORWARD operation may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the TableLockWaitTimeout parameter setting in the General section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. See *Appendix C, "Configuration Parameters"* for details on this parameter.

If a forwarded message fails in delivery due to a communication error, you must explicitly use the MESSAGE FORWARD to resend the message. Once resent, MESSAGE FORWARD re-executes the message.

Note

When working with the messages, be sure the autocommit mode is always switched off.

Example

Forward message, wait for the reply for 60 seconds

```
MESSAGE MyMsg001 FORWARD TIMEOUT 60 ;
```

Forward message to a master server that runs on the "mastermachine.acme.com" machine. Do not wait for the reply message.

```
MESSAGE MyRegistrationMsg FORWARD TO "tcp mastermachine.acme.com 1313"
```

Forward message, wait for the reply for 5 minutes and commit the data of the subscribed publications to replica database in transactions of max. 1000 rows.

```
MESSAGE MyMsg001 FORWARD TIMEOUT 300 COMMITBLOCK 1000 ;
```

Return values from Replica

Error Code	Description
13XXX	Table level error
21XXX	Communication error
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
25005	Message *message_name* is already active
25013	Message name *message_name* not found
25018	Illegal message state
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	Message *message_name* is in use
25040	User id *user-id* is not found
25041	Subscription to publication *publication_name* not found
25048	Publication *publication_name* request info not found
25052	Node *node_name* already exists
25054	Table *table_name* is not set for synchronization history
25055	Connect information is allowed only when not registered
25056	Autocommit not allowed

Error Code	Description
25057	The replica database has already been registered to a master database
25060	Column *column_name* does not exist on publication *publication_name* resultset in table *table_name*

Return values from Master

Error code	Description
13XXX	Table level error
13124	User id *num* not found
25016	Message not found, replica ID *replica_id*, message ID *message_id*
25056	Autocommit not allowed
Bytecount	The size of this message in bytes.
Info	NULL

If the MESSAGE FORWARD also retrieves the reply, the command returns a result set. The result set returned is the same as the one returned with the command MESSAGE GET REPLY.

MESSAGE FROM REPLICA EXECUTE

MESSAGE *message_name* FROM REPLICA *replica_name* EXECUTE

Usage

The execution of a message stops, if a DBMS level error such as a duplicate insert occurs during the execution or an error is raised from a procedure by putting the SYS_ROLLBACK parameter to the transactions bulletin board. This kind of error is recoverable by fixing the reason for the error, for example, by removing the duplicate row from the database, and then executing the message.

Note

When working with the messages, be sure the autocommit mode is always switched off.

Usage in Master

Use this command in the master to execute a failed message.

Usage in Replica

This command is not available in the replica.

Example

```
MESSAGE MyMsg0002 FROM REPLICA bills_laptop EXECUTE;
```

Return values

Error code	Description
13047	No privilege for operation
25009	Replica *replica_name* not found
25013	Message name *message_name* not found
25018	Illegal message state
25056	Autocommit not allowed

MESSAGE GET REPLY

MESSAGE *unique_message_name* GET REPLY

[TIMEOUT FOREVER **|** *seconds*]

[COMMITBLOCK *block_size_in_rows*]

[NO EXECUTE]

[OPTIMISTIC **|** PESSIMISTIC]

Usage

If the reply to a sent message has not been received by the MESSAGE FORWARD command, it can be requested separately from the master database by using the MESSAGE GET REPLY command in the replica database.

If the reply message contains subscriptions of large publications, the size of the subscription's commit block, that is, the number of rows that are committed in one transaction, can be limited using the COMMITBLOCK property. This has a positive impact on the performance of the replica database. It is recommended that there are no on-line users in the database when the COMMITBLOCK property is in use.

If the execution of a reply message with the COMMITBLOCK property fails in the replica database, it cannot be re-executed. The failed message must be deleted from the replica database and re-subscribed from the master database.

If NO EXECUTE is specified, when the reply message is available at the master, it is only read and stored for later execution. Otherwise, the reply message is downloaded from the master and executed in the same statement. Using NO EXECUTE reduces bottlenecks in communication lines by allowing reply messages for later execution in different transactions.

You can define the reply message to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise if the optimistic mode is used, the MESSAGE GET REPLY operation may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the TableLockWaitTimeout parameter setting in the General section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. See *Appendix C, "Configuration Parameters"* for details on this parameter.

If a reply message fails in delivery due to a communication error (WITHOUT COMMITBLOCK), you must explicitly use the MESSAGE GET REPLY to resend the message. Once resent, MESSAGE GET REPLY re-executes the message.

Note

When working with the messages, be sure the autocommit mode is always switched off.

Usage in Master

MESSAGE GET REPLY cannot be used in the master.

Usage in Replica

Use MESSAGE GET REPLY in the replica to fetch a reply of a message from the master.

Example

```
MESSAGE MyMessage001 GET REPLY TIMEOUT 120
MESSAGE MyMessage001 GET REPLY TIMEOUT 300 COMMITBLOCK 1000
```

Return values from Replica

Fatal errors in transaction propagation abort the message and return an error code to the replica. To propagate the aborted message you need to correct the fatal errors and restart the message with command MESSAGE EXECUTE FROM REPLICA.

If a subscription fails in the master, an error message about the failed subscription is added to the result set. Other parts of the message are executed normally. The failed publication must be re-subscribed from the master in a separate synchronization message.

If a subscription (that is, the execution of the reply message) fails in the replica, the message is still available in the replica database and can be restarted with the MESSAGE *msg_name* EXECUTE command.

Error code	Description
13XXX	Table level error
13124	User id *num* not found
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
21XXX	Communication error
25005	Message *message_name* is already active
25013	Message name *message_name* not found
25018	Illegal message state
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	Message *message_name* is in use
25036	Publication *publication_name* not found or publication version mismatch
25040	User id *user_id* is not found
25041	Subscription to publication *publication_name* not found
25048	Publication *publication_name* request info not found
25054	Table *table_name* is not set for synchronization history
25056	Autocommit not allowed

Error code	Description
25057	Already registered to master *master_name*
25060	Column *column_name* does not exist on publication *publication_name* resultset in table *table_name*

Return values from Master

Error code	Description
13XXX	Table level error
13124	User id *num* not found
25012	Message reply timed out
25016	Message not found, replica id *replica-id*, message id *message-id*
25043	Reply message is too long (*size_of_messages* bytes). Maximum is set to *max_message_size* bytes.
25056	Autocommit not allowed

Result set

MESSAGE GET REPLY returns a result set table. The columns of the result set are as follows:

Column Name	Description
Partno	Message part number
Type	The type of result set row. Possible different types are: 0: Message part start 3: Task 4: Subscription task 5: Type of subscription FULL or INCREMENTAL
Masterid	Master ID
Msgid	Message ID
Errcode	Message error code. Zero if successful.
Errstr	Message error string. NULL is successful.

Column Name	Description
Insertcount	Number of inserted rows to replica. Type=3: Total number of insert Type=4: Row inserts restored from replica history to replica base table Type=5: Insert operations received from master
Deletecount	Number of rows deleted from replica. Type = 3: Total number of deletes Type = 4: Row deletes restored from replica base table Type = 5: Delete operations received from master
Bytecount	Size of message in bytes. Indicated in result received from command MESSAGE END. Otherwise 0.

Info	Information of the current task.
	Type = 0: then Message name
	Type = 3: Publication name
	Type = 4: Table name
	Type = 5: FULL/INCREMENTAL

Errcode	Message error code. Zero if successful.

PUT_PARAM ()

put_param (*param_name*,*param_value*)

Usage

With SOLID *Intelligent Transaction,* SQL statements or procedures of a transaction can communicate with each other by passing parameters to each other using a parameter bulletin board. The bulletin board is a storage of parameters that is visible to all statements of a transaction.

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.

Use the put_param() function to place a parameter on the bulletin board. If the parameter already exists, the new value overwrites the previous one.

These parameters are not propagated to the master.

Because put_param() is a SQL function, it can be used only within a procedure or in a SQL statement.

Both the parameter name and value are of type VARCHAR.

Usage in Master

Put_param() function can be used in the master for setting parameters to the parameter bulletin board of the current transaction.

Usage in Replica

Put_param() function can be used in replicas for setting parameters to the parameter bulletin board of the current transaction.

Example

```
Select put_param('myparam', '123abc') ;
```

Return values

Error Code	Description
13086	Invalid data type in a parameter.

When executed successfully put_param() returns the new value of the assigned parameter.

REVOKE SUBSCRIBE

REVOKE SUBSCRIBE ON *publication_name* FROM {PUBLIC **|** *user_name*,

[*user_name*] ... **|** *role_name*, [*role_name*] ... }

Usage

This command revokes access rights to a publication from a user or role defined in the master database.

Usage in Master

Use this command to revoke access rights to a publication from a user or role.

Usage in Replica

This command is not available in a replica database.

Example

```
REVOKE SUBSCRIBE ON customers_by_area FROM joe_smith;
REVOKE SUBSCRIBE ON customers_by_area FROM all_salesmen;
```

Return values

Error Code	Description
13137	Illegal grant/revoke mode
13048	No grant option privilege
25010	Publication *name* not found

SAVE

SAVE *sql_statement*

Usage

The statements of a transaction that need to be propagated to the master database must be explicitly saved to the transaction queue of the replica database. Adding a SAVE command before the statement does this.

Only those users that are defined in the master database are allowed to save statements. The saved statements are executed in the master database using the access rights of the same user that was used in the replica for saving the statement.

Usage in Master

This command cannot be used in the master.

Usage in Replica

Use this command in the replica to save statements for propagation to the master.

Example

```
SAVE INSERT INTO mytbl (col1, col2) VALUES ('calvin', 'hobbes')
SAVE CALL SP_UPDATE_MYTBL('calvin_1', 'hobbes')
SAVE CALL SP_DELETE_MYTBL('calvin')
```

Return values

Error Code	Description
25001	Internal error
25003	Cannot save SAVE statements
25070	Statement can be saved only for one master in transaction.

SAVE PROPERTY

SAVE PROPERTY *property_name* VALUE '*value_string*'

SAVE PROPERTY *property_name* VALUE NONE

SAVE DEFAULT PROPERTY *property_name* VALUE '*value_string*''

SAVE DEFAULT PROPERTY *property_name* VALUE NONE

SAVE DEFAULT PROPAGATE PROPERTY WHERE *name* (=|<|<=|>|>=|<>)'*value*'

SAVE DEFAULT PROPAGATE PROPERTY NONE

Usage

It is possible to assign properties to the current active transaction with the following command:

SAVE PROPERTY *property_name* VALUE '*value_string*'

The statements of the transaction in the master database can use these properties. Properties are only available in the replica database that apply to the command MESSAGE APPEND *unique_message_name* PROPAGATE WHERE *property* > '*value_string*'. When the transaction is executed in the master database, the saved properties are placed on the parameter bulletin board of the transaction. If the saved properties already exist, the new value overwrites the previous one.

It is also possible to define default properties that are saved to all transactions of the current connection. The command for this is:

SAVE DEFAULT PROPERTY *property_name* VALUE '*value_string*'

A SAVE DEFAULT PROPAGATE PROPERTY WHERE command can be used to save default transaction propagation criteria. This can be used for example to set the propagation priority of transactions created in the current connection.

SAVE DEFAULT PROPAGATE PROPERTY WHERE *priority* > '*value*' can be used in a connection level to append all MESSAGE *unique_message_name* APPEND PROPAGATE

TRANSACTIONS statements to have the default WHERE statement. If the WHERE statement is entered also in the PROPAGATE statement, it will override the statement set with the DEFAULT PROPAGATE PROPERTY.

A property or a default property can be removed by re-saving the property with value string NONE.

Usage in Master

This command cannot be used in the master database.

Usage in Replica

You can use these commands in the replica to set properties for a transaction that is saved for propagation to the master. The property's value can be read in the master database.

Example

```
SAVE PROPERTY conflict_rule VALUE 'override'
SAVE DEFAULT PROPERTY userid VALUE 'scott'
SAVE DEFAULT PROPERTY userid VALUE NONE
SAVE DEFAULT PROPAGATE PROPERTY WHERE priority > '2'
```

Return values

Error Code	Description
13086	Invalid data type in a parameter.

Result set

SAVE PROPERTY does not return a result set.

SET SYNC *master_or_replica*

SET SYNC *master_or_replica yes_or_no*

where:

master_or_ replica ::= MASTER | REPLICA

yes_or_ no ::= YES | NO

Usage

When a database is created and configured for synchronization use, you must use this command to specify whether the database is a master, replica, or both. Only those with SYS_SYNC_ADMIN_ROLE can set the database role.

The database is a master database if there are replicas in the domain that subscribe publications from this database and/or propagate transactions to it. Databases that can subscribe to publications from a master database are replicas. In a multi-tier synchronization, intermediate level databases serve a dual role, as both a master and replica databases.

When specifying this command, note that you can use the command once or twice to set the database for a dual role. For example:

```
SET SYNC MASTER YES;
SET SYNC REPLICA YES;
-or-
SET SYNC MASTER YES REPLICA YES;
```

Note that when you set the database for dual roles, SET SYNC REPLICA YES in the first example does not override SET SYNC MASTER YES, nor does REPLICA YES override MASTER YES in the second example. In both examples, only the following explicit statement can override the status of the master database:

```
SET SYNC MASTER NO;
```

Once overridden, the current database is set as replica only.

Examples

```
-- configure as replica
SET SYNC REPLICA YES;
-- configure as master
SET SYNC MASTER YES;
-- configure as replica, not master
SET SYNC REPLICA YES MASTER NO;
-- configure as replica and master
SET SYNC REPLICA YES MASTER YES;
```

Return values

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
13133	Not a valid license for this product
25051	Unfinished messages found

SET SYNC CONNECT

SET SYNC CONNECT '*connect_string*' TO MASTER *master_name*

Usage

This command changes the network name associated with the master database name. Use this command in a replica whenever you have changed network names in databases that a replica connects to. Network names are defined in the `Listen` parameter of the `solid.ini` configuration file in a master or dual role (master and replica) database.

Usage in Master

This command cannot be used in a master.

Usage in Replica

Use this command in a replica to change the master's network name.

Example

```
SET SYNC CONNECT 'tcp server.company.com 1313'
     TO MASTER hq-master ;
```

Return values

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
21300	Illegal network protocol
25007	Master *master_name* not found
25019	Database is not a replica database

SET SYNC NODE

SET SYNC NODE *unique_node_name* **|** NONE

Usage

Each catalog of a SOLID *SynchroNet* 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.

The SET SYNC NODE NONE option removes the node name from the current catalog. This option is used when you are dropping a synchronized database and removing its registration.

▶ **Note**

When using the SET SYNC NODE NONE option, be sure the catalog associated with the node name is not defined as a master, replica, or both. To remove the node name, the catalog must be defined as SET SYNC MASTER NO and/or SET SYNC REPLICA NO. If you do try to set the node name to NONE on a master and/or replica catalog, SOLID *SynchroNet* returns error message 25082.

You can use the SET NODE *unique_node_name* option to rename a node name only if the replica database is not registered and/or there are no replicas registered in the master database,

Usage in Master

Use this command in the master to set or remove the node name from the current catalog.

Usage in Replica

Use this command in the replica to set or remove the node name from the current catalog.

Example

```
SET SYNC NODE SalesmanJones ;
```

Return Values

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
25059	After registration nodename cannot be changed
25082	Node name can not be removed if node is master or replica.

SET SYNC PARAMETER

SET SYNC PARAMETER *parameter_name* '*value_as_string*';

SET SYNC PARAMETER *parameter_name* NONE;

Usage

This command defines persistent catalog-level parameters that are visible via the parameter bulletin board to all transactions that are executed in that catalog. Each catalog has a different set of parameters.

If the parameter already exists, the new value overwrites the previous one. An existing parameter can be deleted by setting its value to NONE. All parameters are stored in the SYS_BULLETIN_BOARD system table.

These parameters are not propagated to the master.

In addition to system specific parameters, you can also store in the system table a number of system parameters that configure the synchronization functionality. Available system parameters are listed at the end of the SQL reference.

Usage in Master

Use the SET SYNC PARAMETER in the master for setting database parameters.

Usage in Replica

Use the SET SYNC PARAMETER in replicas for setting database parameters.

Example

```
SET SYNC PARAMETER db_type 'REPLICA'
SET SYNC PARAMETER db_type NONE
```

Return Values

Error Code	Description
13086	Invalid data type in a parameter

SET SYNC USER

SET SYNC USER *master_username* IDENTIFIED BY *password*

SET SYNC USER NONE

Usage

This command is used to define the username and password for the registration process when the replica database is being registered in the master database. To use this command, you are required to have SYS_SYNC_ADMIN_ROLE access.

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 in a replica database. If you want to designate a different master user for a replica, you must map the replica ID on the replica database with the master ID on the master database. For details, read *“Mapping Replica User ID with Master User ID”* on page 12-11.

You define the registration username in the master database. The name you specify must have sufficient rights to execute the replica registration tasks. 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.

After the registration has been successfully completed, you must reset the sync user to NONE; otherwise, if a master user saves statements, propagates messages, or subscribes or registers to publications, the following error message is returned:

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

Usage in Master

This command is not available in the master database.

Usage in Replica

Use this command in the replica to set the user name.

Example

```
SET SYNC USER homer IDENTIFIED BY marge ;
SET SYNC USER NONE ;
```

SOLID *SynchroNet* System Parameters

SOLID *SynchroNet* system parameters are divided into the following categories:

- **Read only system parameters** that are maintained by SOLID *SynchroNet* and can only be read by using GET_PARAM(*parameter_name*) syntax.

  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.

- **Updateable system parameters** that can be set and updated by the user through PUT_PARAM(*parameter_name*, *value*). Updateable system parameters are used by SOLID *SynchroNet*.

  Like the category above, the life cycle of these parameters is also one transaction.

- **Database level system parameters** that are set using SET SYNC PARAMETER *parameter_name value* syntax.

  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 earlier in this chapter. The parameters within each category are described in the following sections.

Read only system parameters

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 transactions id from replica database. For non-*SynchroNet* transactions, the value of this parameter is NULL.

Example:

```
GET_PARAM('SYS_TRAN_ID');
```

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. For non-*SynchroNet* transactions, the value of this parameter is NULL.

Example:

```
GET_PARAM('SYS_TRAN_USERID');
```

SYS_IS_PROPAGATE

This parameter value is 'YES' if the transaction is a propagated transaction that is being executed in the master. For non-*SynchroNet* transactions, the value of this parameter is NULL.

Example:

```
GET_PARAM('SYS_IS_PROPAGATE');
```

Updateable System Parameters

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', using the PUT_PARAM function causes *SynchroNet* 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.

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');
```

SYS_ERROR_CODE

This parameter can be used together with the SYS_ROLLBACK parameter. Users can set their own error code to 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');
```

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:

```
PUT_PARAM('SYS_ERROR_TEXT','User defined error text');
```

Database level system parameters

SYS_TRAN_MAXRETRY

This parameter provides a way to handle concurrency conflicts and deadlocks, which occur when transactions update or delete rows. When set, it retries a transaction that has failed due to a conflict. The value for SYS_TRAN_MAXRETRY specifies the maximum number of retries. The value of this parameter can be set only in the replica database.

Valid values for the parameter are between 0 - *n*

where *n* can be any number.

When the 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_R_MAXBYTES_OUT

This parameter can be used for setting the maximum length of synchronization messages that can be sent from the replica database to the master database. The value of this parameter can be set only in the replica database.

Valid values for the parameter are between 0 – 2 GB. If the value of the parameter is set to 0 then default value (2GB) is used.

If the master database receives a message longer than the value of SYS_R_MAXBYTES_OUT, *SynchroNet* issues the following error message:

`25042 - Message is too long (<number> bytes) to forward. Maximum is set to <number> bytes.`

Example:

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

SYS_R_MAXBYTES_IN

This parameter can be used for setting the maximum length of synchronization messages that can be received to the replica database from the master database. The value of this parameter can be set only in the replica database.

Valid values for the parameter are between 0 – 2 GB. If the value of the parameter is set to 0 then default value (2GB) is used.

If the master database sends a message longer than value of SYS_R_MAXBYTES_IN, *SynchroNet* issues the following error message:

`25043 - Replymessage is too long (<number> bytes). Maximum is set to <number> bytes.`:

Example:

```
SET SYNC PARAMETER SYS_R_MAXBYTES_IN '1048576000';
```

Part IV

Appendixes

Part IV provides detailed information about system tables, error messages, configuration parameters, reserved words, and standard SOLID SQL syntax and data types. It contains the following appendixes:

- *Appendix A, “System Tables for Data Synchronization”*
- *Appendix B, “Error Codes”*
- *Appendix C, “Configuration Parameters”*
- *Appendix D, “Data Types”*
- *Appendix E, “SOLID SQL Syntax”*
- *Appendix F, “Database System Views and System Tables”*
- *Appendix G, “Reserved Words”*
- *Appendix H, “SOLID SynchroNet Command Line Options”*

A

System Tables for Data Synchronization

SOLID *SynchroNet* contains a number of system tables that are used for implementing synchronization functionality. In general, these tables are for internal use only. However, you may need to know the contents of these tables when developing and troubleshooting a new application.

Note that the tables are presented in alphabetical order.

SYS_BULLETIN_BOARD

This table contains persistent parameters that are always available in the parameter bulletin board when transactions are executed in this database.

Column Name	Description
PARAM_NAME	Name of the persistent parameter
PARAM_VALUE	Value of the parameter
PARAM_CATALOG	Defines the master/replica catalog

SYS_PUBLICATION_ARGS

This table contains the publication input arguments in this master database

Column Name	Description
PUBL_ID	Internal ID of the publication
ARG_NUMBER	Sequence number of the argument
NAME	Name of the argument
TYPE	Type of the argument
LENGTH_OR_PRECISION	Length or precision of the argument.

Column Name	Description
SCALE	Scale of the argument

SYS_PUBLICATION_REPLICA_ARGS

This table contains the definition of the publication arguments in a replica database.

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is subscribed
PUBL_ID	Internal ID of the publication
ARG_NUMBER	Sequence number of the argument
NAME	Name of the argument
LENGTH_OR_PRECISION	Length or precision of the argument
SCALE	Scale of the argument

SYS_PUBLICATION_REPLICA_STMTARGS

This table contains the mapping between the publication arguments and the statements in the replica.

Column Name	Description
MASTER_CATALOG	Name of the master catalog from which the data is subscribed
MASTER_ID	Internal ID of the master from which the data is subscribed
PUBL_ID	Internal ID of the publication
STMT_NUMBER	Sequence number of the statement
STMT_ARG_NUMBER	Sequence number of the statement
PUBL_ARG_NUMBER	Sequence number of the publication argument

SYS_PUBLICATION_REPLICA_STMTS

This table contains the definition of the publication statements in a replica database.

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is subscribed
PUBL_ID	Internal ID of the publication

Column Name	Description
STMT_NUMBER	Sequence number of the statement
REPLICA_CATALOG	Name of the target catalog in the replica database
REPLICA_SCHEMA	Name of the target schema in the replica database
REPLICA_TABLE	Name of the target table in the replica database
TABLE_ALIAS	Alias name of the target table
REPLICA_FROM_STR	SQL FROM tables as string
WHERE STR	SQL WHERE arguments as string
LEVEL	Level of this SQL statement in this publication hierarchy

SYS_PUBLICATION_STMTARGS

This table contains mapping between the publication arguments and the statements in the master database.

Column Name	Description
PUBL_ID	Internal ID of the publication
STMT_NUMBER	Sequence number of the statement
STMT_ARG_NUMBER	Sequence number of the statement argument
PUBL_ARG_NUMBER	Sequence number of the publication argument

SYS_PUBLICATION_STMTS

This table contains the publication statements in the master database.

Column Name	Description
PUBL_ID	Internal ID of the publication
MASTER_SCHEMA	Name of the publication schema in the master database
MASTER_TABLE	Name of the table in the master database
REPLICA_SCHEMA	Name of the schema in the replica database
REPLICA_TABLE	Name of the table in the replica database
TABLE_ALIAS	The alias name of the target table
MASTER_SELECT_STR	SQL SELECT INTO columns as string

Column Name	Description
REPLICA_SELECT_STR	SQL SELECT INTO columns as string
MASTER_FROM_STR	SQL SELECT FROM tables as string
REPLICA_FROM_STR	SQL SELECT FROM tables as string
WHERE_STR	SQL WHERE arguments as a string
DELETEFLAG_STR	For internal use
LEVEL	Level of this SQL statement in the publication hierarchy

SYS_PUBLICATIONS

This table contains the publications that have been defined in this master database.

Column Name	Description
ID	Internal ID of the publication
NAME	Name of the publication
CREATOR	User ID of the creator of the publication
CREATTIME	Date and time when the publication was created
ARGCOUNT	Number of input arguments for this publication
STMTCOUNT	Number of statement contained in this publication
TIMEOUT	N/A
TEXT	Contents of the CREATE PUBLICATION statement
PUBL_CATALOG	Defines the master catalog

SYS_PUBLICATIONS_REPLICA

This table contains publications that are being used in this replica database.

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is subscribed
ID	Internal ID of the publication
NAME	Name of the publication
CREATOR	User ID of the creator of the publication
ARGCOUNT	Number of input arguments for this publication

Column Name	Description
STMTCOUNT	Number of statements contained by this publication

SYS_SYNC_BOOKMARKS

This table contains bookmarks that are being used in a master database.

Column Name	Description
BM_ID	Internal ID of the bookmark
BM_CATALOG	Reserved for future use
BM_NAME	Name of the bookmark
BM_VERSION	Internal version information of the bookmark in the master
BM_CREATOR	User ID of the creator of the bookmark
BM_CREATIME	Create time of the bookmark

SYS_SYNC_INFO

This table contains synchronization information, one row for each node.

Column Name	Description
NODE_NAME	Master or replica node
NODE_CATALOG	Catalog where node belongs
IS_MASTER	IF YES, this node is a master
IS_REPLICA	If YES, this node is a replica
CREATIME	Node create data and time
CREATOR	Node creator user name

SYS_SYNC_HISTORY_COLUMNS

For future use.

SYS_SYNC_MASTER_MSGINFO

This table contains information about the currently active message in the master database.

Data in this table is used to control the synchronization process between the replica and master database. This table also contains information that is useful for troubleshooting purposes. If the execution of a message halts in the master database due to an error, you can query this table to obtain the reason for the problem, as well as the transaction and statement that caused the error.

Column Name	**Description**
STATE	Current state of the message. The following values are possible: ■ 0 = DELETED N/A (internal non-persistent state) ■ 1 = ERROR - Error has occurred during message processing; the reason for the error was recorded in the error-columns of the row. ■ 10 = RECEIVED - master has received a message from the replica ■ 11 = SAVED - message has been saved in the master database and is being processed ■ 12 = READY – master has processed the message ■ 13 = SENT – N/A (internal non-persistent state)
REPLICA_ID	ID of the replica database from which the message was sent
MASTER_ID	ID of the database to which the master is sent
MSG_ID	Internal ID of the message
MSG_NAME	Name of the message given by the user
MSG_TIME	Create time of the message
MSG_BYTE_COUNT	Size of the message in bytes
CREATE_UID	ID of the user who created the message
FORWARD_UID	ID of the user who forwarded the message
ERROR_CODE	Code of the error that caused termination of the message execution. You can determine the transaction and statement that caused the error from the TRX_ID and STMT_ID information
ERROR_TEXT	Description of the error that caused termination of the message execution.
TRX_ID	Sequence number of the transaction that caused the error
STMT_ID	Sequence number of the statement of a transaction that caused an error
ORD_ID_COUNT	N/A (internal use only)

Column Name	Description
ORD_ID	N/A (internal use only)

SYS_SYNC_MASTER_RECEIVED_MSGPARTS

This table contains parts of the messages that were received in the master database from a replica database, but not yet processed in the master database.

Column Name	Description
REPLICA_ID	Internal ID of the replica database from which the message was received
MSG_ID	Internal ID of the message
PART_NUMBER	Sequence number of the message part
DATA_LENGTH	Length of the data in the message part
DATA	Data of the message part

SYS_SYNC_MASTER_RECEIVED_MSGS

This table contains messages that were received in the master database from a replica database, but are not yet processed in the master database.

Column Name	Description
REPLICA_ID	Internal ID of the replica database from which the message has been received
MSG_ID	Internal ID of the message
CREATIME	Create time of the message
CREATOR	User ID of the user who created the message.

SYS_SYNC_MASTER_STORED_MSGPARTS

This table contains parts of the message result sets that were created in the master database, but not yet sent to the replica database.

Column Name	Description
REPLICA_ID	Internal ID of the replica database to which the message will be sent
MSG_ID	Internal ID of the message

Column Name	Description
ORDER_ID	Sequence number of the result set
RESULT_SET_ID	Internal ID of the result set
RESULT_SET_TYPE	Type of the result set
PART_NUMBER	Sequence number of the message part in the result set
DATA_LENGTH	Length of the data in the message part in the result set
DATA	Data of the message part

SYS_SYNC_MASTER_STORED_MSGS

This table contains messages that were created in the master database, but not yet sent to the replica database.

Column Name	Description
REPLICA_ID	Internal ID of the replica database to which the message will be sent
MSG_ID	Internal ID of the message
CREATIME	Create time of the message
CREATOR	User ID of the user who created the message.

SYS_SYNC_MASTER_SUBSC_REQ

This table contains the list of requested subscriptions waiting to be executed in the master.

Column Name	Description
BOOKMARK_ID	Internal ID of the bookmark in the subscription
REPLICA_ID	Internal ID of the replica from which the statement has arrived
MSG_ID	Internal ID of the message in which the statement has arrived
ORD_ID	Sequence number of the subscription
TRX_ID	Internal ID of the transaction to which the subscription belongs
STMT_ID	Internal ID of the statement in the subscription
REQUEST_ID	N/A
PUBL_ID	Internal ID of the subscribed publication
VERSION	Internal version information of the subscription in the master

Column Name	Description
BOOKMARK_ID	Internal ID of the bookmark in the subscription
REPLICA_VERSION	Internal version information of the subscription in the replica
FULLSUBSC	Indicates if the subscription is full or incremental

SYS_SYNC_MASTER_VERSIONS

This table contains the list of subscriptions (that have been subscribed) to replica databases from the master database.

Column Name	Description
REPLICA_ID	Internal ID of the replica database
REQUEST_ID	Sequence number of the subscription
VERS_TIME	Create time of the subscription
PUBL_ID	ID of the publication.
TABNAME	Name of the table of the publication
TABSCHEMA	Name of the schema of the table
PARAM_CRC	N/A (for internal use only)
PARAM	Parameters of the publication in binary format
VERSION	Version of the data that has been requested from the replica database

SYS_SYNC_MASTERS

This table contains the list of master databases accessed by the replica.

Column Name	Description
NAME	Given name of the master database
ID	Internal ID of the master database
REMOTE_NAME	N/A
REPLICA_NAME	Given name of the replica database
REPLICA_ID	Surrogate identifier for the replica database
REPLICA_CATALOG	Defines the replica catalog which is registered to this master
CONNECT	Connect string of the master database

Column Name	Description
CREATOR	ID of the user who set the database as a master
ISDEFAULT	For future use.

SYS_SYNC_RECEIVED_STMTS

This table contains the propagated statements that have been received in the master database.

Column Name	Description
REPLICA	Internal ID of the replica from which the statement has arrived
MSG	Internal ID of the message in which the statement has arrived
ORD_ID	N/A
TXN_ID	Internal ID of the transaction to which the statement belongs
ID	Sequence number of the statement within the transaction
CLASS	Type of the constant
STRING	the SQL statement as a string
ARG_COUNT	Number of parameters bound to the statement
ARG_TYPES	Types of the parameters bound to the statement
ARG_VALUES	Values of the parameters in binary format
USER_ID	ID of the user who has saved the statement
REQUEST_ID	N/A

SYS_SYNC_REPLICA_MSGINFO

This table contains information about currently active messages in the replica database.

Data in this table is used to control the synchronization process between the replica and master database. This table also contains information that is useful for troubleshooting purposes. If the execution of a message halts in the replica database due to an error, you can query this table to obtain the reason for the problem, as well as the transaction and statement that caused the error.

Column Name	Description
STATE	Current state of the message. The following values are possible: ▪ 0 = DELETED N/A (internal non-persistent state) ▪ 1 = ERROR - Internal error has occurred during message processing; the reason for the error was recorded in the error-columns of the row. ▪ 20 = R_INIT – N/A (internal non-persistent state) ▪ 21 = R_INITEND - N/A (internal non-persistent state) ▪ 22 = R_SAVED - Replica has saved an outgoing message ▪ 23 = R_SENT - Replica has sent a message to the master ▪ 24 = R_RECEIVED - Replica has received a reply message from the master ▪ 25 = R_EXECUTE - The reply message in a replica is ready for execution ▪ 26 = R_EXECUTE_NOTIFYMASTER - Replica has received a reply, but not yet confirmed it with the master
MASTER_ID	ID of the master database to which the message is sent
MASTER_NAME	Name of the master database to which the message is sent
MSG_ID	Internal ID of the message
MSG_NAME	Name of the message given by the user
MSG_TIME	Create time of the message
MSG_BYTE_COUNT	Size of the message in bytes
CREATE_UID	ID of the user who created the message
FORWARD_UID	ID of the user who sent the message
ERROR_CODE	Code of the error that caused the message execution to terminate
ERROR_TEXT	Description of the error that caused the message execution to terminate

SYS_SYNC_REPLICA_RECEIVED_MSGPARTS

This table contains parts of the reply messages that have been received into the replica database from the master database, but are not yet processed in the replica database.

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received
MSG_ID	Internal ID of the message
PART_NUMBER	Sequence number of the message part
DATA_LENGTH	Length of the data in the message part
RESULT_SET_TYPE	Type of the result set
DATA	Data of the message part

SYS_SYNC_REPLICA_RECEIVED_MSGS

This table contains reply messages that were received in the replica database from the master database, but not yet processed in the replica database.

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received
MSG_ID	Internal ID of the message
CREATIME	Create time of the message
CREATOR	User ID of the user who created the message.

SYS_SYNC_REPLICA_STORED_MSGS

This table contains messages that were created in the replica database, but not yet sent to the master database.

Column Name	Description
MASTER_ID	Internal ID of the master database to which the message will be sent
MSG_ID	Internal ID of the message
CREATIME	Create time of the message
CREATOR	user ID of the user who has created the message.

SYS_SYNC_REPLICA_STORED_MSGPARTS

This table contains parts of the messages that were created in the replica database, but not yet sent to the master database.

Column Name	Description
MASTER_ID	Internal ID of the master database to which the message will be sent
MSG_ID	Internal ID of the message
PART_NUMBER	Sequence number of the message part
DATA_LENGTH	Length of the data in the message part
DATA	Data of the message part

SYS_SYNC_REPLICA_VERSIONS

This table contains the list of subscriptions (that have been subscribed) to this replica database from the master database.

Column Name	Description
BOOKMARK_ID	Internal ID of the bookmark in the subscription
REQUEST_ID	Internal ID of the publication request in the subscription
VERS_TIME	Create time of the subscription
PUBL_ID	ID of the subscribed publication
MASTER_ID	ID of the master database from which the publication has been subscribed
PARAM_CRC	Internal use only
PARAM	Parameters of the subscription
VERSION	Version number of subscribed publication in the master database
LOCAL VERSION	Version number of subscribed publication in the replica database
PUBL_NAME	Name of the publication
REPLY_ID	ID of the publication reply

SYS_SYNC_REPLICAS

This table contains the list of replica databases registered with the master.

Column Name	Description
NAME	Given name of the replica database
ID	Internal ID of the replica database
MASTER_NAME	N/A
MASTER_CATALOG	Defines the catalog where the replica is registered
CONNECT	N/A

SYS_SYNC_SAVED_STMTS

This table contains statements that have been saved in replica database for later propagation.

Column Name	Description
MASTER	Internal ID of the master database to which the statement will be propagated
TRX_ID	Internal ID of the transaction to which the statement belongs
ID	Sequence number of the statement within the transaction
CLASS	Type of the constant
STRING	The SQL statement as a string
ARG_COUNT	Number of parameters bound to the statement
ARG_TYPES	Types of parameters bound to the statement
ARG_VALUES	Values of the parameters in binary format
USER_ID	ID of the user who has saved the statement
REQUEST_ID	N/A

SYS_SYNC_SAVED_STMTS

This table contains statements that have been saved in replica database for later propagation.

Column Name	Description
MASTER	Internal ID of the master database to which the statement will be propagated

Column Name	Description
TRX_ID	Internal ID of the transaction to which the statement belongs
ID	Sequence number of the statement within the transaction
CLASS	Type of the constant
STRING	The SQL statement as a string
ARG_COUNT	Number of parameters bound to the statement
ARG_TYPES	Types of parameters bound to the statement
ARG_VALUES	Values of the parameters in binary format
USER_ID	ID of the user who has saved the statement
REQUEST_ID	N/A

SYS_SYNC_USERMAPS

This table maps replica user ids to master users in the SYS_SYNC_USERS table.

Column Name	Description
REPLICA_UID	Replica user ID mapped to master user.
MASTER_ID	Master ID
REPLICA_USERNAME	Replica user name
MASTER_USERNAME	Master user name
PASSW	Encrypted password for master user name

SYS_SYNC_USERS

This table contains a list of users that have access to the synchronization functions of the replica database. These functions include saving transactions and creating synchronization messages.

In a replica the data of this table is downloaded from the master in a message with the command:

MESSAGE *unique-message-name* APPEND SYNC_CONFIG

['*sync-config-arg*']

Column Name	Description
MASTER_ID	Internal ID of the master database
ID	Internal ID of the user
NAME	User name
PASSW	Encrypted password of the user

B

Error Codes

Error Categories

Synchronization Errors

These errors are may be encountered when creating or maintaining the *SynchroNet* environment. They occur when using specific *SynchroNet* commands, which are SOLID SQL extensions.

SQL Errors

These errors are caused by erroneous SQL statements and are detected by the SOLID SQL Parser. Administrative actions are not needed.

Database Errors

These errors are detected by the SOLID *SynchroNet* and may demand administrative actions.

Executable Errors

These errors are caused by the failure of a SOLID *SynchroNet* executable or a command line argument related error. They enable implementing intelligent error handling logic in system startup scripts.

System Errors

These errors are detected by the operating system and demand administrative actions.

Table Errors

These errors are caused by erroneous SQL statements and detected by SOLID *SynchroNet*. Administrative actions are not needed.

Server Errors

These errors are caused by erroneous administrative actions or client requests. They may demand administrative actions.

Communication Errors

These errors are caused by network errors or faulty configuration of the SOLID *SynchroNet* software. These errors demand administrative actions.

Procedure Errors

These errors are caused by errors in the definition or execution of a stored procedure. Administrative actions are not needed.

Sorter Errors

These errors are caused by external sorter algorithm errors when solving queries that require ordering rows.

SOLID Synchronization Errors

Error code	Description
25001	Internal error
25002	Internal error
25003	Cannot save SAVE statements. It is not possible to save a "SAVE" statement for later propagation. For example, the following SQL statement returns an error: `SAVE CALL MYPROC(1, ’foo’)` *SynchroNet* commands that return this error: SAVE *sql_statement*
25004	Dynamic parameters are not supported. Input parameters of a subscription must be given as literals. They cannot be dynamically bound to the statement. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND SUBSCRIBE *publication_name*

Error code	**Description**
25005	Message *message_name* is already active. A message of the specified name that was created appears to still be active. A message becomes active when the following MESSAGE command is executed: MESSAGE *message_name* BEGIN The message is automatically deleted when the reply of the message has been successfully executed in the replica database. *SynchroNet* commands that return this error: MESSAGE *message_name* BEGIN
25006	Message *message_name* not active A message has already been committed or ended using the MESSAGE END statement. New tasks cannot be appended to the message using the MESSAGE APPEND command. Probable cause for this error is that the AUTOCOMMIT mode is used in the connection. You must first remove the message with MESSAGE *message_name* DELETE command. Then switch autocommit off and run the script again. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND *synchronization_task*
25007	Master *master_name* not found A replica attempts to perform an operation to a master database that cannot be found. *SynchroNet* commands that return this error: SET SYNC CONNECT *connect_string* TO MASTER *master_name* DROP MASTER *master_name* SAVE *sql_statement*
25008	Internal error
25009	Replica *replica_name* not found The replica name specified in a command cannot be found. *SynchroNet* commands that return this error: DROP SUBSCRIPTION *publication_name*(*parameter_list*) [FROM REPLICA *replica_name*] MESSAGE *message_name* [FROM REPLICA *replica_name*] DELETE

Error code	**Description**
25010	Publication *publication_name* not found. The publication name of a subscription is incorrect. *SynchroNet* commands that return this error: MESSAGE APPEND SUBSCRIBE *publication_name*(*parameter_list*) DROP PUBLICATION *publication_name*(*parameter_list*) [FROM REPLICA *replica_name*]
25011	Wrong number of parameters to publication *publication_name*. A subscription to a publication contains incorrect number of parameters. The data types of the given subscription parameters must match the input parameter definition of the publication. *SynchroNet* commands that return this error: DROP SUBSCRIPTION publication_name (*parameter_list*) [FROM REPLICA *replica_name*] MESSAGE *message_name* APPEND SUBSCRIBE *publication_name* (*parameter_list*)
25012	Message reply timed out. A reply message has not arrived to the replica database within the given timeout. The reason is that the reply message is not yet ready in the master database. The message needs to be retrieved later using “MESSAGE *message_name* GET REPLY” command. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds*

Error code	Description
25013	Message name *message_name* not found. The message with the given name does not exist. The message name is given, when the message is created with command MESSAGE message_name BEGIN. The message name is released when the reply message has been successfully executed in the replica database. Message names must be unique within the replica database. A message can be deleted from the database with command MESSAGE *message_name* [FROM REPLICA *replica_name*] DELETE. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND MESSAGE *message_name* END MESSAGE *message_name* FORWARD MESSAGE *message_name* GET REPLY MESSAGE *message_name* DELETE
25014	Internal error
25015	Syntax error: *error_message*, line *line_number* Publication syntax is not correct. See the CREATE PUBLICATION syntax reference for correct syntax. *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name*
25016	Message not found, replica id *replica_id*, message id *message_id* Message not found in master during processing. This can happen if the message is explicitly deleted in master. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD MESSAGE *message_name* GET REPLY MESSAGE *message_name* RESTART

Error code	Description
25017	No unique key found for table *table_name*. The primary key for the table has not been defined. Each table that is part of an incremental publication must have a primary key defined. The synchronization history mechanism cannot function without explicitly defined primary keys. *SynchroNet* commands that return this error: ALTER TABLE *table_name* SET SYNCHISTORY
25018	Illegal message state. An internal error has occurred in the message processing. It is not possible to continue executing the message after this error. Delete the message using the following command: MESSAGE *message_name* [FROM REPLICA *replica_name*] DELETE *SynchroNet* commands that return this error: MESSAGE *message_name*…
25019	Database is not a replica. A synchronization message can be created only in a database that has been registered to be a replica database. For information on registering a replica database, see the example code from "Getting started with SOLID *SynchroNet*. " *SynchroNet* commands that return this error: MESSAGE *message_name* BEGIN
25020	Database is not a master. A command that can be executed only in a master database has been attempted to execute in a non-master database. A database can be set to be a master database of a system by entering the following command: SET SYNC MASTER YES *SynchroNet* commands that return this error: MESSAGE *message_name* FROM REPLICA *replica_name* RESTART MESSAGE *message_name* FROM REPLICA *replica_name* DELETE DROP REPLICA *replica_name* DROP SUBSCRIPTION *subscription_name* FROM REPLICA *replica_name*

Error code	Description
25021	Database is not master or replica database. In order to create or drop publication definitions or set the SYNCHISTORY property of a table, the database must be defined to be either master or replica (or both). *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name…* DROP PUBLICATION *publication_name* ALTER TABLE *table_name* SET SYNCHISTORY
25022	User generated error. The execution of a transaction has been cancelled and rolled back in the master database. Because of the failed transaction, the execution of the message that contained the transaction has been stopped. User can request *SynchroNet* to roll back a transaction by setting following parameters to the bulletin board of the transaction: PutParam('SYS_ROLLBACK', 'YES') PutParam('SYS_ERROR_CODE', *numeric_value_as_string)* PutParam('SYS_ERROR_TEXT', *error_text_as_string*) If the SYS_ERROR_CODE parameter is not specified or it contains an invalid value, the error number 25022 is returned. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds*
25023	Replica registration failed. An error has occurred during replica registration. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds*

Error code	Description
25024	Master not defined. No definition for the master exists or the configuration changed during message processing. *SynchroNet* was unable to properly initialize the synchronization environment. You can check the master from the replica's system table SYS_SYNC_MASTERS. All successfully registered replicas are found from the master database system table SYS_SYNC_REPLICAS. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* MESSAGE *message_name* APPEND SUBSCRIBE *publication_name*
25025	Node name not defined. Before setting up a master database or registering a replica database, the node name of the database must be set. This can be done with the following command: SET SYNC NODE *node_name* *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND REGISTER REPLICA
25026	Not a valid master user. A user who has not been defined in the master database, attempts to perform a *SynchroNet* SQL command. The user definitions of the master database can be downloaded to replica with the following command: MESSAGE *message_name* APPEND SYNC_CONFIG *SynchroNet* commands that return this error: SAVE *sql_statement* MESSAGE *message_name…*
25027	Internal error.
25028	Message *message_name* can include only one system subscription. System subscriptions (REGISTER REPLICA and SYNC_CONFIG) must be kept in separate messages. These tasks must be the only ones of their messages. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND SUBSCRIBE *publication_name*
25029	Internal error.

Error code	**Description**
25030	Replica *replica_name* is already registered. A replica attempts to register itself using a name that is already in use. Replica names must be unique. If you know that the chosen replica name is no longer used by any other replicas, drop it from the master database with the command DROP REPLICA *replica_name*. Then register the replica again. Otherwise, change the newly created replica's name and register it again. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds*
25031	Transaction is active, operation failed. A replica attempts to process a message when having an active transaction. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* MESSAGE *message_name* EXECUTE
25032	All publication SQL statements must return rows. The publication definition contains SQL operations that don't return rows. Only SELECT statements are allowed in the publication. *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name*
25033	Publication *publication_name* already exists. A publication has been attempted to create with a name that is already in use. *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name*
25034	Message name *message_name* already exists. Each message must have a name that is unique within the database. *SynchroNet* commands that return this error: MESSAGE *message_name* BEGIN

Error code	**Description**
25035	Message *message_name* is in use. A *SynchroNet* message is locked when executed or attempted to delete. A locked message cannot be re-executed or deleted. If you get this error while attempting to create a new *SynchroNet* message, it is probably due to an existing message with the same name. You can check existing messages from the system table SYS_SYNC_REPLICA_MSGINFO in the replica or from the system table SYS_SYNC_MASTER_MSGINFO in the master database. *SynchroNet* commands that return this error: MESSAGE *message_name* BEGIN MESSAGE *message_name* FROM REPLICA *replica_name* DELETE MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds*
25036	Publication *publication_name* not found or publication version mismatch A publication has been dropped or redefined at master during message processing. Recover by DROP SUBSCRIPTION at replica. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* MESSAGE *message_name* EXECUTE
25037	Publication column count mismatch in table *table_name* Database definitions at master and replica do not match. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* MESSAGE *message_name* EXECUTE
25038	Table is referenced in publication *publication_name*, drop or alter operations are not allowed A table which is referenced in a publication can not be dropped or altered. *SynchroNet* commands that return this error: DROP TABLE *table_name* ALTER TABLE *table_name*

Error code	Description
25040	User id *user_id* is not found. User information has been changed at the replica during message execution. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* MESSAGE *message_name* EXECUTE
25041	Subscription to publication *publication_name* not found. The subscription that is expected to be in the replica is not found. This error occurs if the subscription is explicitly dropped at the replica. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY TIMEOUT *timeout_in_seconds* DROP SUBSCRIPTION *subscription_name* DROP SUBSCRIPTION *subscription_name* REPLICA r*eplica_name*
25042	Message is too long (*number* bytes) to forward. Maximum is set to *number* bytes. The length of a message to be forwarded exceeds the limit for message's length. The limit can be set by variable SYS_R_MAXBYTES_OUT. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD
25043	Replymessage is too long (*number* bytes). Maximum is set to *number* bytes. The length of a message to be received as a reply exceeds the limit for message's length. The limit can be set by variable SYS_R_MAXBYTES_IN. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY
25044	SYNC_CONFIG system publication takes only character arguments. Publication SYNC_CONFIG has been attempted to be subscribed with arguments of wrong type. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND SUBSCRIBE SYNC_CONFIG

Error code	Description
25045	Master/replica node support disabled. A node can be either a replica or a master but not both. *SynchroNet* commands that return this error: SET SYNC MASTER YES SET SYNC REPLICA YES
25046	Commit and rollback are not supported in propagated transactions. This error is caused when a transaction attempts to execute a COMMIT or ROLLBACK command in the master database. The error is returned to the *SynchroNet* engine running the procedure. The message containing the procedure will fail.
25047	Internal error.
25048	Publication *publication_name* request info not found. A publication has been dropped while message is being executed. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY
25049	Referenced table *table_name* not found in subscription hierarchy. A publication has referenced a table which does not exist. *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name*
25051	Unfinished messages found. Replica mode has been attempted to be switched off while there are messages either waiting to be forwarded or being executed at master. *SynchroNet* commands that return this error: SET SYNC REPLICA NO
25052	Node *node_name* already exists. A replica has been attempted to be registered with a name that already exists. The registration is appended to the message by MESSAGE *message_name* APPEND REGISTER REPLICA. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY
25053	Internal error.

Error code	Description
25054	Table *table_name* is not set for synchronization history. A table in the master database has the SYNCHISTORY property set, but the corresponding table in the replica does not. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds*
25055	Connect information is allowed only when not registered. The connect info in MESSAGE *message_name* FORWARD TO *connect_info options* is allowed only if the replica has no yet been registered to the master database. *SynchroNet* commands that return this error: MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds*
25056	Autocommit not allowed. The *SynchroNet* statement must be executed with autocommit mode turned off. *SynchroNet* commands that return this error: All MESSAGE *message_name* ... commands DROP SUBSCRIPTION *subscription_name* DROP SUBSCRIPTION *subscription_name* REPLICA r*eplica_name* DROP REPLICA *replica_name* DROP MASTER *master_name* EXPORT SUBSCRIPTION IMPORT '*filename'*
25057	Already registered to master *master_name*. The replica database has already been registered to a master database. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY (when registering a replica) MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds* (when registering a replica)
25058	Internal error.

Error code	**Description**
25059	After registration nodename cannot be changed. The SYNC NODE NAME property of a database cannot be changed if the master has any registered replicas or replica has already been registered to a master database. *SynchroNet* commands that return this error: SET SYNC NODE NAME *unique_node_name*
25060	Column *column_name* does not exist on publication *publication_name* resultset in table *table_name.* This error occurs when a replica finds out that the master is transferring data that does not include primary key values that the replica requires. *SynchroNet* commands that return this error: MESSAGE *message_name* GET REPLY MESSAGE *message_name* FORWARD TIMEOUT *timeout_in_seconds*
25061	Where condition for table *table_name* must refer to an outer table of the publication. If a publication contains nested SELECTs, the WHERE clause of the inner SELECT must refer to the outer table of the outer SELECT. *SynchroNet* commands that return this error: CREATE PUBLICATION *publication_name*
25062	User *user_id* is not mapped to master *user_id*. Dropping the user mapping failed because user is not mapped to a given master. *SynchroNet* commands that return this error: ALTER USER *replicauser* SET MASTER *mastername* USER
25063	User *user_id* is already mapped to master *user_id.* User is already mapped to a given master. *SynchroNet* commands that return this error: ALTER USER *replicauser* SET MASTER *mastername* USER
25064	Unfinished message *message_name* found for replica *replica_name*. Dropping the replica failed because there are unfinished messages. *SynchroNet* commands that return this error: DROP REPLICA

Error code	Description
25065	Unfinished message *message_name* found for master *master_name.* Dropping the master failed because there are unfinished messages. *SynchroNet* commands that return this error: DROP MASTER
25066	Synchronization bookmark *bookmark_name* already exists. Cannot create synchronization bookmark since the name already exists. *SynchroNet* commands that return this error: CREATE SYNC BOOKMARK
25067	Synchronization bookmark *bookmark_name* not found. Bookmark name is not an existing bookmark. *SynchroNet* commands that return this error: DROP SYNC BOOKMARK
25068	Export file *file_name* open failure. Failed to open export file for EXPORT SUBSCRIPTION. *SynchroNet* commands that return this error: EXPORT SUBSCRIPTION
25069	Import file *file_name* open failure. Failed to open import file for IMPORT. *SynchroNet* commands that return this error: IMPORT *file_name*
25070	Statements can be saved only for one master in transaction. Statements cannot be saved for multiple masters in one transaction. *SynchroNet* commands that return this error: SAVE
25071	Not registered to publication *publication_name*. Replica must be registered to a publication before the publication can be subscribed to the replica. *SynchroNet* commands that return this error: MESSAGE *message_name* APPEND SUBSCRIBE *publication_name*

Error code	Description
25072	Already registered to publication *publication_name*. Replica is already registered to a publication. *SynchroNet* commands that return this error: REGISTER REPLICA
25074	User definition not allowed for this operation. Master user attempts to perform synchronization operation, but is denied access in the replica database because the registration user is still the active user. After the registration process, the command SET SYNC *username* must be set to NONE. *SynchroNet* commands that return this error: SAVE *sql_statement* DROP SUBSCRIPTION *publication_name* (in replica) MESSAGE *message_name* APPEND SUBSCRIBE *publication_name* MESSAGE *message_name* APPEND PROPAGATE TRANSACTIONS MESSAGE *message_name* APPEND REGISTER PUBLICATION MESSAGE *message_name* APPEND UNREGISTER PUBLICATION MESSAGE *message_name* EXECUTE (in replica)
25082	Node name can not be removed if node is master or replica. Node name cannnot be set to NONE on a synchronized master and/or replica catalog. *SynchroNet* commands that return this error: SET SYNC NODE NONE

SOLID SQL Errors

Error code	Description
SQL Error 1	Parsing error ‘syntax error’ The SQL parser could not parse the SQL string. Check the syntax of the SQL statement and try again.

Error code	**Description**
SQL Error 2	Table *table* can not be opened You may not have privileges to access the table and its data.
SQL Error 3	Table *table* can not be created Table can not be created. You may not have privileges for this operation.
SQL Error 4	Illegal type definition *column* A column type in your CREATE TABLE statement is illegal. Use a legal type for the column.
SQL Error 5	Table table can not be dropped Table can not be dropped. Only the owner (that is, the creator) can drop it.
SQL Error 6	Illegal value specified for column *column* The value specified for column is invalid. Check the value for the column.
SQL Error 7	Insert failed The server failed to do the insertion. You may not have INSERT privilege on the table or it may be locked.
SQL Error 8	Delete failed The server failed to do the deletion. You may not have DELETE privilege on the table or the row may be locked.
SQL Error 9	Row fetch failed The server failed to fetch a row. You may not have SELECT privilege on the table or there may be an exclusive lock on the row.
SQL Error 10	View *view* can not be created You cannot create this view. You may not have SELECT privilege on one or more tables in the query-specification of your CREATE VIEW statement.
SQLError11	View *view* cannot be dropped. You cannot drop this view. Only the owner (i.e. the creator) of the view can drop it.

Error code	Description
SQLError12	Illegal view definition *view* The view definition is illegal. Check the syntax of the definition.
SQLError13	Illegal column name *column* Column name is illegal. Check that the name is not a reserved name.
SQL Error 14	Call to function *function* failed Function call to function failed. Check the arguments and their types.
SQL Error 15	Arithmetic error An arithmetical error occurred. Check the operators, values and types.
SQL Error 16	Update failed The server failed to update a row. There may a lock on a row.
SQL Error 17	View is not updatable This view is not updatable. UPDATE, INSERT and DELETE operations are not allowed.
SQL Error 18	Inserted row does not meet check option condition You tried to insert a row, but one or more of the column values do not meet column constraint definition.
SQL Error 19	Updated row does not meet check option condition You tried to update a row, but one or more of the column values do not meet column constraint definition.
SQL Error 20	Illegal CHECK constraint A check constraint given to the table is illegal. Check the types of the check constraint of this table.
SQL Error 21	Insert failed because of CHECK constraint You tried to insert a row, but the values do not meet the check option conditions.

Error code	**Description**
SQL Error 22	Update failed because of CHECK constraint You tried to update a row, but the values do not meet the check option conditions.
SQL Error 23	Illegal DEFAULT value The DEFAULT value for the column given is illegal.
SQL Error 25	Duplicate columns in INSERT column list You have included a column in column list twice. Remove duplicate columns.
SQL Error 26	At least one column definition required in CREATE TABLE You need to specify at least one column definition in a CREATE TABLE statement.
SQL Error 27	Illegal REFERENCES column list There are wrong number of columns in your REFERENCES list.
SQL Error 28	Only one PRIMARY KEY allowed in CREATE TABLE You can use only one PRIMARY KEY in CREATE TABLE.
SQL Error 29	GRANT failed Granting privileges failed. You may not have privileges for this operation.
SQL Error 30	REVOKE failed Revoking privileges failed. You may not have privileges for this operation.
SQL Error 31	Multiple instances of a privilege type You tried to grant privileges to a role or a user. You have included multiple instances of a privilege type in the list of privileges.
SQL Error 32	Illegal constant *constant* Illegal constant was found. Check the syntax of the statement.

Error code	Description
SQL Error 33	Column name list of illegal length You have entered different number of columns in CREATE VIEW statement to the view and to the table.
SQL Error 34	Conversion between types failed An expression in UPDATE statement has illegal type for a column.
SQL Error 35	Column names not allowed in ORDER BY for UNION You can not use column name in an ORDER BY for UNION statement.
SQL Error 36	Nested aggregate functions Nested aggregate functions can not be used. For example: SUM(AVG(*column*)).
SQL Error 37	Aggregate function with no arguments An aggregate function was entered with no arguments. For example: SUM().
SQL Error 38	Set operation between different row types You have tried to execute a set operation of tables with incompatible row types. The row types in a set operation must be compatible.
SQL Error 39	COMMIT WORK failed Committing a transaction failed.
SQL Error 40	ROLLBACK WORK failed Rolling back a transaction failed.
SQL Error 41	Savepoint could not be created A savepoint could not be created.
SQL Error 42	Could not create index *index* An index could not be created. You may not have privileges for this operation. You need to be an owner of the table or have SYS_ADMIN_ROLE to have privileges to create index for the table.

Error code	Description
SQL Error 43	Could not drop index *index* An index could not be dropped. You may not have privileges for this operation. You need to be an owner of the table or have SYS_ADMIN_ROLE to have privileges to drop index from the table.
SQL Error 44	Could not create schema *schema* A schema could not be created.
SQL Error 45	Could not drop schema *schema* A schema could not be dropped.
SQL Error 46	Illegal ORDER BY specification You tried to use an ORDER BY column that does not exist. Refer to an existing column in the ORDER BY specification.
SQL Error 47	Maximum length of identifier is 31 You have exceeded the maximum length for the identifier.
SQL Error 48	Subquery returns more than one row You have used a subquery that returns more than one row. Only subqueries returning one row may be used in this situation.
SQL Error 49	Illegal expression *expression* You tried to insert or update a table using an aggregate function (SUM, MAX, MIN or AVG) as a value. This is not allowed.
SQL Error 50	Ambiguous column name *column* You have referenced a column which is exists in more than one table. Use syntax *table*.*column* to indicate which table you want to use.
SQL Error 51	Non-existent function *function* You tried to use a function which does not exist.
SQL Error 52	Non-existent cursor *cursor* You tried to use a cursor which is not created.

Error code	Description
SQL Error 53	Function call sequence error A function was called in wrong order. Check the sequence and success of the function calls.
SQL Error 54	Illegal use of a parameter A parameter was used illegally. For example: SELECT * FROM TEST WHERE ? < ?;
SQL Error 55	Illegal parameter value A parameter has an illegal value. Check the type and value of the parameter.
SQL Error 56	Only ANDs and simple condition predicates allowed in UPDATE CHECK All search condition predicates are not supported.
SQL Error 57	Opening the cursor did not succeed Server failed to open a cursor. You may not have cursor open at this moment.
SQL Error 58	Column *column* is not referenced in group-by-clause You tried to group rows using *column*. All columns in *group_by_clause* must be listed in your *select_list*. A star (‘*’) notation is not allowed with GROUP BY.
SQL Error 59	Comparison between incompatible types You tried to compare values which have incompatible types. Incompatible types are for example an integer and a date value.
SQL Error 60	Reference to the insert table not allowed in the source query You have referenced in subquery a table where you are inserting values. This is not allowed.
SQL Error 61	Reference to the update table not allowed in subquery You have referenced in subquery a table where you are updating values. This is not allowed.
SQL Error 62	Reference to the delete table not allowed in subquery You have referenced in subquery a table where you are deleting values. This is not allowed.

Error code	Description
SQL Error 63	Subquery returns more than one column You have used a subquery that returns more than one column. Only subqueries returning one column may be used.
SQL Error 64	Cursor *cursor* not updatable The cursor opened is not updatable.
SQL Error 65	Insert or update tried on pseudo column You tried to update a pseudo column (ROWID, ROWVER). Pseudo columns are not updatable.
SQL Error 66	Could not create user *user* A user could not be created. You may not have privileges for this operation.
SQL Error 67	Could not alter user *user* A user could not be altered. You may not have privileges for this operation.
SQL Error 68	Could not drop user *user* A user could not be dropped. You may not have privileges for this operation.
SQL Error 69	Could not create role *role* A role could not be created. You may not have privileges for this operation.
SQL Error 70	Could not drop role *role* A role could not be dropped. You may not have privileges for this operation.
SQL Error 71	Grant *role* failed Granting role failed. You may not have privileges for this operation.
SQL Error 72	Revoke *role* failed Revoking role failed. You may not have privileges for this operation.

Error code	Description
SQL Error 73	Comparison of vectors of different length You have tried to compare row value constructors that have different number of dimensions. For example you have compared (a,b,c) to (1,1).
SQL Error 74	Expression * not compatible with aggregate expression The aggregate expression can not be used with * columns. Specify columns using their names when used with this aggregate expression. This usually happens when GROUP BY expression is used with the * columns.
SQL Error 75	Illegal reference to table *table* You have tried to reference a table which is not in the FROM list. For example: SELECT T1.* FROM T2.
SQL Error 76	Ambiguous table name *table* You have used the syntax *table*.*column name* ambiguously. For example: SELECT T1.* FROM T1 A,T1 B WHERE A.F1=0;
SQL Error 77	Illegal use of aggregate expression You tried to use aggregate expression illegally. For example: SELECT ID FROM TEST WHERE SUM(ID) = 3;
SQL Error 78	Row fetch failed The server failed to fetch a row. You may not have SELECT privilege on the table or there may be an exclusive lock on the row.
SQL Error 79	Subqueries not allowed in CHECK constraint You tried to use subquery in a check constraint.
SQL Error 80	Sorting failed External sorter is out of disk space or cache memory. Modify parameters in configuration file `solid.ini`.
SQL Error 81	SET syntax results in error
SQL Error 82	Improper type used with LIKE
SQL Error 83	Syntax error
SQL Error 84	Parser error *statement*
SQL Error 85	Incorrect number of values for INSERT

Error code	Description
SQL Error 86	Illegal ROWNUM constraint
SQL Error 88	Subquery not allowed in UPDATE expression Subqueries cannot be used with UPDATE statements.
SQL Error 93	Illegal GROUP BY expression GROUP BY expression is illegal.

SOLID Database Errors

Error code	Description
Database Error 10001	Key value is not found. Internal error: a key value cannot be found from the database index.
Database Error 10002	Operation failed. This is an internal error indicating that the index of the table accessed is in inconsistent state. Try to drop and create the index again to recover the error.
Database Error 10004	Redefinition. Unexpected failure occurred in the database engine. This error may also occur during recovery: either an index or a view has been redefined during recovery. The server is not able to do the recovery. Delete log files and start the server again.
Database Error 10005	Unique constraint violation. You have violated a unique constraint. This happens when you have tried to insert or update a column which has a unique constraint and the value inserted or updated is not unique. This may also occur when you create users, tables or roles having same names in separate transactions.

Error code	Description
Database Error 10006	Concurrency conflict, two transactions updated or deleted the same row. Two separate transactions have modified a same row in the database simultaneously. This has resulted in a concurrency conflict.
Database Error 10007	Transaction is not serializable. The transaction committed is not serializable.
Database Error 10010	No checkpoint in database. This error occurs when the server has crashed in the middle of creating a new database. Delete the database and log files and try to create the database again.
Database Error 10011	Database headers are corrupted. The headers in the database are corrupted. This may be caused by a disk error or other system failure. Restore the database from the backup.
Database Error 10012	Node split failed. This is an internal error.
Database Error 10013	Transaction is read-only. You have tried to write inside a transaction that is set read-only. Remove the write operation or unset the read-only mode in the transaction.
Database Error 10014	Resource is locked. This error occurs when you are trying to use a key value in an index which has been concurrently dropped.
Database Error 10016	Log file is corrupted. One of the log files of the database is corrupted. You can not use these log files. Delete them and start the server again.
Database Error 10017	Too long key value. The maximum length of the key value has been exceeded. The maximum value is one third of the size of the index leaf.

Error code	**Description**
Database Error 10019	Backup is active You have tried to start a backup when a backup process is already in progress.
Database Error 10020	Checkpoint creation is active. You have tried to start a checkpoint when a checkpoint creation is already in progress.
Database Error 10021	Failed to delete log file. The deletion of a log file in making a backup has failed. Reasons for the failure can be: ■ The log file has already been deleted from the operating system. ■ The log file has a read-only attribute.
Database Error 10023	Wrong log file, maybe the log file is from another database. The log file in the database directory is from another SOLID *SynchroNet* database. Copy the correct log files to the database directory. The log file in the database directory is from another SOLID *SynchroNet* database. Copy the correct log files to the database directory.
Database Error 10024	Illegal backup directory. The backup directory is either an empty string or a dot indicating that the backup will be created in the current directory.
Database Error 10026	Transaction is timed out. An idle transaction has exceeded the maximum idle transaction time. The transaction has been aborted. The maximum value is set in parameter AbortTimeOut in SRV section. The default value is 120 minutes.
Database Error 10027	No active search. Internal error.

Error code	Description
Database Error 10028	Referential integrity violation, foreign key values exist. You tried to delete a row that is referenced from a foreign key.
Database Error 10029	Referential integrity violation, referenced column values do not exist. The definition of a foreign key does not uniquely identify a row in the referenced table.
Database Error 10030	Backup directory '*directory name*' does not exist. Backup directory is not found. Check the name of the backup directory.
Database Error 10031	Transaction detected a deadlock, transaction is rolled back. Deadlock detected. If necessary, begin transaction again.
Database Error 10032	Wrong database block size specified. The block size of the database file differs from the block-size given in the configuration file `solid.ini`.
Database Error 10033	Primary key unique constraint violation. Your primary key definition is not unique.
Database Error 10034	Sequence name *sequence* conflicts with an existing entity. Choose a unique name for a sequence. The specified name is already used.
Database Error 10035	Sequence does not exist. Check the name of the sequence.
Database Error 10036	Data dictionary operation is active for accessed sequence. Create or drop operation is active for the accessed sequence. Try again.
Database Error 10037	Can not store sequence value, the target data type is illegal. The valid target data types are INTEGER and BINARY.
Database Error 10038	Illegal column value for descending index. Corrupted data found in descending index. Drop the index and create it again.

Error code	Description
Database Error 10040	Log file write failure, probably the disk containing the log files is full. Shut down the server and reserve more disk space for log files.
Database Error 10041	Database is read-only.
Database Error 10042	Database index check failed, the database file is corrupted.
Database Error 10043	Database free block list corrupted, same block twice in free list.
Database Error 10044	Primary key can not contain blob attributes.
Database Error 10046	Operation failed, data dictionary operation is active.
Database Error 10047	Replicated transaction is aborted.
Database Error 10048	Replicated transaction contains schema changes, operation failed.
Database Error 10049	Slave server not available any more, transaction aborted
Database Error 10050	Replicated row contains BLOb columns that cannot be replicated.
Database Error 10054	Opening the database file failed. Probably another SOLID process is already running in the same directory.
Database Error 10055	Too little cache memory has been specified for the SOLID process.
Database Error 10056	Cannot open *database file*. *Error text (number).* Most likely the SOLID process does not have correct access rights to the database file.
Database Error 10057	The database is irrevocably corrupted. Revert to the latest backup.

Error code	Description
Database Error 10058	Database version (*number*) does not match with SOLID version. Possible causes for this error include: ▪ a version of SOLID that is too old is used with this database ▪ the database has been corrupted
Database Error 10059	Database version (*number*) does not match with SOLID version. Possible causes for this error include: ▪ a version of SOLID that is too old is used with this database ▪ the database has been corrupted

Database error 10060	Cannot perform roll-forward recovery in read-only mode.
	
	Read-only mode can be specified in 3 ways. To restart SOLID in normal mode, verify that:
	▪ SOLID process is not started with command-line option **-x** read only
	▪ `solid.ini` does not contain the following parameter setting:
	`[General]`
	`ReadOnly=yes`
	▪ license file does not have read-only limitation

Database error 10061	Out of database cache memory blocks. SOLID process cannot continue because there is too little cache memory allocated for the SOLID process. Typical cause for this problem is a heavy load from several concurrent users. To allocate more cache memory, set the following `solid.ini` parameter to a higher value: `[IndexFile]` `CacheSize=`*`cache size in bytes`* NOTE: Allocated cache memory size should not exceed the amount of physical memory.

Error code	**Description**
Database error 10062	Failed to write to *log filename* at *offset*. Verify that the disk containing the log files is not full and is functioning properly. Also, log files should not be stored on shared disks over the network.
Database error 10063	Cannot create new *log filename* because such a file already exists in the log file directory. Probably your log file directory also contains logs from some other database. SOLID process cannot continue until invalid log files are removed from the log file directory. Remove *log filename* and all other log files with greater sequence numbers.
Database error 10064	Illegal log file name template. Most likely, the log file name template specified in: `[Logging]` `FileNameTemplate=`*`name`* contains too few or too many sequence number digit positions. There should be at least 4 and at most 10 digit positions.
Database error 10066	Cannot open *log filename*. Check the following log file name template in `solid.ini`: `[Logging]` `FileNameTemplate=`*name* and verify that: ■ it can be expanded into a valid file name in this environment ■ SOLID process has appropriate privileges to the log files directory.
Database error 10067	Cannot create database because old *log filename* exists in the log files directory. Possibly the database has been deleted without deleting the log files or there are log files from some other database in the log files directory of the database to be created.

Error code	**Description**
Database error 10068	Roll-forward recovery cannot be performed because the configured log file *block size number* does not match with *block size number* of existing *filename.* To enable recovery, edit `solid.ini` to include parameter setting: `[Logging]` `BlockSize=`*`blocksize in bytes`* and restart the SOLID process. After successful recovery, you can change the log file block size by performing these steps: **1.** Shut down the SOLID process. **2.** Remove old log files. **3.** Edit new block size into `solid.ini` **4.** Restart SOLID.
Database error 10069	Roll-forward recovery failed because *relation id number* was not found. Database has been irrevocably corrupted. Please restore the database.
Database error 10070	Roll-forward failed because *relation id number* was not found. Database has been irrevocably corrupted. Please restore the database from the latest backup.
Database error 10073	Database is inconsistent. Illegal index block type *size*, *address*, *routine*, *reachmode*. Please restore the database from the latest backup.
Database error 10074	Roll-forward recovery failed. Please revert to the latest backup.

Database error 10075	The database you are trying to use has been originally created with different database block size settings than your current settings.
	
	Edit the `solid.ini` file to contain the following parameter setting:
	
	`[IndexFile]`
	`BlockSize=`*`blocksize in bytes`*

Error code	**Description**
Database error 10076	Roll-forward recovery failed because *tablename* or *viewname* is redefined in the log *filename.* Possible causes for this error include: ■ another SOLID process is using the same log file directory ■ old log files are present in the log file directory SOLID process cannot use this corrupted log file to recover. In order to continue, you have the following alternatives: **1.** Revert to the last backup **2.** Revert to the last checkpoint **3.** Revert to the last committed transaction within the last valid log file
Database error 10077	No base catalog given for database conversion (use -C *catalogname)* A database's base catalog must be provided when converting the database to a new format.

SOLID Executable Errors

Error code	**Description**
Executable Error 10	Failed to open database
Executable Error 11	Failed to connect to database
Executable Error 12	Database test failed
Executable Error 13	Database fix failed
Executable Error 14	License error
Executable Error 15	Database must be converted
Executable Error 16	Database does not exist
Executable Error 17	Database exists
Executable Error 18	Database not created
Executable Error 19	Database create failed

Error code	Description
Executable Error 20	Communication init failed
Executable Error 21	Communication listen failed
Executable Error 22	Service operation failed
Executable Error 50	Illegal command line argument
Executable Error 51	Failed to change directory
Executable Error 52	Input file open failed
Executable Error 53	Output file open failed
Executable Error 54	Server connect failed
Executable Error 55	Operation init failed

SOLID System Errors

Error code	Description
System Error 11000	File open failure. The server is unable to open the database file. Reason for the failure can be: ■ The database file has been set read-only. ■ You do not have rights to open the database file in write mode. ■ Another SOLID *SynchroNet* is using the database file. ■ Correct the error and try again.
System Error 11001	File write failure. Server is unable to write to the disk. The database files may have a read-only attribute set or you may not have rights to write to the disk. Add rights or unset read-only attribute and try again.
System Error 11002	File write failed, disk full. Server failed to write to the disk, because the disk is full. Free disk space or move the database file to another disk. You can also split the database file to several disks using the FileSpec_[1-N] parameter in IndexFile section.

Error code	**Description**
System Error 11003	File write failed, configuration exceeded. Writing to the database file failed, because the maximum database file size set in FileSpec_[1-N] parameter is exceeded.
System Error 11004	File read failure. An error occurred reading a file. This may indicate a disk error in your system.
System Error 11005	File read beyond end of file. Internal error.
System Error 11006	File read failed, illegal file address. An error occurred reading a file. This may indicate a disk error in your system.
System Error 11007	File lock failure. The server failed to lock the database file. This error occurs in the Windows version, if you do not have SHARE.EXE loaded. To correct the failure: 1. Exit Windows 2. Load SHARE.EXE 3. Delete the database file SOLID.DB and log files. 4. Start Windows and launch SOLID *SynchroNet*.
System Error 11008	File unlock failure. Server failed to unlock a file.
System Error 11009	File free block list corrupted. internal error.
System Error 11010	Too long file name. Filename specified in parameter FileSpec_[1-N] is too long. Change the name to a proper file name.
System Error 11011	Duplicate file name specification. Filename specified in parameter FileSpec_[1-N] is not unique. Change the name to a proper file name.

Error code	**Description**
System Error 11012	License information not found, exiting from SOLID *SynchroNet* Check the existence of your `solid.lic` file.
System Error 11013	License information is corrupted. Your `solid.lic` file has been corrupted.
System Error 11014	Database age limit of evaluation license expired.
System Error 11015	Evaluation license expired.
System Error 11016	License is for different CPU architecture.
System Error 11017	License is for different OS environment.
System Error 11018	License is for different version of this OS.
System Error 11019	License is not valid for this server version.
System Error 11020	License information is corrupted.
System Error 11021	Problem with Your license, please contact Solid Information Technology Ltd. immediately.
System Error 11022	Desktop license is only for local *protocol* communication, cannot use protocol *protocol* for listening.
System Error 11024	Desktop license is only for local communication, cannot use name *name* for listening.
System Error 11025	License file *filename* is not compatible with this server executable. Server has been started with an incompatible license file. You need to update your license file to match the server version.
System Error 11026	Backup directory contains a file which could not be removed. Some file could not be removed from the backup directory. The backup directory may point to a wrong location.
System Error 11027	No such parameter section *section*. Parameter was not found from the specified section in the `solid.ini` file.
System Error 11028	No such parameter *section.name*. Parameter does not exist.

Error code	Description
11029	Not allowed to set parameter value. User is not allowed to set the parameter value.
11030	Cannot set values to multiple parameters. Only one parameter can be set at one time.
11031	Illegal type for parameter. Parameter type is illegal.
11032	Cannot set new value for parameter *section.name*. A new value cannot be set for the parameter.

SOLID Table Errors

Error code	Description
Table Error 13001	Illegal character constant *constant.* An illegal character constant was found in the SQL statement.
Table Error 13002	Type CHAR not allowed for arithmetic. You have entered a calculation having a character type constant. Character constants are not supported in arithmetical.
Table Error 13003	Aggregate function *function* not available for ordinary call. Aggregate functions can not be used for ordinary function calls.
Table Error 13004	Illegal aggregate function *parameter* parameter. An illegal parameter has been given to an aggregate function. Aggregate function parameters can only be column names or numbers.
Table Error 13005	SUM and AVG not supported for CHAR type. Aggregate functions SUM and AVG are not supported for character type parameters.

Error code	Description
Table Error 13006	SUM or AVG not supported for DATE type. Aggregate functions SUM and AVG are not supported for date type parameters.
Table Error 13007	Function *function* is not defined. The function you tried to use is not defined.
Table Error 13009	Division by zero. A division by zero has occurred.
Table Error 13011	Table *table* does not exist. You have referenced a table which does not exist or you do not have REFERENCES privilege on the table.
Table Error 13013	Table name *table* conflicts with an existing entity. Choose a unique name for a table. The specified name is already used.
Table Error 13014	Index *index* does not exist. You have referenced an index which does not exist.
Table Error 13015	Column *column* does not exist on table *table.* You have referenced a column in a table which does not exist.
Table Error 13016	User does not exist. You have referenced a user which does not exist.
Table Error 13018	Join table is not supported Joined tables are not supported in this version of SOLID *SynchroNet*.
Table Error 13019	Transaction savepoints are not supported. Transaction savepoints are not supported in this version of SOLID *SynchroNet*.
Table Error 13020	Default values are not supported. Default column values are not supported in this version of SOLID *SynchroNet*.

Error code	**Description**
Table Error 13021	Foreign keys are not supported. Foreign keys are not supported in this version of SOLID *SynchroNet*.
Table Error 13022	Descending keys are not supported. Descending keys are not supported in this version of SOLID *SynchroNet*.
Table Error 13023	Schema is not supported. Schema is not supported in this version of SOLID *SynchroNet*.
Table Error 13025	Update through a cursor with no current row. You have tried to update using cursor, but you do not have current row in the cursor.
Table Error 13026	Delete through a cursor with no current row You have tried to delete using cursor, but you do not have current row in the cursor.
Table Error 13028	View *view* does not exist. You have referenced a view which does not exist.
Table Error 13029	View name *view* conflicts with an existing entity. Choose a unique name for a view. The specified name is already used.
Table Error 13030	No value specified for NOT NULL column *column*. You have not specified a value for a column which is defined NOT NULL.
Table Error 13031	Data dictionary operation is active for accessed table or key. You can not access the table or key, because a data dictionary operation is currently active. Try again after the data dictionary operation has completed.
Table Error 13032	Illegal type *type*. You have tried to create a table with a column having an illegal type.

Error code	Description
Table Error 13033	Illegal parameter *parameter* for type *type.* The type of the parameter you entered is illegal in this column.
Table Error 13034	Illegal constant *constant.* You have entered an illegal constant.
Table Error 13035	Illegal INTEGER constant *constant.* You have entered an illegal integer type constant. Check the syntax of the statement and try again.
Table Error 13036	Illegal DECIMAL constant *constant.* You have entered an illegal decimal type constant. Check the decimal number and try again.
Table Error 13037	Illegal DOUBLE PREC constant *constant.* You have entered an illegal double precision type constant. Check the number and try again.
Table Error 13038	Illegal REAL constant *constant.* You have entered an illegal real type constant. Check the real number and try again.
Table Error 13039	Illegal assignment. You have tried to assign an illegal value for a column.
Table Error 13040	Aggregate *function* function is not defined. The aggregate function you tried to use is not supported.
Table Error 13041	Type DATE not allowed for arithmetic. DATE type columns or constants are not allowed in arithmetical.
Table Error 13042	Power arithmetic not allowed for NUMERIC and DECIMAL data type. Decimal and numeric data types do not support power arithmetical.
Table Error 13043	Illegal date constant *constant.* A date constant is illegal. The correct form for date constants is: YYYY-MM-DD.

Error code	**Description**
Table Error 13045	Reference privileges are not supported. Reference privileges are not supported in this version of SOLID *SynchroNet*.
Table Error 13046	Illegal user name *user.* User name entered is not legal. A legal user name is at least 2 and at most 31 characters in length. A user name may contain characters from A to Z, numbers from 0 to 9 and underscore character '_'.
Table Error 13047	No privileges for operation. You have no privileges for the attempted operation.
Table Error 13048	No grant option privilege for entity *name.* You have no privileges to grant privileges for the entity.
Table Error 13049	Column privileges cannot be granted WITH GRANT OPTION Granting column privileges WITH GRANT OPTION is not supported in this version of SOLID *SynchroNet*.
Table Error 13050	Too long constraint value. Maximum constraint length has been exceeded. Maximum constraint length is 255 characters.
Table Error 13051	Illegal column name *column.* You have tried to create a table with an illegal column name.
Table Error 13052	Illegal comparison operator *operator* for a pseudo column *column.* You have tried to use an illegal comparison operator for a pseudo column. Legal comparison operators for pseudo columns are: equality '=' and non-equality '<>'.
Table Error 13053	Illegal data type for a pseudo column. You have tried to use an illegal data type for a pseudo column. Data type of pseudo columns is BINARY.

Error code	Description
Table Error 13054	Illegal pseudo column data, maybe data is not received using pseudo column. You have tried to compare pseudo column data with non-pseudo column data. Pseudo column data can only be compared with data received from a pseudo column.
Table Error 13055	Update not allowed on pseudo column. Updates are not allowed on pseudo columns.
Table Error 13056	Insert not allowed on pseudo column. Inserts are not allowed on pseudo columns.
Table Error 13057	Index name *index* already exists. You have tried to create an index, but an index with the same name already exists. Use another name for the index.
Table Error 13058	Constraint checks were not satisfied on column *column.* Column has constraint checks which were not satisfied during an insert or update.
Table Error 13059	Reserved system name *name.* You tried to use a name which is a reserved system name such as PUBLIC and SYS_ADMIN_ROLE.
Table Error 13060	User name *user* not found. You tried to reference a user name which is not created.
Table Error 13061	Role name *role* not found. You tried to reference a role name which is not created.
Table Error 13062	Admin option is not supported. Admin option is not supported in this version of SOLID *SynchroNet*.
Table Error 13063	Name *name* already exists. You tried to use a role or user which already exists. User names and role names must all be different, that is, you can not have a user named HOBBES and a role named HOBBES.

Error code	**Description**
Table Error 13064	Not a valid user name *user.* You tried to create an invalid user name. A valid user name has at least 2 characters and at most 31 characters.
Table Error 13065	Not a valid role name *role.* You tried to create an invalid role name. A valid user name has at least 2 characters and at most 31 characters.
Table Error 13066	User *user* not found in role *role.* You tried to revoke a role from a user and the user did not have that role.
Table Error 13067	Too short password. You have entered a too short password. Password length must be at least 3 characters.
Table Error 13068	Shutdown is in progress. You are unable to complete this operation, because server shutdown is in progress.
Table Error 13070	Numerical overflow. A numerical overflow has occurred. Check the values and types of numerical variables.
Table Error 13071	Numerical underflow. A numerical underflow has occurred. Check the values and types of numerical variables.
Table Error 13072	Numerical value out of range. A numerical value is out of range. Check the values and types of numerical variables.
Table Error 13073	Math error. A mathematical error has occurred. Check the mathematics in the statement and try again.
Table Error 13074	Illegal password. You have tried to enter an illegal password.

Error code	Description
Table Error 13075	Illegal role name *role.* You have tried to enter an illegal role name. A legal role name is at least 2 and at most 31 characters in length. A user role may contain characters from A to Z, numbers from 0 to 9 and underscore character '_'.
Table Error 13076	NOT NULL must not be specified for added column *column.* You have tried to add a column to a table using ALTER TABLE statement. NOT NULL constraint is not allowed in ALTER TABLE statement when the table already includes data.
Table Error 13077	Last column can not be dropped. You have tried to drop the final column in a table. This is not allowed; at least one column must remain in the table.
Table Error 13078	Column already exist on table. You have tried to create a column which already exists in a table.
Table Error 13079	Illegal search constraint. Check the search engine. There may be mismatch between data types.
Table Error 13080	Incompatible types, can not modify column *column* from type *type* to type *type.* You have tried to modify column to a data type that is incompatible with the original definition, such as VARCHAR and INTEGER
Table Error 13081	Descending keys are not supported for binary columns. You can not define descending key for a binary column.
Table Error 13082	Function *function*: parameter * not supported. You can not use parameter star (*) with ODBC Scalar Functions.
Table Error 13083	Function *function*: Too few parameters. The function expects more parameters. Check the function call.

Error code	Description
Table Error 13084	Function *function*: Too many parameters. The function expects fewer parameters. Check the function call.
Table Error 13085	Function *function*: Run-time failure. An error was detected during the execution of the function. Check the parameters.
Table Error 13086	Function *function*: type mismatch in parameter *parameter number.* A erroneous type of parameter detected in the given position of the function call. Check the function call.
Table Error 13087	Function *function*: illegal value in parameter *parameter number.* An illegal value for a parameter detected in the given position of the function call. Check the function call.
Table Error 13090	Foreign key column *column* data type not compatible with referenced column data type. References specification error. Check that the column data type are compatible between referencing and referenced tables.
Table Error 13091	Foreign key does not match to the primary key or unique constraint of the referenced table. References specification error. Check that the column data type are compatible between referencing and referenced tables and that the foreign key is unique for the referenced table.
Table Error 13092	Event name *event* conflicts with an existing entity. Choose a unique name for an event. The specified name is already used.
Table Error 13093	Event *event* does not exist. You referenced to a nonexistent event. Check the name of event.
Table Error 13094	Duplicate column *column* in primary key definition. Duplicate columns are not allowed in a table-constraint-definition. Remove duplicate columns from the definition.

Error code	Description
Table Error 13095	Duplicate column *column* in unique constraint definition. Duplicate columns are not allowed in a table-constraint-definition. Remove duplicate columns from the definition.
Table Error 13096	Duplicate column *column* in index definition. Duplicate columns are not allowed in CREATE INDEX statement. Remove duplicate columns.
Table Error 13097	Primary key columns must be NOT NULL. Error in a *column_constraint_definition*. Define primary key columns NOT NULL. For example: CREATE TABLE DEPT (DEPTNO INTEGER NOT NULL, DNAME VARCHAR, PRIMARY KEY(DEPTNO));
Table Error 13098	Unique constraint columns must be NOT NULL. Error in a *column_constraint_definition*. Define unique columns NOT NULL. For example: CREATE TABLE DEPT4 (DEPTNO INTEGER NOT NULL, DNAME VARCHAR, UNIQUE(DEPTNO));
Table Error 13099	No REFERENCES privileges to referenced columns in table *table.* You do not have privileges to reference to the table.
Table Error 13100	Illegal table mode combination. You have defined illegal combination of locking. Check locking type of tables.
Table Error 13101	Only execute privileges can be used with procedures.
Table Error 13102	Execute privileges can be used only with procedures.
Table Error 13103	Illegal grant or revoke operation.
Table Error 13104	Sequence name *sequence* conflicts with an existing entity. Choose a unique name for a sequence. The specified name is already used.
Table Error 13105	Sequence *sequence* does not exist. You referenced a nonexistent sequence. Check the name of sequence.
Table Error 13106	Foreign key reference exists to table *table.*

Error code	Description
Table Error 13107	Illegal set operation. You tried to execute a non-existent set operation.
Table Error 13108	Comparison between incompatible types *datatype* and *datatype.*
Table Error 13109	There are schema objects for this user, drop failed
Table Error 13110	NULL values given for NOT NULL column *column*.
Table Error 13111	Ambiguous entity name *name*.
Table Error 13112	Foreign keys are not supported with main memory tables.
Table Error 13113	Illegal arithmetics between types *datatype* and *datatype*.
Table Error 13114	String operations are not allowed on values stored as BLOBs or CLOBs.
Table Error 13115	Function *function_name*: Too long value (stored as CLOB) in parameter *parameter.* The parameter value was stored as CLOB and cannot be used with a function.
Table Error 13116	Column *column_name* specified more than once. Column was specified more than once in the GRANT or REVOKE statement.
Table Error 13117	Wrong number of parameters Wrong number of parameters when converting subscription parameters to base publication parameter types.
Table Error 13118	Column privileges are supported only for base tables. Column privileges are allowed only for base tables; they cannot be used, for example, for views.
Table Error 13119	Types *column_type* and *column_type* are not union compatible. Column types are not union compatible.
Table Error 13120	Too long entity name '*entity_name*' Entity name is too long, maximum entrity name is 254 characters.

Error code	Description

Table Error 13121	Too many columns, maximum number of columns is *value*.
	
	Too many columns; the default maximum number of columns is 1000. It can be changed using the the following parameter in the `solid.ini` file:
	
	`[Srv]`
	`MaxOpenCursors=n`

Table Error 13122	Operation is not supported for a table with sync history. Operation is not supported because the table has synchronization history defined.
Table Error 13123	Table '*table_name*' is not empty. Some operations are allowed only for empty tables.
Table Error 13124	User id *user_id* not found. Internal user id was not found; the user may have been dropped.
Table Error 13125	Illegal LIKE pattern '*pattern*' Illegal like pattern was given as a search constraint.
Table Error 13126	Illegal type *datatype* for LIKE pattern. Only CHAR and WCHAR allowed for LIKE search constraints.
Table Error 13127	Comparison failed because at least one of the was too long. Comparison failed because at least one of the column values was stored as a BLOB or CLOB.
Table Error 13128	LIKE predicate failed bacause value is too long. LIKE predicate failed because the column value is stored as a CLOB.
Table Error 13129	LIKE Predicate failed bacause pattern is too long. LIKE predicate failed bacause pattern value is stored as a CLOB.
Table Error 13130	Illegal type *datatyp*e for LIKE ESCAPE character. Like ESCAPE character must be CHAR or WCHAR type.

Error code	**Description**
Table Error 13131	Too many nested triggers. Maximum number of nested triggers is reached. Triggers may be nested, for example, by activating other triggers from a trigger or causing recursive cycle when activating triggers. Default value for maximum allowed nested triggers is 16. It can be changed using a configuration parameter: `{SQL]` `MaxNestedTriggers=`*n*
Table Error 13132	Too many nested procedures. Maximum number of nested procedures is reached. Procedures may be nested, for example, by activating other procedures from a procedure or causing a recursive cycle when activating procedures. Default value for maximu allowed nested procedures is 16. It can be changed using a configuration parameter: `SQL` `MaxNestedTriggers=`*n*
Table Error 13133	Not a valid license for this product. The license file is for another SOLID product.
Table Error 13134	Operation is allowed only for base tables. Given operation is available only for base tables.
Table Error 13137	Illegal grant/revoke mode Grant or revoke mode is not allowed for given database objects.
Table Error 13138	Index *index_name* given in index hint does not exist. Index name given in optimizer hint is not found for a table.
Table Error 13139	Catalog *catalog_name* does not exist. Catalog name is not a valid catalog.
Table Error 13140	Catalog *catalog_name* already exists. Catalog name is an existing catalog.

Error code	Description
Table Error 13141	Schema *schema_name* does not exist. Schema name is not a valid schema.
Table Error 13142	Schema *schema_name* already exists. Schema name is an existing schema.
Table Error 13143	Schema *schema_name* is an existing user. Schema name specifies an existing user name.
Table Error 13144	Commit and rollback are not allowed inside trigger. Commit or rollback are not supported inside trigger execution. This error is also given if a trigger calls a procedure that tries to execute commit or rollback command.
Table Error 13145	Sync parameter not found. Parameter name given in command SET SYNC PARAMETER *name* NONE is not found.
Table Error 13146	There are schema objects for this catalog, drop failed. Catalog contains schema object and cannot be dropped. Schema objects like tables and procedures need to be dropped before catalog can be dropped.
Table Error 13151	Cannot drop a column that is part of primary or unique key. Table definition contains a column that is part of a primary or unique key in an index.

SOLID Server Errors

Error code	Description
Server Error 14501	Operation failed. This error occurs when a timed command fails. Check the arguments of timed commands.
Server Error 14502	RPC parameter is invalid. A network error has occurred.

Error code	**Description**
Server Error 14503	Communication error. A communication error has occurred.
Server Error 14504	Duplicate cursor name *cursor*. You have tried to declare a cursor with a cursor name which is already in use. Use another name.
Server Error 14505	Connect failed, illegal user name or password. You have entered either a user name or a password that is not valid.
Server Error 14506	Server is closed, no new connections allowed. You have tried to connect to a closed server. Connecting was aborted.
Server Error 14507	Maximum number of licensed user connections exceeded. You have tried to connect to a server which has all licenses currently in use. Connecting was aborted.
Server Error 14508	The operation has timed out. You have launched an operation that has been aborted.
Server Error 14509	Version mismatch. A version mismatch has occurred. The client and server are different versions. Use same versions in the client and the server.
Server Error 14510	Communication write operation failed. A write operation failed. This indicates a network problem. Check your network settings.
Server Error 14511	Communication read operation failed. A read operation failed. This indicates a network problem. Check your network settings.
Server Error 14512	There are users logged to the server. You can not shutdown the server now. There are users connected to the server.
Server Error 14513	Backup process is active. You can not shutdown the server now. The backup process is active

Error code	Description
Server Error 14514	Checkpoint creation is active. You can not shutdown the server now. The checkpoint creation is active.
Server Error 14515	Invalid user id. You tried to drop a user, but the user id is not logged in to the server.
Server Error 14516	Invalid user name. You tried to drop a user, but the user name is not logged in to the server.
Server Error 14517	Someone has updated the at commands at the same time, changes not saved. You tried to update timed commands at the same time another user was doing the same. Your changes will not be saved.
Server Error 14518	Connection to the server is broken, connection lost. Possible network error. Reconnect to the server.
Server Error 14519	The user was thrown out from the server, connection lost. Possible network error.
Server Error 14521	Failed to create a new thread for the client.
Server Error 14529	The operation timed out.
Server Error 14530	The connected client does not support UNICODE data types. Connected client is an old version client that does not support UNICODE data types. UNICODE data type columns cannot be used with old clients.

Server Error 14531	Too many open cursor, max limit is *value*.
	
	There are too many open cursors for one client; maximum number of open cursors for one connection is 1000. The value can be changed using a configuration value:
	
	`[Srv]`
	`MaxOpenCursors=n`

Error code	Description
Server Error 14533	Operation cancelled Operation was cancelled because client application called ODBC or JDBC cancel function.
Server Error 14534	Only administrative statements are allowed. Only administrative statements are allowed for the connection.

SOLID Communication Errors

Error code	Description
Communication Error 21300	Protocol *protocol* is not supported. Protocol is not supported.
Communication Error 21301	Cannot load the dynamic link library <library> or one of its components. The server was unable to load the dynamic link library or a component needed by this library. Check the existence of necessary libraries and components.
Communication Error 21302	Wrong version of dynamic link library *library.* The version of this library is wrong. Update this library to a newer version.
Communication Error 21303	Network adapter card is missing or needed <protocol> software is not running. The network adapter card is missing or not functioning.
Communication Error 21304	Out of *protocol* resources The network protocol is out of resources. Increase the protocols resources in the operating system.
Communication Error 21305	An empty or incomplete network name was specified. The network name specified is not legal. Check the network name.

Error code	Description
Communication Error 21306	Server *network name* not found, connection failed. The server was not found. 1) Check that the server is running. 2) Check that the network name is valid. 3) Check that the server is listening given network name.
Communication Error 21307	Invalid connect info *network name.* The network name given as the connect info is not legal. Check the network name.
Communication Error 21308	Connection is broken (*protocol read/write* operation failed with code *internal code*). The connection using the protocol is broken. Either a read or a write operation has failed with an internal error *internal code*.
Communication Error 21309	Failed to accept a new client connection, out of *protocol* resources. The server was not able to establish a new client connection. The protocol is out of resources. Increase the protocol's resources in the operating system.
Communication Error 21310	Failed to accept a new client connection, listening of *network name* interrupted. The server was not able to establish a new client connection. The listening has been interrupted.
Communication Error 21311	Failed to start a selecting thread for *network name.* A thread selection has failed for *network name*.
Communication Error 21312	Listening info *network name* already specified for this server. A network name has already been specified for this server. A server can not use a same network name more than once.
Communication Error 21313	Already listening with the network name *network name.* You have tried to add a network name to a server when it is already listening with that network name. A server can not use a same network name more than once.

Error code	Description
Communication Error 21314	Cannot start listening, network name *network name* is used by another process. The server can not start listening with the given network name. Another process in this computer is using the same network name.
Communication Error 21315	Cannot start listening, invalid listening info *network name.* The server can not start listening with the given listening info. The given network name is invalid. Check the syntax of the network name.
Communication Error 21316	Cannot stop the listening of *network name*. There are clients connected. You can not stop listening of this network name. There are clients connected to this server using this network name.
Communication Error 21317	Failed to save the listen information into the configuration file . The server failed to save this listening information to the configuration file. Check the file access rights and format of the configuration file.
Communication Error 21318	Operation failed because of an unusual *protocol* return code *code.* Possible network error. Create connection again.
Communication Error 21319	RPC request contained an illegal version number. Either the message was corrupted or there may be a mismatch between server and client versions.
Communication Error 21320	Called RPC service is not supported in the server. There maybe a mismatch between server and client versions.
Communication Error 21321	Protocol *protocol* is not valid, try using switch '-a' for specifying another adapter id instead of *switch*. This is returned if the NetBIOS LAN adapter id given in listen/connect string is not valid.

Error code	Description
Communication Error 21322	The host machine given in connect info '%s' was not found. This is returned in clients if the host machine name given in connect info is not valid.
Communication Error 21323	Protocol *protocol* can not be used for listening in this environment. This message is displayed if the server end communication using specified protocol is not supported.
Communication Error 21324	The process does not have the privilege to create a mailbox.
Communication Error 21325	Only one listening name is supported in this server. In some operating systems like Novell Netware only one listening name is supported.
Communication Error 21326	Failed to establish an internal *number* socket connection *code* number. SOLID uses one connect socket for internal use. Creation of this socket has failed; the local loopback may not be working correctly.

SOLID Communication Warnings

Error code	Description
Warning Code 21100	Illegal value *value* for configuration parameter *parameter*, using default. An illegal value was given to the parameter *parameter*. The server will use a default value for this parameter.
Warning Code 21101	Invalid protocol definition *protocol* in configuration file. The protocol is defined illegally in the configuration file. Check the syntax of the definition.

SOLID Procedure Errors

Error code	Description
Procedure Error 23001	Undefined symbol *symbol.* You have used a symbol that has not been defined in a procedure definition.
Procedure Error 23002	Undefined cursor *cursor.* You have used a cursor that has not been defined in a procedure definition.
Procedure Error 23003	Illegal SQL operation *operation*.
Procedure Error 23004	Syntax error: parse error, line *line number.* Check the syntax of your procedure.
Procedure Error 23005	Procedure *procedure* not found.
Procedure Error 23006	Wrong number of parameters for procedure *procedure.*
Procedure Error 23007	Procedure name *value* conflicts with an existing entity. Choose a unique name for a procedure. The specified name is already used.
Procedure Error 23009	Event *event* does not exist, line *line number.*
Procedure Error 23010	Incompatible event *event* parameter type, line *line number.*

Error code	Description
Procedure Error 23011	Wrong number of parameter for event *event*, line *line number.*
Procedure Error 23012	Duplicate wait for event *event*, line *line number.*
Procedure Error 23013	Undefined sequence *sequence.*
Procedure Error 23014	Duplicate sequence name *sequence.*
Procedure Error 23015	Sequence *sequence* not found.
Procedure Error 23016	Incompatible variable type in call to sequence *sequence*, line *line number.*
Procedure Error 23017	Duplicate symbol *symbol.* You have duplicate definitions for a symbol.
Procedure Error 23018	Procedure owner *owner* not found.
Procedure Error 23019	Duplicate cursor name '*cursor'*
Procedure Error 23020	Illegal option *option* for WHENEVER SQLERROR … statement.
Procedure Error 23021	RETURN ROW not allowed in procedure with no return type, line *line number.*
Procedure Error 23022	SQL String variable *variable* must be of character data type, line *line number.*
Procedure Error 23023	Call syntax error: *syntax*, line *line number.*
Procedure Error 23024	Trigger *trigger_name* not found. Trigger name not found.
Procedure Error 23025	Trigger name *trigger_name* conflicts with an existing entity. Trigger name conflicts with some other database object. Triggers share the same name space, as for example, in table and procedures.
Procedure Error 23026	Variable *variable* is ot of character type, line *line number.* A CHAR or WCHAR variable is required for the operations like RETURN SQLERROR *variable*.

Error code	**Description**
Procedure Error 23027	Duplicate reference to column *column_name* in trigger definition. One column can be reference only once in the trigger definition.
Procedure Error 23028	Commit and rollback are not allowed in triggers. Trigger body may not contain commit or rollback statements.
Procedure Error 23501	Cursor *cursor* is not open.
Procedure Error 23502	Illegal number of columns in EXECUTE ... *procedure* in cursor *cursor.*
Procedure Error 23503	Previous SQL operation *operation* failed in cursor *cursor.*
Procedure Error 23504	Cursor *cursor* is not executed.
Procedure Error 23505	Cursor *cursor* is not a SELECT statement.
Procedure Error 23506	End of table in cursor *cursor.*
Procedure Error 23507	Illegal type conversion in cursor *cursor* from type *data type* to type *data type.*
Procedure Error 23508	Illegal assignment, line *line number.*
Procedure Error 23509	In *procedure* line *line number* Stmt *statement* was not in error state in RETURN SQLERROR OF ...
Procedure Error 23510	In *procedure* line *line number* Transaction cannot be set read only, because it has written already.
Procedure Error 23511	In *procedure* line *line number* USING part is missing for dynamic parameters for *procedure.*
Procedure Error 23512	In *procedure* line *line number* USING list is too short for *procedure.*
Procedure Error 23513	In *procedure* line *line number* Comparison between incompatible types *data type* and *data type.*
Procedure Error 23514	In *procedure* line *line number* type *data type* is illegal for logical expression.
Procedure Error 23515	In *procedure* line *line number* assignment of parameter *parameter* in *list* list failed.

Error code	Description
Procedure Error 23516	In CALL *procedure* assignment of parameter *parameter* failed.
Procedure Error 23518	User error: *error_text* User generated error in a procedure or trigger. User can generate this error by using a statement RETURN SQLERROR *string* or RETURN SQLERROR *variable*. Variable must be of CHAR or WCHAR type.
Procedure Error 23519	Fetch previous is not supported for procedures. Fetch previous row does not work for result sets returned by a procedure.

SOLID Sorter Errors

Error code	Description
Sorter Error 24001	Sort failed due to insufficient configured TmpDir space
Sorter Error 24002	Sort failed due to insufficient physical TmpDir space
Sorter Error 24003	Sort failed due to insufficient sort buffer space
Sorter Error 24004	Sort failed due to too long row (internal failure)
Sorter Error 24005	Sort failed due to I/O error

C

Configuration Parameters

By managing the parameters of your SOLID *SynchroNet*, you can modify the environment, performance, and operation of the server.

When SOLID *SynchroNet* is started, it attempts to open the configuration file `solid.ini` in the current directory. The configuration values for the server parameters are included in this file. If the file does not exist, SOLID *SynchroNet* will use the default settings for the parameters. Also, if a value for a parameter is not set in the `solid.ini` file, SOLID *SynchroNet* will use a default value for the parameter. The default values depend on the operating system you are using.

Generally, the default settings offer the best performance and operability, but in some special cases modifying a parameter will improve performance. You can change the parameters in the following ways:

- Using the SOLID *DBConsole* Configuration page.
- Entering the command `parameter` in SOLID *DBConsole* (Query window or command line) or SOLID *Remote Control* (teletype).
- Entering `ADMIN COMMAND 'parameter'` in SOLID *SQL Editor* (teletype).
- Manually editing the configuration file `solid.ini`.

General Section

[General]	Description	Default
MaxOpenFiles	the maximum number of files kept concurrently open during SOLID *SynchroNet* sessions	OS depend.
BackupDirectory	makes a backup of the database if the default ’`backup`’ is used or may also be given as an argument. For example, backup abc, creates a backup on directory ’`abc`’. All directory definitions are relative to the SOLID *SynchroNet* working directory unless the full path is provided.	’backup’ directory
BackupCopyLog	if set to `yes`, backup operation will copy log files to the backup directory	yes
BackupDeleteLog	if set to `yes`, old log files will be deleted after backup operation	yes
BackupCopyIniFile	if set to `yes`, `solid.ini` file will be copied to the backup directory	yes
BackupCopySolmsgout	If set to yes, `solmsg.out` file is copied to the backup directory	yes
Checkpoint Interval	the number of inserts made in the database that causes automatic checkpoint creation	5000
MergeInterval	the number of index inserts made in the database that causes the merge process to start	Cache size depend.
Readonly	if set to `yes`, database is set to read-only mode	no
LongSequential SearchLimit	the number of sequential fetches after which search is treated as long sequential search	500
SearchBuffer Limit	the maximum percentage of search buffers from the total buffered memory reserved for open cursors	50
Transaction HashSize	the hash table size for incomplete transactions	Cache size depend.

TableLockWaitTimeout	the time in seconds that a transaction waits to get a lock. When messages are executed in the replica, it is possible to run them in pessimistic or mixed concurrency mode, which means table level locks are used. There are times when a transaction will acquire an exclusive lock to a table. If there is a conflict, this setting provides the transaction's wait period until the exclusive or shared lock is released. This parameter is used for synchronized databases only. Table level locks are used when the PESSIMISTIC keyword is explicitly provided in the following *SynchroNet* commands: IMPORT SUBSCRIPTION MESSAGE *message_name* EXECUTE (but not with NO EXECUTE option) MESSAGE *message_name* FORWARD MESSAGE *message_name* GET REPLY DROP SUBSCRIPTION	30

IndexFile Section

[IndexFile]	Description	Default
FileSpec_[1-N]	the file name followed with maximum size (in bytes) of that database file, for example: `c:\sol1.db 2000000` This parameter also has an optional parameter after the maxsize: physical drive number. The number value itself is not essential, but it is used as a hint for I/O threads on which I/O requests can be parallelized. This file must be stored to a local drive using local disk names to avoid problems with network I/O and to achieve better performance.	solid.db 2147483647
BlockSize	the block size of the index file in bytes; use multiple of 2 KB: minimum 2 KB, maximum 32 KB	8192
CacheSize	the size of database cache memory for the server in bytes; the minimum 512 KB	OS depend.
ExtendIncrement	the number of blocks that is allocated at one time when SOLID *SynchroNet* needs to allocate more space for the database file	50
ReadAhead	sets the number of prefetched index leafs during long sequential searches	4
PreFlushPercent	Percentage of page buffer which is kept clean by preflush thread	5

Logging Section

[Logging]	Description	Default
LogEnabled	whether logging is enabled or not	yes
BlockSize	the block size of log files	2048
MinSplitSize	when this file size is reached, logging will be continued to the following log file after the next checkpoint	1 MB
FileNameTemplate	the path and naming convention used when creating log files; template characters are replaced with sequential numbering; for example: `c:\solid\log\sol#####.log` This file must be stored to a local drive using local disk names to avoid problems with network I/O and to achieve better performance.	sol#####.log
DigitTemplate Char	the template character that will be replaced in the name template of the log file	#

Communication Section

[Com]	Description	Default
Listen	the network name for the server; the protocol and server-name that SOLID *SynchroNet* uses when it starts listening to the network	OS depend.
Connect	the network name for client; the protocol and name that a SOLID *SynchroNet* client uses for server connection; in a Windows environment ODBC Data Source Name overrides the value of this parameter	OS depend.
MaxPhysMsgLen	the maximum length of a single physical network message in bytes; longer network messages will be split into smaller messages of this size	OS depend.
ReadBufSize	the buffer size in bytes for the data read from the network	OS depend.
WriteBufSize	the buffer size in bytes for the data written into the network	OS depend.
Trace	if this parameter is set to yes, trace information on network messages is written to a file specified with the TraceFile parameter	no
TraceFile	if this parameter is set to **yes**, trace information on network messages is written to a file specified with this parameter	`soltrace.out` (written to the current working directory of the server or client depending on which end the tracing is started)

Data Sources

[Data Sources]	Description	Default
<logical name> = <network name>, <Description>	These parameters can be used to give a logical name to a SOLID *SynchroNet.*	

Server Section

[Srv]	Description	Default
At	The syntax is: *At* := *timed_command* [, *timed_command*] *timed_command* := [*day*] *HH:MM command argument* *day* := **sun | mon | tue | wed | thu | fri | sat** If entered, allows you specify a command to automate an administrative task, such as executing system commands, creating backups, checkpoints, and database status reports. For example: AT=20:30 makecp, 21:00 backup,sun 23:00 shutdown	If you specify backup, the default is the backup directory set with the BackupDirectory parameter in the General section. Also if the day is not given, the command is executed daily.
RowsPerMessage	the number of rows returned from the server in one network message	10
ConnectTimeOut	specifies the continuous idle time in minutes after that an connection is dropped; negative or zero value means infinite	480
AbortTimeOut	specifies the time in minutes after that an idle transaction is aborted; negative or zero value means infinite	120
Threads	the number of threads used for database access in SOLID *SynchroNet*.	OS depend.
Echo	if set to `yes,` contents of `solmsg.out` file are displayed also at the server's command window	no
Name	the informal name of the server, equivalent to the -n command line option	
AllowConnect	if set to no only connections from *Remote Control* or *DBConsole* are allowed	yes
MessageLogSize	The maximum size of the solmsg.out file in bytes. The deafult is 60 KB.	OS depend.
MaxOpenCursors	The maximum number of cursors that a database client can have simultaneously open.	1000

SQL Section

[SQL]	Description	Default
Info	Set the level of informational messages [0-8] printed from the server (0=no info, 8=all info); information is written into the file defined by parameter InfoFileName.	0
SQLInfo	Set the level of informational SQL level messages [0-8] (0=no info, 8=all info); information is written into a file defined by parameter InfoFileName.	no default
InfoFileName	Default global info file name.	`soltrace.out`
InfoFileSize	Maximum size of the info file. The default is 1 MB.	no default
InfoFileFlush	If set to yes, flushes info file after every write operation	yes
SortArraySize	Size of the array that SQL uses when ordering result set; for optimal performance this should be as big as the biggest retrieved result set that cannot be ordered by key values; for large sorts use external sorter,	OS depend.
ProcedureCache	Size of cache memory for parsed procedures in number of procedures.	5
MaxNestedProcedures	Maximum number of allowed nested procedures. If this parameter is defined too high, the server stack may become insufficient depending on the operating system.	16
TriggerCache	Size of cache memory that each user has for triggers.	10
MaxNestedTriggers	Maximum number of allowed nested triggers. This maximum number includes both direct and indirect nesting, so both A->A->A and A->B->A are counted as three nested triggers.	16
MaxBlobExpression Size	Maximum size of LONG VARCHAR columns in KBs that can be used in string functions.	64
EmulateOldTIMESTAMPDIFF	If included in the `solid.ini` file and set to "Yes", the old TIMESTAMPDIFF behavior is emulated by the server. This old behavior returns the integer number of intervals of type *interval* by which *timestamp_exp2* is greater than *timestamp_exp1*. Otherwise, the default is the new behavior which returns the integer number of *interval* as the amount of full units between *timestamp_exp1* and *timestamp_exp2*.	The default "No" returns the integer number of *interval* as the amount of full units between *timestamp_exp1* and *timestamp_exp2*

Sorter Section

[Sorter]	Description	Default
MaxCacheUse Percent	maximum percentage of cache pages used for sorting; range from 10% to 50%	
MaxMemPerSort	maximum memory available in bytes for one sort	
MaxFilesTotal	maximum number of files used for sorting	
TmpDir_[1-N]	name of the directory that contains temporary files created during sorting	no default

Hints Section

[Hints]	Description	Default
EnableHints	If included in the `solid.ini` file and set to "Yes", all hints that are in the `solid.ini` file are enabled.	Yes

Synchronizer Section

[Synchronizer]	Description	Default
NeedPublicationRegistration	If included in the `solid.ini` file and set to "Yes", disables implicit registration for all publications. This means you must explicitly register publications for user subscriptions using the MESSAGE APPEND REGISTER PUBLICATION command. If set to “No,” explicit registration is not required and implicit registration is enabled. Setting the value to “No” is used for backward compatibility with *SynchroNet* versions prior to 2.0. When setting the value to “No,” note that the statement DROP SUBSCRIPTION also drops the publication registration if the last subscription to the publication was dropped.	Yes

D

Data Types

Supported Data Types

The tables in this appendix list the supported data types by category. the following abbreviations are used in each table.

Abbreviation	Description
DEFLEN	the defined length of the column; for example, for CHAR(24) the precision and length is 24

DEFPREC	the defined precision;
	for example, for NUMERIC(10,3) it is 10

DEFSCALE	the defined scale;
	for example, for NUMERIC(10,3), it is 3

MAXLEN	the maximum length of column
N/A	not applicable

Character Data Types

Data type	Size	Precision	Scale	Length	Display size
CHAR, WCHAR	2 G*	DEFLEN	N/A	DEFLEN	DEFLEN
VARCHAR, WVARCHAR	2 G**	DEFLEN	N/A	DEFLEN	DEFLEN
LONG VAR-CHAR, LONG WVARCHAR	2 G	MAXLEN	N/A	MAXLEN	MAXLEN
* default is 1 ** default is 254					

Numeric Data Types

Data type	Range	Precision	Scale	Length	Display size
DECIMAL	±3.6e16	16	DEFSCALE	18	18
NUMERIC	±3.6e16	DEFPREC	DEFSCALE	DEFPREC +2	DEFPREC +2
TINYINT	[-128, 127] [0, 255]	3	0	1 (bytes)	4 (signed) 3 (unsigned)
SMALLINT	[-32768, 32767] [0, 65535]	5	0	2 (bytes)	6 (signed) 5 (unsigned)
INTEGER	$[-2^{31}, 2^{31-1}]$ $[0, 2^{32-1}]$	10	0	4 (bytes)	11 (signed) 10 (unsigned)
REAL	±1.7014117 e38	7	N/A	4 (bytes)	13
FLOAT	±8.9884657 e307	15	N/A	8 (bytes)	22
DOUBLE PRECISION	±8.9884657 e307	15	N/A	8 (bytes)	22

Binary Data Types

Data type	Size	Precision	Scale	Length	Display size
BINARY	2 G*	DEFLEN	N/A	DEFLEN	DEFLEN x 2
VARBINARY	2 G**	DEFLEN	N/A	DEFLEN	DEFLEN x 2
LONG VAR-BINARY	2 G	MAXLEN	N/A	MAXLEN	MAXLEN x 2
* default is 1 ** default is 254					

Date Data Type

Data type	Range	Precision	Scale	Length	Display size
DATE	N/A	10*	N/A	6**	10*
* the number of characters in the yyyy-mm-dd format ** the size of the DATE_STRUCT structure					

Time Data Type

Data type	Range	Precision	Scale	Length	Display size
TIME	N/A	8*	N/A	6**	8*
* the number of characters in the hh:mm:ss format ** the size of the TIME_STRUCT structure					

Timestamp Data Type

Data type	Range	Precision	Scale	Length	Display size
TIMESTAMP	N/A	19*	9	16**	19/29***
* the number of characters in the 'yyyy-mm-dd hh:mm:ss.fffffffff' format ** the size of the TIMESTAMP_STRUCT structure *** size is 29 with a decimal fraction part					

The Smallest Possible Non-zero Numbers

Data type	Value
DOUBLE	2.2250738585072014e-308
REAL	1.175494351e-38

Description of Different Column Values in the Tables

The range of a numeric column refers to the minimum and maximum values the column can store. The size of character columns refers to the maximum length of data that can be stored in the column of that data type.

The precision of a numeric column refers to the maximum number of digits used by the data type of the column. The precision of a non-numeric column refers to the defined length of the column.

The scale of a numeric column refers to the maximum number of digits to the right of the decimal point. Note that for the approximate floating point number columns, the scale is undefined, since the number of digits to the right of the decimal point is not fixed.

The length of a column is the maximum number of bytes returned to the application when data is transferred to its default C type. For character data, the length does not include the null termination byte. Note that the length of a column may differ from the number of bytes needed to store the data on the data source.

The display size of a column is the maximum number of bytes needed to display data in character form.

E

SOLID SQL Syntax

SOLID *SynchroNet* SQL syntax is based on the ANSI X3H2-1989 level 2 standard including important ANSI X3H2-1992 (SQL2) extensions. User and role management services missing from previous standards are based on the ANSI SQL3 draft.

This appendix presents a simplified description of the SQL statements including some examples. The same information is included in the **SOLID Programmer Guide.**

ADMIN COMMAND

ADMIN COMMAND '*command_name*'

command_name ::= BACKUP | BACKUPLIST | CLOSE | DESCRIBE PARAMETER

| ERRORCODE | EXIT | HELP | INFO | MAKECP | MESSAGES

| SHUTDOWN | MONITOR | NOTIFY | OPEN | PARAMETERS | PERFMON

| PID | PROTOCOLS | REPORT | SHUTDOWN | STATUS

| STATUS BACKUP | THROWOUT | TRACE | USERID | USERLIST

| VERSION

Usage

This SQL extension executes administrator commands. The *command_name* in the syntax is a SOLID *DBConsole* or SOLID *Remote Control* (teletype) command string (without the quotes); for example, `backup`.

If you are entering these commands in the SOLID *SQL Editor* (teletype), be sure to use the full SOLID SQL syntax; for example, `ADMIN COMMAND ’backup’`. Abbreviations for ADMIN COMMANDs are also available; for example, `ADMIN COMMAND ’bak’.` To access a list of abbreviated commands, execute `ADMIN COMMAND 'help'`.

The result set contains two columns: RC INTEGER and TEXT VARCHAR(254). Integer column RC is a command return code (0 if success), varchar column TEXT is the command reply. The TEXT field contains the same lines that are displayed on SOLID *DBConsole* screen, one line per one result row.

Note that all options of the ADMIN COMMAND are not transactional and cannot be rolled back.

Following is a description of the syntax for each ADMIN COMMAND command option:

Option Syntax	Description
ADMIN COMMAND 'backup [*backup_directory*]'	Makes a backup of the database. The default backup directory is the one defined in the configuration parameter **General.BackupDirectory**. The backup directory may also be given as an argument. For example, **backup abc** creates backup on directory ‘abc’. All directory definitions are relative to the SOLID *SynchroNet* working directory.
ADMIN COMMAND 'backuplist'	Displays a status list of last backups.
ADMIN COMMAND 'close'	Closes the server from new connections; no new connections are allowed.
ADMIN COMMAND 'describe parameter *param*'	Returns description text of parameter. The following example describes parameter [Com]Trace=y/n. `ADMIN COMMAND 'describe parameter com.trace'`
ADMIN COMMAND 'errorcode *SOLID_error_code'*	Displays a description of an error code. Gives the code number as an argument. For example, '**errorcode 10033**'
ADMIN COMMAND 'help'	Displays available commands.

Option Syntax	Description
ADMIN COMMAND 'info *options*'	Returns server information. *Options* are one or more of the following values, each separated by a space: ■ **Numusers** - number of current users ■ **Maxusers** - maximum number of users ■ **Sernum** - Server serial number ■ **Dbsize** - database size ■ **Logsize** - size of log files ■ **Uptime** - server up since ■ **Bcktime** - timestamp of last successfully completed backup ■ **Cptime** - timestamp of last successfully completed checkpoint ■ **tracestate** - Current trace statem ■ **monitorstate** - Current monitor state, number of users withmonitor enabled, or -1 if all ■ **openstate** - Current open or close state Values are returned in the same order as requested, one row for each value. Example: `ADMIN COMMAND ’info dbsize logsize’`
ADMIN COMMAND 'makecp'	Makes a checkpoint.
ADMIN COMMAND 'messages [**-n**] [**warnings** | **errors**] [*count*]'	Displays server messages. Optional severity and message numbers can also be defined. For example: ADMIN COMMAND ’messages warnings 100’ displays last 100 warnings.
ADMIN COMMAND 'monitor {**on** | **off**} [**user** *username* | *user id*]'	Sets server monitoring on and off. Monitoring logs user activity and SQL calls to `soltrace.out` file

Option Syntax	Description
ADMIN COMMAND 'notify {**user** *username* | *user id* | **ALL** } *message*'	This command sends an event to a given user with event identifier NOTIFY. This identifier is used to cancel an event-waiting thread when the statement timeout is not long enough for a disconnect or to change the event registration. The following example sends a notify message to a user with user id **5**; the event then gets the value of the message parameter. ADMIN COMMAND 'notify user 5 Canceled by admin'
ADMIN COMMAND 'open'	Opens server for new connections; new connections are allowed.
ADMIN COMMAND 'parameter [*option*][*name*[=*value*]]'	Displays and sets server parameter values. For example: ▪ **parameter** used alone displays all parameters. ▪ **parameter genera**l displays all parameters from section “general.” ▪ **parameter general.readonly** displays a single parameter “readonly” from section “general.” ▪ **parameter com.trace=yes** sets communication trace on If **-r** is used, then only the current parameter values are returned.

ADMIN COMMAND 'perfmon [*options*] [*subsystem prefix*]'	Returns performance statistics from the server. Options are:
	▪ **-c** returns all values as the counter
	▪ **-d** returns short descriptions
	▪ **-v** returns current values
	▪ **-t** returns total values
	
	By default, some values are averages/second.
	
	The subsystem prefix is used to find matches to value names. Only those values are returned that match the subsystem prefix.
	
	The following example returns all information:
	
	`ADMIN COMMAND 'perfmon'`
	
	The following example returns all values whose name starts with prefix **File** as counters.
	
	`ADMIN COMMAND 'perfmon-c File'`

Option Syntax	**Description**
ADMIN COMMAND 'pid'	Returns server process id.
ADMIN COMMAND 'protocols'	Returns list of available communication protocols, one row for each protocol. Example: `ADMIN COMMAND 'protocols'`
ADMIN COMMAND 'report *file-name'*	Generates a report of server info to a file given as an argument.
ADMIN COMMAND 'shutdown'	Stops SOLID *SynchroNet*.
ADMIN COMMAND 'status'	Displays server statistics.
ADMIN COMMAND 'status backup'	Displays status of the last started backup. The status can be one of the following: ■ If the last backup was successful or any backups have not been requested, the output is 0 SUCCESS. ■ If the backup is in process; for example, started but not ready yet, the output is 14003 ACTIVE. ■ If the last backup failed, the output is: *errorcode* ERROR where the *errcode* shows the reason for the failure
ADMIN COMMAND 'throwout {*username* | *userid* | **all }**	Exits users from SOLID *SynchroNet*. To exit a specified user, give the user id as an argument. To throw out all users, use the keyword ALL as an argument.
ADMIN COMMAND 'trace {**on** | **off**} **sql** | **rpc** | **sync'**	Sets server trace on or off. This command is similar to the **monitor** command, but traces different entities and a different levels. By default, the output is written to the `soltrace.out` file.
ADMIN COMMAND 'userid'	Returns user identification number of the current connection. Example: `ADMIN COMMAND 'userid'`
ADMIN COMMAND 'userlist [**-l**] [*name* | *id*]'	Displays a list of users. option -l displays more detailed output.
ADMIN COMMAND 'version'	Displays server version info.

ALTER TABLE

ALTER TABLE *base_table_name*

{ADD [COLUMN] *column_identifier data_type* |

DROP [COLUMN] *column_identifier* | RENAME [COLUMN]

column_identifier column_identifier |

MODIFY [COLUMN]

column_identifier data-type} | MODIFY SCHEMA *schema_name* |

SET {OPTIMISTIC | PESSIMISTIC}

Note

Keywords CASCADE and RESTRICT are not supported in the SQL grammar of SOLID *SynchroNet*. Objects are always dropped with drop behavior RESTRICT.

Usage

The structure of a table may be modified through the `ALTER TABLE` statement. Within the context of this statement, columns may be added, modified, or removed.

The server allows users to change the width of a column using the ALTER TABLE command. A column width can be increased at any time (that is, whether a table is empty [no rows] or non-empty). However, the ALTER TABLE command disallows decreasing the column width when the table is non-empty; a table must be empty to decrease the column width.

Note that a column cannot be dropped if it is part of a unique or primary key.

The owner of a table can be changed using the ALTER TABLE *base_table_name* MODIFY SCHEMA *schema_name* statement. This statement gives all rights to the new owner of the table including creator rights. The old owner's access rights to the table, excluding the creator rights, are preserved.

Individual tables can be set to optimistic or pessimistic with the command ALTER TABLE base_table_name `SET {OPTIMISTIC | PESSIMISTIC}`. By default, all tables are optimistic. A database-wide default can be set in the `General` section of the configuration file with the parameter `Pessimistic = yes`.

Example

```
ALTER TABLE TEST ADD X INTEGER;

ALTER TABLE TEST RENAME COLUMN X Y;

ALTER TABLE TEST MODIFY COLUMN X SMALLINT;

ALTER TABLE TEST DROP COLUMN X;
```

ALTER TRIGGER

ALTER TRIGGER *trigger_name_attr* SET ENABLED | DISABLED

trigger_name_attr := [*catalog_name*.[*schema_name*]]*trigger_name* |

Usage

You can alter trigger attributes using the ALTER TRIGGER command. The valid attributes are ENABLED and DISABLED trigger.

The ALTER TRIGGER command causes a Solid server to ignore the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive or disable a trigger that is currently active.

You must be the owner of a table, or a user with DBA authority to alter a trigger from the table.

Example

```
ALTER TRIGGER SET ENABLED trig_on_employee;
```

ALTER USER

ALTER USER *user_name* IDENTIFIED BY *password*

Usage

The password of a user may be modified through the `ALTER USER` statement.

Example

```
ALTER USER MANAGER IDENTIFIED BY O2CPTG;
```

CALL

CALL *procedure_name* [(*parameter* [, *parameter* ...])]

Usage

Stored procedures are called with statement `CALL`.

Example

```
CALL proctest;
```

COMMIT

COMMIT WORK

Usage

The changes made in the database are made permanent by `COMMIT` statement. It terminates the transaction.

Example

```
COMMIT WORK;
```

CREATE CATALOG

CREATE CATALOG *catalog_name*

Usage

Catalogs allow you to logically partition databases so you can organize your data to meet the needs of your business or application. A database can have one or more catalogs. Users are prompted for a default catalog name when creating a new database or converting an old database to a new format. This default catalog name allows for backward compatibility of SOLID databases prior to version 3.5.

A catalog can have zero or more *schema_names*. The default schema name is *user_id*. A schema can have zero or more database object names. A database object can be qualified by a schema or user ID.

The catalog name is used to qualify a database object name. Database object names are qualified in all DML statements as:

catalog_name.schema_name.database_object

or

catalog_name.user_id.database_object

Only a user with DBA authority (SYS_ADMIN_ROLE) can create a catalog for a database.

To use schemas, a schema name must be created before creating the database object name. However, a database object name can be created without a schema name. In such cases, database objects are qualified using *user_id* only. For details on creating schemas, read *"CREATE SCHEMA"* on page E-21.

A catalog context can be set in a program using:

SET CATALOG *catalog_name*

A catalog can be dropped from a database using:

DROP CATALOG *catalog_name*

When dropping a catalog name, all objects associated with the catalog name must be dropped prior to dropping the catalog.

Following are the rules for resolving catalog names:

- A fully qualified name (*catalog_name*.*schema_name*.*database_object_name*) does not need any name resolution, but will be validated.
- If a catalog context is not set using SET CATALOG, then all database object names are resolved always using the default catalog name as the catalog name. The database object name is resolved using schema name resolution rules. For details on these rules, read *"CREATE SCHEMA"* on page E-21.
- If a catalog context is set and the catalog name cannot be resolved using the *catalog_name* in the context, then *database_object_name* resolution fails.
- To access a database system catalog, users do not need to know the system catalog name. Users can specify ""._SYSTEM.*table*. *SynchroNet t*ranslates the empty string "" used as a catalog name to the default catalog name. *SynchroNet* also provides automatic resolution of _SYSTEM schema to the system catalog, even when the catalog name is not provided.

Examples

```
CREATE CATALOG C;
SET CATALOG C;
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T;
SELECT * FROM T;
-- the name T is resolved to C.S.T
-- Assume the userid is SMITH
CREATE CATALOG C;
SET CATALOG C;
CREATE TABLE T;
SELECT * FROM T;
--The name T is resolved to C.SMITH.T

-- Assume there is no Catalog context set.
-- Meaning the default catalog name is BASE or the setting
-- of the base catalog.
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T;
SELECT * FROM T;
--The name T is resolved to <BASE>.S.T

CREATE CATALOG C1;
SET CATALOG C1;
CREATE SCHEMA S1;
SET SCHEMA S1;
CREATE TABLE T1 (c1 INTEGER);

CREATE CATALOG C2;
SET CATALOG C2;
CREATE SCHEMA S2;
SET SCHEMA S2;
CREATE TABLE T1 (c2 INTEGER)

SET CATALOG BASE;
SET SCHEMA USER;
SELECT * FROM T1;
-- This select will give an error as it
-- cannot resolve the T1.
```

CREATE EVENT

CREATE EVENT *event_name* [(*parameter_definition* [, *parameter_definition* ...])]

Usage

Event alerts are used to signal an event in the database. Events are simple objects with a name. The use of event alerts removes resource consuming database polling from applications.

An event object is created with the SQL statement

`CREATE EVENT` *`event_name`* `[`*`parameter_list`*`]`

The name can be any user-specified alphanumeric string. The parameter list specifies parameter names and parameter types. The parameter types are normal SQL types.

Events are dropped with the SQL statement

`DROP EVENT` *`event_name`*

Events are triggered and received inside stored procedures. Special stored procedure statements are used to trigger and receive events.

The event is triggered with the stored procedure statement

`POST EVENT` *`event_name`* `[`*`parameters`*`]`

Event parameters must be local variables or parameters in the stored procedure where the event is triggered. All clients that are waiting for the posted event will receive the event.

To make a procedure wait for an event to happen, the `WAIT EVENT` construct is used in a stored procedure:

```
wait_event_statement ::=

   WAIT EVENT

      [event_specification ...]

   END WAIT


event_specification ::=

   WHEN event_name (parameters) BEGIN

      statements

   END EVENT
```

Each connection has its own event queue. To specify the events to be collected in the event queue command REGISTER EVENT *event_name* (*parameters*) is used. Events are removed from the event queue with command UNREGISTER EVENT *event_name* (*parameters*).

Example of a procedure that waits for an event:

```
"create procedure event_wait(i1 integer)
returns (result varchar)
begin
declare i integer;
declare c char(4);

i := 0;

wait event
   when test1 begin
      result := 'event1';
      return;
   end event

   when test2(i) begin
   end event

   when test3(i, c) begin
  end event
end wait

if i <> 0 then
   result := 'if';
   post event test1;
```

```
else
   result := 'else';
   post event test2(i);
   post event test3(i, c);
end if
end";
```

The creator of an event or the database administrator can grant and revoke access rights. Access rights can be granted to users and roles. The select access right gives waiting access to an event. The insert access right gives triggering access to an event.

Example

```
CREATE EVENT ALERT1(I INTEGER, C CHAR(4));
```

CREATE INDEX

CREATE [UNIQUE] INDEX *index_name*

ON *base_table_name*

(*column_identifier* [ASC | DESC]

[, *column_identifier* [ASC | DESC]] ...)

Usage

Creates an index for a table based on the given columns. Keyword `UNIQUE` specifies that columns being indexed must contain unique values. Keywords `ASC` and `DESC` specify whether the given columns should be indexed in ascending or descending order. If not specified ascending order is used.

Example

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);
CREATE INDEX X_TEST ON TEST (I, J);
```

CREATE PROCEDURE

CREATE PROCEDURE *procedure_name* [(*parameter_definition*

[, *parameter_definition* ...])]

[RETURNS (*parameter_definition* [, *parameter_definition* ...])]

BEGIN *procedure_body* END;

parameter_definition ::= *parameter_name data_type*

procedure_body ::= [*declare_statement*; ...]

procedure_statement; [*procedure_statement*; ...]

declare_statement ::= DECLARE *variable_name data_type*

procedure_statement ::= *prepare_statement* | *exec_statement* | *fetch_statement* |

control_statement | *post_statement* | *wait_event_statement* |

wait_register_statement

prepare_statement ::= EXEC SQL PREPARE *cursor_name sql_statement*

execute_statement ::=

EXEC SQL EXECUTE

cursor_name

[USING (*variable* [, *variable* ...])]

[INTO (*variable* [, *variable* ...])] |

EXEC SQL {CLOSE | DROP} *cursor_name* |

EXEC SQL {COMMIT | ROLLBACK} WORK |

EXEC SQL SET TRANSACTION {READ ONLY | READ WRITE} |

EXEC SQL WHENEVER SQLERROR {ABORT | ROLLBACK [WORK], ABORT}

EXEC SEQUENCE *sequence_name*.CURRENT INTO *variable* |

EXEC SEQUENCE *sequence_name*.NEXT INTO *variable* |

EXEC SEQUENCE *sequence_name* SET VALUE USING *variable*

fetch_statement ::= EXEC SQL FETCH *cursor_name* *post_statement* ::= POST EVENT *event_name* [*parameters*]

wait_event_statement ::=

    WAIT EVENT

        [*event_specification* ...]

    END WAIT

event_specification ::=

    WHEN *event_name* (*parameters*) BEGIN

        statements

    END EVENT

wait_register-statement ::=

    REGISTER EVENT*event_name* (*parameters*) |

    UNREGISTER EVENT*event_name* (*parameters*)

control_statement ::=

    SET *variable_name* = *value* | *variable_name* := *value* |

    WHILE *expression*

        LOOP *procedure_statement*... END LOOP |

    LEAVE |

    IF *expression* THEN *procedure_statement* ...

        [ELSEIF *procedure_statement* ... THEN] ...

        ELSE *procedure_statement* ... END IF |

    RETURN | RETURN SQLERROR OF *cursor_name* | RETURN ROW

Usage

Stored procedures are simple programs, or procedures, that are executed in the server. The user can create a procedure that contains several SQL statements or a whole transaction and execute it with a single call statement. Usage of stored procedures reduces network traffic and allows more strict control to access rights and database operations.

Procedures are created with the statement

CREATE PROCEDURE *name body*

and dropped with the statement

DROP PROCEDURE *name*

Procedures are called with the statement

CALL *name* [*parameter* ...]

Procedures can take several input parameters and return a single row or several rows as a result. The result is built from specified output parameters. Procedures are thus used in ODBC in the same way as the SQL SELECT statement.

Procedures are owned by the creator of the procedure. Specified access rights can be granted to other users. When the procedure is run, it has the creator's access rights to database objects.

The stored procedure syntax is a proprietary syntax modeled from SQL3 specifications and dynamic SQL. Procedures contain control statements and SQL statements.

The following control statements are available in the procedures:

Control statement	**Description**
set *variable* = *expression*	Assigns a value to a variable. The value can be either a literal value (e.g., 10 or 'text') or another variable. Parameters are considered as normal variables.
variable := *expression*	Alternate syntax for assigning values to variables.

while
  expr
loop
  statement-list
end loop
	Loops while expression is true.

leave	Leaves the innermost while loop and continues executing the procedure from the next statement after the keyword end loop.
if *expr* then *statement-list1* else *statement-list2* end if	Executes *statements-list1* if expression *expr* is true; otherwise, executes *statement-list2*.
if *expr1* then *statement-list1* elseif *expr2* then *statement-list2* end if	If *expr1* is true, executes *statement-list1*. If *expr2* is true, executes *statement-list2*. The statement can optionally contain multiple *elseif* statements and also an *else* statement.
return	Returns the current values of output parameters and exits the procedure. If a procedure has a one *return row* statement, *return* behaves like *return norow*.
return sqlerror of *cursor-name*	Returns the sqlerror associated with the cursor and exits the procedure.
return row	Returns the current values of output parameters and continues execution.
return norow	Returns the end of the set and exits the procedure.

All SQL DML and DDL statements can be used in procedures. Thus the procedure can, for example, create tables or commit a transaction. Each SQL statement in the procedure is atomic.

Preparing SQL Statements

The SQL statements are first prepared with the statement

EXEC SQL PREPARE *cursor sql-statement*

The *cursor* specification is a cursor name that must be given. It can be any unique cursor name inside the transaction. Note that if the procedure is not a complete transaction, other open cursors outside the procedure may have conflicting cursor names.

Executing Prepared SQL Statements

The *SQL statement* is executed with the statement

EXEC SQL EXECUTE *cursor* [*opt-using*] [*opt-into*]

The optional *opt-using* specification has the syntax

USING (*variable_list*)

where *variable_list* contains a list of procedure variables or parameters separated by a comma. These variables are input parameters for the SQL statement. The SQL input parameters are marked with the standard question mark syntax in the prepare statement. If the SQL statement has no input parameters, the USING specification is ignored.

The optional *opt_nto* specification has the syntax

INTO (*variable_list*)

where *variable_list* contains the variables that the column values of the SQL SELECT statement are stored into. The INTO specification is effective only for SQL SELECT statements.

After the execution of UPDATE, INSERT and DELETE statements an additional variable is available to check the result of the statement. Variable SQLROWCOUNT contains the number of rows affected by the last statement.

Fetching Results

Rows are fetched with the statement

EXEC SQL FETCH *cursor*

If the fetch completed successfully, the column values are stored into the variables defined in the *opt_into* specification.

Checking for Errors

The result of each EXEC SQL statement executed inside a procedure body is stored into the variable SQLSUCCESS. This variable is automatically generated for every procedure. If the previous SQL statement was successful, a value one is stored into SQLSUCCESS. After a failed SQL statement, a value zero is stored into SQLSUCCESS.

EXEC SQL WHENEVER SQLERROR {ABORT **|** [ROLLBACK [WORK], ABORT}

is used to decrease the need for IF NOT SQLSUCCESS THEN tests after every executed SQL statement in a procedure. When this statement is included in a stored procedure all return values of executed statements are checked for errors. If statement execution returns an error, the procedure is automatically aborted. Optionally the transaction can be rolled back.

The error string of latest failed SQL statement is stored into variable SQLERRSTR.

Using Transactions

EXEC SQL {COMMIT **|** ROLLBACK} WORK

is used to terminate transactions.

EXEC SQL SET TRANSACTION {READ ONLY **|** READ WRITE}

is used to control the type of transactions.

Using Sequencer Objects and Event Alerts

Refer to the usage of the CREATE SEQUENCE and CREATE EVENT statements.

Procedure Stack Functions

The following functions may be used to analyze the current contents of the procedure stack: PROC_COUNT(), PROC_NAME(N), PROC_SCHEMA(N).

PROC_COUNT() returns the number of procedures in the procedure stack. This includes the current procedure.

PROC_NAME(N) returns the Nth procedure name is the stack. First procedure position is zero.

PROC_SCHEMA(N) returns the schema name of the Nth procedure in procedure stack.

Example 1

```
"create procedure test2(tableid integer)
   returns (cnt integer)
begin
   exec sql prepare c1 select count(*) from
      sys_tables where id > ?;
   exec sql execute c1 using (tableid) into
      (cnt);
   exec sql fetch c1;
   exec sql close c1;
end";
```

Example 2

```
"create procedure return_tables
returns (name varchar)
begin
        exec sql whenever sqlerror rollback, abort;
        exec sql prepare c1 select table_name
             from sys_tables;
        exec sql execute c1 into (name);
        while sqlsuccess loop
            exec sql fetch c1;
            if not sqlsuccess
            then leave;
            end if
            return row;
        end loop;
        exec sql close c1;
end";
```

CREATE ROLE

CREATE ROLE *role_name*

Usage

Creates a new user role.

Example

```
CREATE ROLE GUEST_USERS;
```

CREATE SCHEMA

CREATE SCHEMA *schema_name*

Usage

Creates a collection of database objects, such as tables, views, indexes, events, triggers, sequences, and stored procedures for a database user. The schema name is used to qualify a database object name. Database object names are qualified in all DML statements as:

catalog_name.schema_name.database_object_name

or

user_id.database_object_name

To logically partition a database, users can create a catalog before they create a schema. For details on creating a catalog, read *"CREATE CATALOG"* on page E-8. Note that when creating a new database or converting an old database to a new format, users are prompted for a default catalog name.

To use schemas, a schema name must be created before creating the database object name (such as a table name or procedure name). However, a database object name can be created without a schema name. In such cases, database objects are qualified using *user_id* only.

You can specify the database object names in a DML statement explicitly by fully qualifying them or implicitly by setting the schema name context using:

SET SCHEMA *schema_name*

A schema can be dropped from a database using:

DROP SCHEMA *schema_name*

When dropping a schema name, all objects associated with the schema name must be dropped prior to dropping the schema.

A schema context can be removed using:

SET SCHEMA USER

Following are the rules for resolving schema names:

- A fully qualified name (*schema_name.database_object_name*) does not need any name resolution, but will be validated.
- If a schema context is not set using SET SCHEMA, then all database object names are resolved always using the user id as the schema name.

- If the database object name cannot be resolved from the schema name, then the database object name is resolved from all existing schema names.
- If name resolution finds either zero matching or more than one matching database object name, then a Solid server issues a name resolution conflict error.

Examples

```
-- Assume the userID is SMITH.
CREATE SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);
SET SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (ID INTEGER);
SELECT ID FROM EMPLOYEE;
-- In this case, the table is qualified to FINANCE.EMPLOYEE
SELECT EMP_ID FROM EMPLOYEE;
-- This will give an error as the context is with FINANCE and
-- table is resolved to FINANCE.EMPLOYEE

--The following are valid schema statements: one with a schema context,
--the other without.
SELECT ID FROM FINANCE.EMPLOYEE;
SELECT EMP_ID FROM SMITH.EMPLOYEE
--The following statement will resolve to schema SMITH without a schema
--context
SELECT EMP_ID FROM EMPLOYEE;
```

CREATE SEQUENCE

CREATE [DENSE] SEQUENCE *sequence_name*

Usage

Sequencer objects are objects that are used to get sequence numbers.

Using a dense sequence guarantees that there are no holes in the sequence numbers. The sequence number allocation is bound to the current transaction. If the transaction rolls back, also the sequence number allocations are rolled back. The drawback of dense sequences is that the sequence is locked out from other transactions until the current transaction ends.

Using a sparse sequence guarantees uniqueness of the returned values, but they are not bound to the current transaction. If a transaction allocates a sparse sequence number and later rolls back, the sequence number is simply lost.

The advantage of using a sequencer object instead of a separate table is that the sequencer object is specifically fine-tuned for fast execution and requires less overhead than normal update statements.

Sequence values can be incremented and used within SQL statements. These constructs can be used in SQL:

sequence_name.CURRVAL

sequence_name.NEXTVAL

Sequences can also be used inside stored procedures. The current sequence value can be retrieved using the following stored procedure statement:

EXEC SEQUENCE *sequence_name*.CURRENT INTO *variable*

The new sequence value can be retrieved using the following stored procedure statement:

EXEC SEQUENCE *sequence_name*.NEXT INTO *variable*

Sequence values can be set with the following stored procedure statement:

EXEC SEQUENCE *sequence_name* SET VALUE USING *variable*

Select access rights are required to retrieve the current sequence value. Update access rights are required to allocate new sequence values. These access rights are granted and revoked in the same way as table access rights.

Examples

```
CREATE DENSE SEQUENCE SEQ1;
INSERT INTO ORDER (id) VALUES (order_sequence.NEXTVAL);
```

CREATE TABLE

CREATE TABLE *base_table_name* (*column_element* [, *column_element*] ...)

base_table_name ::= *base_table_identifier* | *schema_name*.*base_table_identifier* |

catalog_name.*schema_name*.*base_table_identifier*

column_element ::= *column_definition* | *table_constraint_definition* *column-definition* ::= *column_identifier*

data_type [*column_constraint_definition* [*column_constraint_definition*] ...]

column-constraint-definition ::= NOT NULL **|** NOT NULL UNIQUE **|**

NOT NULL PRIMARY KEY **|** REFERENCES *ref_table_name referenced_columns*

| CHECK (*check_condition*)

table_constraint_definition ::= UNIQUE (*column_identifier* [, *column_identifier*] ...) **|**

PRIMARY KEY (*column_identifier* [, *column_identifier*] ...) **|**

CHECK (*check-condition*) **|**

FOREIGN KEY (*column-identifier* [, *column-identifier]* ...)

REFERENCES *table-name* (*column-identifier* [, *column-identifier*] ...)

Usage

Tables are created through the CREATE TABLE statement. The CREATE TABLE statement requires a list of the columns created, the data types, and, if applicable, sizes of values within each column, in addition to other related alternatives (such as whether or not null values are permitted).

Example

```
CREATE TABLE DEPT (DEPTNO INTEGER NOT NULL, DNAME VARCHAR, PRIMARY
KEY(DEPTNO));

CREATE TABLE DEPT2 (DEPTNO INTEGER NOT NULL PRIMARY KEY, DNAME VARCHAR);

CREATE TABLE DEPT3 (DEPTNO INTEGER NOT NULL UNIQUE, DNAME VARCHAR);

CREATE TABLE DEPT4 (DEPTNO INTEGER NOT NULL, DNAME VARCHAR,
UNIQUE(DEPTNO));
```

```
CREATE TABLE EMP (DEPTNO INTEGER, ENAME VARCHAR, FOREIGN KEY (DEPTNO)
REFERENCES DEPT (DEPTNO));

CREATE TABLE EMP2 (DEPTNO INTEGER, ENAME VARCHAR, CHECK (ENAME IS NOT
NULL), FOREIGN KEY (DEPTNO) REFERENCES DEPT (DEPTNO));
```

CREATE TRIGGER

CREATE TRIGGER *trigger_name time_of_operation triggering_event*

ON *table_name* [REFERENCING *column_reference*] *trigger_body*

where:

trigger_name	:= *literal*
time_of_operation	::= BEFORE | AFTER
triggering_event	:: = INSERT | UPDATE | DELETE
column_reference::= OLD *old_column_name* [AS] *old_col_identifier*	
	[*, REFERENCING column_reference* |
	NEW *new_column_name* [AS] *new_col_identifier*
	[*, REFERENCING column_reference*]
trigger_body	::= *trigger_body*:= [*declare_statement*] *trigger_statement*
	{, *trigger_statement*]}
old_column_name	:= *literal*
new_column_name	:= *literal*
old_col_identifier	:= *literal*
new_col_identifier	:= *literal*

▶ **Note**

This appendix is intended to provide a quick reference to using Solid SQL commands. For details on when and how to use triggers, read *Chapter 3, Stored Procedures, Events, Trig-*

gers, and Sequences of the **SOLID Programmer Guide**.

Usage

A trigger activates a stored procedure code, which a Solid server automatically executes when a user attempts to change the data in a table. You may create one or more triggers on a table, with each trigger defined to activate on a specific INSERT, UPDATE, or DELETE command. When a user modifies data within the table, the trigger that corresponds to the command is activated.

You can only use inline or stored procedures with triggers. In addition, you must first store the procedure with the CREATE PROCEDURE command. A procedure invoked from a trigger body can invoke other triggers.

To create a trigger, you must be a DBA or owner of the table on which the trigger is being defined.

Triggers are created with the statement

CREATE TRIGGER *name body*

and dropped from the system catalog with the statement

DROP TRIGGER *name*

Triggers are disabled by using the statement

ALTER TRIGGER *name*

When you disable a trigger defined on a table, a Solid server ignores the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive.

▶ **Note**

Following is a brief summary of the keywords and clauses used in the CREATE TRIGGER command. For more in depth information on usage, read *Chapter 3, Stored Procedures, Events, Triggers, and Sequences* of the **SOLID Programmer Guide**.

Trigger name

The *trigger_name* identifies the trigger and can contain up to 254 characters.

Trigger body

The *trigger_body* is the statement or statements to be executed when a trigger fires. The trigger body definition is identical to the stored procedure definition. The trigger parameters are the table column values. For details on creating a trigger body, read *"CREATE PROCEDURE"* on page E-13.

A trigger body may also invoke any procedure registered with a Solid server. SOLID procedure invocation rules follow standard procedure invocation practices.

In a procedure definition, you can use COMMIT and ROLLBACK statements. But, in a trigger body, you *cannot* use COMMIT (including AUTOCOMMIT and COMMIT WORK) and ROLLBACK statements; you can use only the WHENEVER SQLERROR ABORT statement.

You must explicitly check for business logic errors and raise an error.

Trigger Event Clauses

The BEFORE or AFTER clause specifies whether to execute the trigger before or after the the invoking DML statement, which modifies data.

The INSERT | UPDATE | DELETE clause indicates the trigger action when a user action is attempted (INSERT, UPDATE, DELETE).

Statements related to processing a trigger occurs first before commits and autocommits from the invoking DML (INSERT, UPDATE, DELETE) statements on tables. If a trigger body or a procedure called within the trigger body attempts to execute a COMMIT (including AUTOCOMMIT or COMMIT WORK) or ROLLBACK, than a Solid server returns an appropriate run-time error.

INSERT specifies that the trigger is activated by an INSERT on the table. Loading *n* rows of data is considered as *n* inserts.

Note

There may be some performance impact if you try to load the data with triggers enabled. Depending on your business need, you may want to disable the triggers before loading and enable them after loading. For details, see *"ALTER TRIGGER"* on page E-7.

DELETE specifies that the trigger is activated by a DELETE on the table.

UPDATE specifies that the trigger is activated by an UPDATE on the table. Note the following rules for using the UPDATE clause:

- The same column cannot be referenced by more than one UPDATE trigger.
- A Solid server allows for recursive update to the same table and does not prohibit recursive updates to the same row.

A separate trigger can be generated for each INSERT, DELETE, or UPDATE operation on a table. You can define, by default, up to one trigger for each combination of table, event (INSERT, UPDATE, DELETE) and time (BEFORE and AFTER). This means there can be a maximum of 6 triggers per table.

Note

The triggers are applied to each row. This means that if there are 10 inserts, a trigger is executed 10 times.

Referencing Clause

This clause is optional when creating a trigger on an INSERT/UPDATE/DELETE operation. It provides a way to reference the current column identifiers in the case of INSERT and DELETE operations, and both the old column identifier and the new updated column identifier by aliasing the table on which an UPDATE operation occurs.

You must specify the *old_column_identifier* or the *new_col_identifier* to access them before and after an UPDATE operation. A Solid server does not provide access to them unless you define them using the REFERENCING subclause.

OLD *old_column_name* AS *old_col_identifier* or
NEW *new_column_name* AS *new_col_identifier*

The subclause of the REFERENCING clause allows you to reference the values of columns both before and after an UPDATE operation.

It produces a set of old and new column values which can be passed to an inline or stored procedure; once passed, the procedure contains logic (for example, domain constraint checking) used to evaluate these parameter values.

Use the OLD AS clause to alias the table's old identifier as it exists before the UPDATE. Use the NEW AS clause to alias the table's new identifier as it exists after the UPDATE.

You cannot use the same name for the *old_column_name* and the *new_column_name*, or for the *old_column_identifier* and the *new_column_identifier.*

Each column that is referenced as NEW or OLD should have a separate REFERENCING subclause.

The statement atomicity in a trigger is such that operations made in a trigger are visible to the next SQL statements inside the trigger. For example, if you execute an INSERT statement in a trigger and then also perform a select in the same trigger, then the inserted row is visible.

In the case of AFTER trigger, an inserted row or an updated row is visible in the after insert trigger, but a deleted row cannot be seen for a select performed within the trigger. In the case of a BEFORE trigger, an inserted or updated row is invisble within the trigger and a deleted row is visible.

The table below summarizes the statement atomicity in a trigger, indicating whether the row is visible to the SELECT statement in the trigger body.

Operation	**BEFORE TRIGGER**	**AFTER TRIGGER**
INSERT	row is invisible	row is visible
UPDATE	previous value is invisible	new value is visible
DELETE	row is visible	row is invisible

Other Restrictions

- You cannot define triggers on a view (even if the view is based on a single table).
- You cannot alter a table that has a trigger defined on it when the dependent columns are affected.
- You cannot create a trigger on a system table.
- To use the stored procedure that a trigger calls, provide the catalog, schema/owner and name of the table on which the trigger is defined. and specify whether to enable or disable the triggers in the table. For more details on triggers and stored procedures, see *Chapter 3, Stored Procedures, Events, Triggers, and Sequences* of the **SOLID Programmer Guide**.
- You cannot execute triggers that reference dropped or altered objects. To prevent this error:
  - Recreate any referenced object that you drop.
  - Restore any referenced object you changed back to its original state (known by the trigger).

- You can use reserved words in trigger statements if they are enclosed in double quotes. For example, the following CREATE TRIGGER statement references a column named "data" which is a reserved word.

```
"CREATE TRIGGER TRIG1 ON TMPT BEORE INSERT
REFERENCING NEW "DATA" AS NEW_DATA
BEGIN
END"
```

Setting the Maximum Number of Nested Triggers

Triggers can invoke other triggers or a trigger can invoke itself (or recursive triggers). You can nest triggers up to 16 levels deep. The maximum number of nested triggers is set in the `MaxNestedTriggers` parameter in the SQL section of the SOLID.INI configuration file:

```
[SQL] MaxNestedTriggers=n
```

where *n* is the maximum number of nested triggers.

The default is 16 triggers.

Setting the Triggers Cache

Triggers are cached on a separate cache in the Solid server; each user has a separate cache for triggers. As the triggers are executed, the trigger procedure logic is cached in the trigger cache and is resued when the trigger is executed again.

The cache size is set in the `TriggerCache` parameter in the SQL section of the SOLID.INI configuration file:

```
[SQL] TriggerCache=n
```

where *n* is the number of triggers that is reserved for the cache.

Checking for Errors

At times, it is possible to receive an error in executing a trigger. The error may be due to execution of SQL statements or business logic. If a trigger returns an error, it causes its invoking DML command to fail. To automatically return errors during the execution of an DML statement, you must use the WHENEVER SQLERROR ABORT statement in the trigger body. Otherwise, errors must be checked explicitly within the trigger body after each procedure call or SQL statement.

For any errors in the user written business logic as part of the trigger body, users can receive errors in a procedure variable using the SQL statement:

RETURN SQLERROR *error_string*

or

RETURN SQLERROR *char_variable*

The error is returned in the following format:

`User error: error_string`

If a user does not specify the RETURN SQLERROR statement in the trigger body, then all trapped SQL errors are raised with a default *error_string* determined by the system. For details, see *"Error Codes"* on page B-1.

▶ **Note**

Triggered SQL statements are a part of the invoking transaction. If the invoking DML statement fails due to either the trigger or another error that is generated outside the trigger, all SQL statements within the trigger are rolled back along with the failed invoking DML command.

Triggers Stack Functions

The following functions may be used to analyze the current contents of the trigger stack:

TRIG_COUNT() returns the number of triggers in the trigger stack. This includes the current trigger. The return value is an integer.

TRIG_NAME(n) returns the nth trigger name in the trigger stack. The first trigger position or offset is zero.

TRIG_SCHEMA(n) returns the nth trigger schema name in the trigger stack. The first trigger position or offset is zero. the return value is a string.

Example

```
"CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST
        BEFORE INSERT
        REFERENCING NEW BI AS NEW_BI
BEGIN
        EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES (
        'BI', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BI;
        SET NEW_BI = 'TRIGGER_BI';
END";
```

CREATE USER

CREATE USER *user_name* IDENTIFIED BY *password*

Usage

Creates a new user with a given password.

Example

```
CREATE USER HOBBES IDENTIFIED BY CALVIN;
```

CREATE VIEW

CREATE VIEW *viewed_table_name* [(*column_identifier* [, *column_identifier*]...)]
AS *query-specification*

Usage

A view can be viewed as a virtual table; that is, a table that does not physically exist, but rather is formed by a query specification against one or more tables.

Example

```
CREATE VIEW TEST_VIEW
   (VIEW_I, VIEW_C, VIEW_ID)
   AS SELECT I, C, ID FROM TEST;
```

DELETE

DELETE FROM *table_name* [WHERE *search_ondition*]

Usage

Depending on your search condition the specified row(s) will be deleted from a given table.

Example

```
DELETE FROM TEST WHERE ID = 5;
DELETE FROM TEST;
```

DELETE (positioned)

DELETE FROM *table_name* WHERE CURRENT OF *cursor_name*

Usage

The positioned DELETE statement deletes the current row of the cursor.

Example

```
DELETE FROM TEST WHERE CURRENT OF MY_CURSOR;
```

DROP CATALOG

DROP CATALOG *catalog_name*

Usage

The DROP CATALOG statement drops the specified catalog from the database. All the objects associated with the specified *catalog_name* must be dropped prior to using this statement; the DROP CATALOG statement is not a cascaded operation.

Example

```
DROP CATALOG C1;
```

DROP EVENT

DROP EVENT *event_name*

Usage

The DROP EVENT statement removes the specified event from the database.

Example

```
DROP EVENT EVENT_TEST;
```

DROP INDEX

DROP INDEX *index_name*

Usage

The `DROP INDEX` statement removes the specified index from the database.

Example

```
DROP INDEX UX_TEST;
```

DROP PROCEDURE

DROP PROCEDURE *procedure_name*

Usage

The `DROP PROCEDURE` statement removes the specified procedure from the database.

Example

```
DROP PROCEDURE PROCTEST;
```

DROP ROLE

DROP ROLE *role_name*

Usage

The DROP ROLE statement removes the specified role from the database.

Example

```
DROP ROLE GUEST_USERS;
```

DROP SCHEMA

DROP SCHEMA *schema_name*

Usage

The DROP SCHEMA statement drops the specified schema from the database. All the objects associated with the specified *schema_name* must be dropped prior to using this statement; the DROP SCHEMA statement is not a cascaded operation.

Example

```
DROP SCHEMA FINANCE;
```

DROP SEQUENCE

DROP SEQUENCE *sequence_name*

Usage

The `DROP SEQUENCE` statement removes the specified sequence from the database.

Example

```
DROP SEQUENCE SEQ1;
```

DROP TABLE

DROP TABLE *base_table_name*

► **Note**

Objects are always dropped with drop behavior RESTRICT.

Usage

The `DROP TABLE` statement removes the specified table from the database.

Example

```
DROP TABLE TEST;
```

DROP TRIGGER

DROP TRIGGER [*catalog_name*[*schema_name*]]*trigger_name*

DROP TRIGGER *trigger_name*
DROP TRIGGER *schema_name.trigger_name*
DROP TRIGGER c*atalog_name.schema_name.trigger_name*

Usage

Drops (or deletes) a trigger defined on a table from the system catalog.

You must be the owner of a table, or a user with DBA authority to delete a trigger from the table.

Example

```
DROP TRIGGER TRIGA;
```

DROP USER

DROP USER *user_name*

Usage

The `DROP USER` statement removes the specified user from the database. All the objects associated with the specified user_name must be dropped prior to using this statement; the DROP USER statement is not a cascaded operation.

Example

```
DROP USER HOBBES;
```

DROP VIEW

DROP VIEW *viewed_table_name*

Usage

The `DROP VIEW` statement removes the specified view from the database.

▶ **Note**

Objects are always dropped with drop behavior RESTRICT.

Example

```
DROP VIEW TEST_VIEW;
```

EXPLAIN PLAN FOR

EXPLAIN PLAN FOR *sql_statement*

Usage

The `EXPLAIN PLAN FOR` statement shows the selected search plan for the specified SQL statement.

Example

```
EXPLAIN PLAN FOR select * from tables;
```

GRANT

GRANT {ALL | *grant_privilege* [, *grant_privilege*]...}
    ON *table_name*
TO {PUBLIC | *user_name* [, *user_name*]... |
    role_name [, *role_name*]... }
[WITH GRANT OPTION]

GRANT *role_name* TO *user_name*

grant_privilege ::= DELETE | INSERT | SELECT |
    UPDATE [(*column_identifier* [, *column_identifier*]...)] |
  REFERENCES [(*column_identifier* [, *column_identifier*]...)]

GRANT EXECUTE ON *procedure_name*
  TO {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

GRANT {SELECT | INSERT} ON *event_name*
  TO {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

GRANT {SELECT | UPDATE} ON *sequence_name*
  TO {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

Usage

The `GRANT` statement is

1. used to grant privileges to the specified user or role.
2. used to grant privileges to the specified user by giving

the user the privileges of the specified role.

If you do use the optional `WITH GRANT OPTION`, you give permission for the user(s) to whom you are granting the privilege to pass it on to other users.

Example

```
GRANT GUEST_USERS TO CALVIN;
GRANT INSERT, DELETE ON TEST TO GUEST_USERS;
```

HINT

--(* vendor (SOLID), product (Engine), option(hint)

--*hint* ***)--**

hint:=

[MERGE JOIN |

LOOP JOIN |

JOIN ORDER FIXED |

INTERNAL SORT |

EXTERNAL SORT |

INDEX [REVERSE] *table_name*.*index_name* |

PRIMARY KEY [REVERSE] *table_name*

FULL SCAN *table_name* |

[NO] SORT BEFORE GROUP BY]

Following is a description of the keywords and clauses used in the syntax:

Pseudo comment identifier

The pseudo comment prefix is followed by identifying information. You must specify the vendor as SOLID, product as Engine, and the option, which is the pseudo comment class name, as hint.

Hint

Hints always follow the SELECT, UPDATE, or DELETE keyword that applies to it.

Note

Hints are not allowed after the INSERT keyword.

Each subselect requires its own hint; for example, the following are valid uses of hints syntax:

INSERT INTO ... SELECT *hint* FROM ...

UPDATE *hint* TABLE ... WHERE *column* = (SELECT *hint* ... FROM ...)

DELETE *hint* TABLE ... WHERE *column* = (SELECT hint ... FROM ...)

Be sure to specify multiple hints in one pseudo comment separated by commas as shown in the following examples:

Example 1

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
*
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

Example 2

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--INDEX TAB1.INDEX1
--INDEX TAB1.INDEX1 FULL SCAN TAB2 *)--
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

Hint is a specific semantic, corresponding to a specific behavior. Following is a list of possible hints:

Hint	**Definition**
MERGE JOIN	Directs the Optimizer to choose the merge join access plan in a select query for all tables listed in the FROM clause. Use this hint when the data is sorted by a join key and the nested loop join causes performance is not adequate. The MERGE JOIN option selects the merge join only where there is an equal predicate between tables. Otherwise, the Optimizer selects LOOP JOIN even if the MERGE JOIN hint is specified. Note that when data is not sorted before performing the merge operation, the Solid query executor sorts the data. When considering the usage of this hint, keep in mind that the merge join with a sort is more resource intensive than the merge join without the sort.
LOOP JOIN	Directs the Optimizer to pick the nested loop join in a select query for all tables listed in the FROM clause. By default, the Optimizer does not pick the nested loop join. Using the loop join when tables are small and fit in memory may offer greater efficiency than using other complex join algorithms.
JOIN ORDER FIXED	Specifies that the Optimizer use tables in a join in the order listed in the FROM clause of the query. This means that the Optimizer does not attempt to rearrange any join order and does not try to find alternate access paths to complete the join. Before using this hint, be sure to run the EXPLAIN PLAN to view the associated plan. This gives you an idea on the access plan used for executing the query with this join order.
INTERNAL SORT	Specifies that the query executor use the internal sort. Use this hint if the expected result set is small (100s of rows as opposed to 1000s of rows); for example, if you are performing some aggregates, ORDER BY with small result sets, or GROUP BY with small result sets, etc. This hint avoids the use of the more expensive external sort.
EXTERNAL SORT	Specifies that the query executor use the external sort. Use this hint when the expected result set is large and does not fit in memory; for example, if the expected result set has 1000s of rows. In addition, specify the SORT working directory in the `solid.ini` before using the external sort hint. If a working directory is not specified, you will receive a run-time error.

Hint	**Definition**
INDEX {REVERSE} *table_name*.*index_name*	Forces a given index scan for a given table. In this case, the Optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. The optional keyword **REVERSE** returns the rows in the reverse order. In this case, the query executor begins with the last page of the index and starts returning the rows in the descending (reverse) key order of the index. Note that in *tablename*.*indexname*, the *tablename* is a fully qualified table name which includes the *catalogname* and *schemaname*.
PRIMARY KEY [REVERSE] *tablename*	Forces a primary key scan for a given table. The optional keyword **REVERSE** returns the rows in the reverse order. If the primary KEY is not available for the given table, then you will receive a run-time error.
FULL SCAN *table_name*	Forces a table scan for a given table. In this case, the optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. In this FULL SCAN, the query executor tries to use the PRIMARY KEY, if one is available. If not, then it uses the SYSTEM KEY.
[NO] SORT BEFORE GROUP BY	Indicates whether the SORT operation occurs before the result set is grouped by the GROUP BY columns. If the grouped items are few (100s of rows) then use NO SORT BEFORE. On the other hand, if the grouped items are large (1000s of rows), then use SORT BEFORE.

Usage

Due to various conditions with the data, user query, and database, the SQL Optimizer is not always able to choose the best possible execution plan. For more efficiency, you may want to force a merge join because you know, unlike the Optimizer, that your data is already sorted.

Or sometimes specific predicates in queries cause performance problems that the Optimizer cannot eliminate. The Optimizer may be using an index that you know is not optimal. In this case, you may want to force the Optimizer to use one that produces faster results.

Optimizer hints is a way to have better control over response times to meet your performance needs. Within a query, you can specify directives or *hints* to the Optimizer, which it then uses to determine its query execution plan. Hints are detected through a pseudo comment syntax from SQL2.

You can place a hint(s) in a SQL statement as a static string, just after a SELECT, INSERT, UPDATE, or DELETE keyword. The hint always follows the SQL statement that applies to it.

Table name resolution in optimizer hints is the same as in any table name in a SQL statement. When there is an error in a hint specification, then the whole SQL statement fails with an error message.

Hints are enabled and disabled using the following configuration parameter in the SOLID.INI:

```
[Hints]
EnableHints = YES | NO
```

The default is **YES**.

Example

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- JOIN ORDER FIXED *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- LOOP JOIN *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX REVERSE MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- SORT BEFORE GROUP BY *)--
AVG(I) FROM TAB1 WHERE I > 10 GROUP BY I2

SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INTERNAL SORT *)--
* FROM TAB1 WHERE I > 10 ORDER BY I2
```

INSERT

INSERT INTO *table_name* [(*column_identifier* [, *column_identifier*]...)]

VALUES (*insert_value*[, *insert_value*]...)

Usage

There are several variations of the `INSERT` statement. In the simplest instance, a value is provided for each column of the new row in the order specified at the time the table was defined (or altered). In the preferable form of the `INSERT` statement the columns are specified as part of the statement and they needn't to be in any specific order as long as the orders of the column and value lists match with one another.

Example

```
INSERT INTO TEST (C, ID) VALUES (0.22, 5);
INSERT INTO TEST VALUES (0.35, 9);
```

INSERT (Using Query)

INSERT INTO *table_name* [(*column_identifier* [, *column_identifier*]...)]

query_specification

Usage

The query specification creates a virtual table. Using the `INSERT` statement the rows of created virtual table are inserted into the specified table (the degree and data types of the virtual table and inserted columns must match).

Example

```
INSERT INTO TEST (C, ID) SELECT A, B FROM INPUT_TO_TEST;
```

REVOKE (Role from User)

REVOKE {*role_name* [, *role_name*]... }

FROM {PUBLIC **|** *user_name* [, *user_name*]... }

Usage

The `REVOKE` statement is used to take a role away from users.

Example

```
REVOKE GUEST_USERS FROM HOBBES;
```

REVOKE (Privilege from Role or User)

REVOKE

{ALL **|** *revoke_privilege* [, *revoke_privilege*]... } ON *table-name*

FROM {PUBLIC **|** *user_name* [, *user_name*]... **|** *role_name* [, *role_name*]... }

revoke-privilege ::= DELETE **|** INSERT **|** SELECT **|**

UPDATE [(*column_identifier* [, *column_identifier*]...)] **|**

REFERENCES

REVOKE EXECUTE ON *procedure_name*
    FROM {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

REVOKE {SELECT | INSERT} ON *event_name* FROM
    {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

REVOKE {SELECT | INSERT} ON *sequence_name*
    FROM {PUBLIC | *user_name* [, *user_name*]... | *role_name* [, *role_name*]... }

Note

Keywords CASCADE and RESTRICT are not supported in the SQL grammar of SOLID *SynchroNet*.

Usage

The `REVOKE` statement is used to take privileges away from users and roles.

Example

```
REVOKE INSERT ON TEST FROM GUEST_USERS;
```

ROLLBACK

ROLLBACK WORK

Usage

The changes made in the database are discarded by `ROLLBACK` statement. It terminates the transaction.

Example

```
ROLLBACK WORK;
```

SELECT

SELECT [ALL | DISTINCT] *select-list*

FROM *table_reference_list*

[WHERE *search_condition*]

[GROUP BY *column_name* [, *column_name*]...]

[HAVING *search_condition*]

[[UNION | INTERSECT | EXCEPT] [ALL] *select_statement*]...

[ORDER BY {*unsigned integer* | *column_name*}

[ASC | DESC]]

Usage

The SELECT statement is used to retrieve information.

!

Important

SOLID provides a consistent view of data within one transaction; that is, it sees the database as it was at the moment it was started. This is implemented by the multiversion SOLID Bonsai Tree that stores the active data, that is, data that has been written to the database since the beginning of the oldest active transaction in central memory. Also a `SELECT` begins a new transaction and if not committed or rolled back, it remains active thus causing the Bonsai Tree to grow.

New data is merged to the main storage tree as soon as no transaction needs to see the old versions of the rows. To ensure the efficient operation of the Bonsai Tree, also commit read-only transactions as soon as all rows are retrieved. This releases the read level and allows the merge process to keep the Bonsai Tree smaller.

Using `AUTOCOMMIT` does not help. This is because SOLID cannot immediately commit SELECTs since the rows need to be retrieved by the client application first. In `AUTOCOMMIT` mode, the next SQL statement processing triggers the commit for previous `SELECT` statement. But if that next statement never comes, the transaction is left open until the connection timeout expires.

Example

```
SELECT ID FROM TEST;

SELECT DISTINCT ID, C FROM TEST WHERE ID = 5;

SELECT DISTINCT ID FROM TEST ORDER BY ID ASC;

SELECT NAME, ADDRESS FROM CUSTOMERS UNION SELECT NAME, DEP FROM
PERSONNEL;
```

SET

SET CATALOG *catalog_name*

SET SQL INFO {ON | OFF} [FILE {*file_name* | "*file_name*" | '*file_name*'}]
    [LEVEL *info_level*]

SET SQL SORTARRAYSIZE {*array-size* | DEFAULT}

SET SQL JOINPATHSPAN {*path-span* | DEFAULT}

SET SQL CONVERTORSTOUNIONS
  {YES [COUNT *value*] | NO | DEFAULT}

SET OPTIMISTIC LOCK TIMEOUT *seconds*

SET LOCK TIMEOUT *timeout-in-seconds*

SET SCHEMA ’*schema_name*’ | ’*user_name*’

SET SCHEMA USER

SET STATEMENT MAXTIME *minutes*

SET TRANSACTION READ ONLY

SET TRANSACTION READ WRITE

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SET TRANSACTION ISOLATION LEVEL
    REPEATABLE READ

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Usage

All the settings are read per user session unlike the settings in the `solid.ini` file which are automatically read each time SOLID *SynchroNet* is started.

In `SQL INFO` the default file is a global `soltrace.out` shared by all users. If the file name is given, all future `INFO ON` settings will use that file unless a new file is set. It is recommended that the file name is given in single quotes, because otherwise the file name is converted to uppercase. The info output is appended to the file and the file is never truncated, so after the info file is not needed anymore, the user must manually delete the file. If the file open fails, the info output is silently discarded.

The default `SQL INFO LEVEL` is `4`. A good way to generate useful info output is to set info on with a new file name and then execute the SQL statement using `EXPLAIN PLAN FOR` syntax. This method gives all necessary estimator information but does not generate output from the fetches which may generate a huge output file.

The sort array is used for in memory sorts in the SQL interpreter. The minimum value for `SORTARRAYSIZE` is `100`. If a smaller value is given, minimum value 100 will be used. If large sorts are needed, it is recommended that the external sorter facility is used (in Sorter section in solid.ini) instead on using very large `SORTARRAYSIZE`.

The `COUNT` parameter in `SQL CONVERTORSTOUNIONS` tells how many ors are converted to unions. The default is 10 which should be enough in most cases.

SET CATALOG sets the catalog name context in a program.

SET OPTIMISTIC LOCK TIMEOUT sets lock time out separately for optimistic tables per connection. Note that when using SELECT FOR UPDATE, the selected rows are locked also for tables with optimistic concurrency control.

SET SCHEMA sets the schema name context when implicitly qualifying a database object name in a session. To remove the schema context, use the SET SCHEMA USER command. For details, read *"SET SCHEMA"* on page E-49.

SET STATEMENT MAXTIME sets connection specific maximum execution time in minutes. Setting is effective until a new maximum time if set. Zero time means no maximum time, which is also the default.

The SET TRANSACTION settings are borrowed from ANSI SQL. It sets the transaction isolation level. To detect conflicts between transactions, define the transaction with a Repeatable Read or Serializable isolation level using the SET TRANSACTION ISOLATION LEVEL command.

Example

```
SET SQL INFO ON FILE 'sqlinfo.txt' LEVEL 5
```

SET SCHEMA

SET SCHEMA {'*schema_name*' | USER | '*user_name*'}

Usage

SOLID *Embedded Engine* supports SQL89 style schemas for database entity name qualifying. All created database entities belong to a schema, and different schemas may contain entities with same name. In keeping with the ANSI SQL2 standard, the *user_name* or *schema_name* may be enclosed in single quotes.

The default schema can be changed with the SET SCHEMA statement. Schema can be changed to the current user name by using the SET SCHEMA USER statement. Alternatively schema can be set to '*user_name*' which must be a valid user name in the database.

The algorithm to resolve entity names [*schema_name*.]*table_identifier* is the following:

1. If *schema_name* is given then *table_identifier* is searched only from that schema.
2. If *schema_name* is not given, then
   a. First *table_identifier* is searched from default schema. Default schema is initially the same as user name, but can be changed with SET SCHEMA statement

b. Then *table_identifier* is searched from all schemas in the database. If more than one entity with same *table_identifier* and type (table, procedure, ...) is found, a new error code 13110 (Ambiguous entity name *table_identifier*) is returned.

The SET SCHEMA statement effects only to default entity name resolution and it does not change any access rights to database entities. It sets the default schema name for unqualified names in statements that are prepared in the current session by an execute immediate statement or a prepare statement.

Example

```
SET SCHEMA 'CUSTOMERS'
```

UPDATE (Positioned)

UPDATE *table_name*

SET [*table_name.*]*column_identifier* = {*expression* | NULL}

[, [*table_name.*]*column_identifier* = {*expression* | NULL}]...

WHERE CURRENT OF *cursor_name*

Usage

The positioned UPDATE statement updates the current row of the cursor. The name of the cursor is defined using ODBC API function named `SQLSetCursorName`.

Example

```
UPDATE TEST SET C = 0.33

WHERE CURRENT OF MYCURSOR
```

UPDATE (Searched)

UPDATE *table-name*

SET [*table_name.*]*column_identifier* = {*expression* | NULL}

[, [*table_name.*]*column_identifier* = {*expression* | NULL}]...

[WHERE *search_condition*]

Usage

The `UPDATE` statement is used to modify the values of one or more columns in one or more rows, according the search conditions.

Example

```
UPDATE TEST SET C = 0.44 WHERE ID = 5
```

Table_reference

Table_reference	
table_reference_list	::= *table_reference* [, *table-reference* ...]
table_reference	::= *table_name* [[AS] *correlation_name*] | *derived_table* [[AS] *correlation_name* [(*derived_column_list*)]] | *joined_table*
table_name	::= *table_identifier* | *schema_name.table_identifier*
derived_table	::= *subquery*
derived_column_list	::= *column_name_list*
joined_table	::= *cross_join* | *qualified_join* | (*joined_table*)
cross_join	::= *table_reference* CROSS JOIN *table_reference*
qualified_join	::= *table_reference* [NATURAL] [*join_type*] JOIN *table_reference* [*join_specification*]
join_type	::= INNER | *outer_join_type* [OUTER] | UNION
outer_join_type	::= LEFT | RIGHT | FULL
join_specification	::= *join_condition* | *named_columns_join*
join_condition	::= ON *search_condition*
named_columns_join	::= USING (*column_name_list*)
column_name_list	::= *column_identifier* [{ , *column_identifier*} ...]

Query_specification

Query_specification	
query_specification	::= SELECT [DISTINCT | ALL] *select_list* *table_expression*
select_list	::= * | *select_sublist* [{, *select_sublist*} ...]
select_sublist	::= *derived_column* | [*table_name* | *table_identifier*].*
derived_column	::= *expression* [[AS] *column_alias*]]
table_expression	::= FROM *table_reference_list* [WHERE *search_condition*] [GROUP BY *column_name_list* [[UNION | INTERSECT | EXCEPT] [ALL] [CORRE-SPONDING [BY (*column_name_list*)]] *query_specification*] [HAVING *search_condition*]

Search_condition

Search_condition	
search_condition	::= *search_item* | *search_item* { AND | OR } *search_item*
search_item	::= [NOT] { *search_test* | (*search_condition*) }
search_test	::= *comparison_test* | *between_test* | *like_test* | *null_test* | *set_test* | *quantified_test* | *existence_test*
comparison_test	::= *expression* { = | <> | < | <= | > | >= } { *expression* | *subquery* }
between_test	::= *column_identifier* [NOT] BETWEEN *expression* AND *expression*
like_test	::= *column_identifier* [NOT] LIKE *value* [ESCAPE *value*]
null_test	::= *column_identifier* IS [NOT] NULL

Search_condition	
set_test	::= *expression* [NOT] IN ({ *value* [,*value*]... | *subquery* })
quantified_test	::= *expression* { = | <> | < | <= | > | >= } [ALL | ANY | SOME] *subquery*
existence_test	::= EXISTS *subquery*

Check_condition

Check_condition	
check_condition	::= *check_item* | *check_item* { AND | OR } *check_item*
check_item	::= [NOT] { *check_test* | (*check_condition*) }
check_test	::= *comparison_test* | *between_test* | l*ike_test* | *null_test* | *list_test*
comparison_test	::= *expression* { = | <> | < | <= | > | >= } { *expression* | *subquery* }
between_test	::= *column_identifier* [NOT] BETWEEN *expression* AND *expression*
like_test	::= *column_identifier* [NOT] LIKE *value* [ESCAPE *value*]
null_test	::= *column_identifier* IS [NOT] NULL
l*ist_test*	::= *expression* [NOT] IN ({ *value* [,*value*]...})

Expression

Expression	
expression	::= *expression_item* | *expression_item* { + | _ | * | / } *expression_item*
expression_item	::= [+ | _] { *value* | *column_identifier* | *function* | *case_expression* | *cast_expression* | (*expression*) }

Expression	
value	::= *literal* | USER | *variable*
function	::= *set_function* | *null_function* | *string_function* | *numeric_function* | *datetime_function* | *system_function* | *datatypeconversion_function*
set_function	::= COUNT (*) | { AVG | MAX | MIN | SUM | COUNT } ({ ALL | DISTINCT } *expression*)
null_function	::= { NULLVAL_CHAR( ) | NULLVAL_INT( ) }
datatypeconversion_function	::= CONVERT_CHAR(*value_exp*) | CONVERT_DATE(*value_exp*) | CONVERT_DECIMAL(*value_exp*) | CONVERT_DOUBLE(*value_exp*) | CONVERT_FLOAT(*value_exp*) | CONVERT_INTEGER(*value_exp*) | CONVERT_LONGVARCHAR(*value_exp*) | CONVERT_NUMERIC(*value_exp*) | CONVERT_REAL(*value_exp*) | CONVERT_SMALLINT(*value_exp*) | CONVERT_TIME(*value_exp*) | CONVERT_TIMESTAMP(*value_exp*) | CONVERT_TINYINT(*value_exp*) | CONVERT_VARCHAR(*value_exp*)
case_expression	::= *case_abbreviation* | *case_specification*
case_abbreviation	::= NULLIF(*value_exp*, *value_exp*) | COALESCE(*value_exp* {, *value_exp*}…)
case_specification	::= CASE [*value_exp*] WHEN *value_exp* THEN {*value_exp*} [WHEN *value_exp* THEN {*value_exp*} …] [ELSE {*value_exp*}] END
cast_expression	::= CAST (*value-exp* AS -*data-type*)

String Function

Function	**Purpose**
ASCII(*str*)	Returns the integer equivalent of string *str*
CHAR(*code*)	Returns the character equivalent of code
CONCAT(*str1*, *str2*)	Concatenates *str2* to *str1*
str1 { + | || } *str2*	Concatenates *str2* to *str1*
INSERT(*str1*, *start*, *length*, *str2*)	Merges strings by deleting length characters from *str1* and inserting *str2*
LCASE(*str*)	Converts string *str* to lowercase
LEFT(*str*, *count*)	Returns leftmost count characters of string *str*
LENGTH(*str*)	Returns the number of characters in *str*
LOCATE(*str1*, *str2* [, *start*])	Returns starting position of *str1* within *str2*
LTRIM(*str*)	Removes leading spaces of *str*
POSITION (*str1* IN *str2*)	Returns starting position of *str1* within *str2*
REPEAT(*str, count*)	Returns characters of *str* repeated count times
REPLACE(*str1*, *str2*, *str3*)	Replaces occurrences of *str2* in *str1* with *str3*
RIGHT(*str, count*)	Returns the rightmost count characters of string *str*
RTRIM(*str*)	Removes trailing spaces in *str*
SPACE(*count*)	Returns a string of count spaces
SUBSTRING(*str*, *start*, *length*)	Derives substring from *str* beginning at *start*
UCASE(*str*)	Converts *str* to uppercase

Numeric Function

Function	**Purpose**
ABS(*numeric*)	Absolute value of *numeric*
ACOS(*float*)	Arccosine of *float*
ASIN(*float*)	Arcsine of *float*

Function	**Purpose**
ATAN(*float*)	Arctangent of *float*
ATAN2(*float1*, *float2*)	Arctangent of the *x* and *y* coordinates, specified by *float1* and *float2*, respectively, as an angle, expressed in radians
CEILING(*numeric*)	Smallest integer greater than or equal to *numeric*
COS(*float*)	Cosine of *float*
COT(*float*)	Cotangent of *float*
DEGREES(*numeric*)	Converts *numeric* radians to degrees
EXP(*float*)	Exponential value of *float*
FLOOR(*numeric*)	Largest integer less than or equal to *numeric*
LOG(*float*)	Natural logarithm of *float*
LOG10(*float*)	Base 10 log of *float*
MOD(*integer1*, *integer2*)	Modulus of *integer1* divided by *integer2*
PI()	Pi as a floating point number
POWER(*numeric*, *integer*)	Value of *numeric* raised to the power of *integer*
RADIANS(*numeric*)	Number of radians converted from *numeric*
ROUND(*numeric*, *integer*)	*Numeric* rounded to *integer*
SIGN(*numeric*)	Sign of *numeric*
SQRT(*float*)	Square root of *float*
TAN(*float*)	Tangent of *float*
TRUNCATE(*numeric*, *integer*)	*Numeric* truncated to *integer*

Date Time Function

Function	**Purpose**
CURDATE()	Returns the current date
CURTIME()	Returns the current time
DAYNAME(*date*)	Returns a string with the day of the week
DAYOFMONTH(*date*)	Returns the day of the month as an integer between 1 and 31

Function	Purpose
DAYOFWEEK(*date*)	Returns the day of the week as an integer between 1 and 7, where 1 represents Sunday
DAYOFYEAR(*date*)	Returns the day of the year as an integer between 1 and 366
EXTRACT (*date field* FROM *date_exp*)	Isolates a single field of a datetime or a interval and converts it to a number.
HOUR(*time_exp*)	Returns the hour as an integer between 0 and 23
MINUTE(*time_exp*)	Returns the minute as an integer between 0 and 59
MONTH(*date*)	Returns the month as an integer between 1 and 12
MONTHNAME(*date*)	Returns the month name as a string
NOW()	Returns the current date and time as a timestamp
QUARTER(*date*)	Returns the quarter as an integer between 1 and 4
SECOND(*time_exp*)	Returns the second as an integer between 0 and 59
TIMESTAMPADD(interval, *integer_exp*, timestamp_exp)	Calculates a timetamp by adding *integer_exp* intervals of type interval to *timestamp_exp* Keywords used to express valid TIMESTAMPADD interval values are: SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR

Function	**Purpose**
TIMESTAMPDIFF(interval, *timestamp-exp1*, *timestamp-exp2*)	Returns the integer number of intervals by which *timestamp-exp2* is greater than *timestamp-exp1* Keywords used to express valid TIMESTAMPDIFF interval values are: SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR
WEEK(*date*)	Returns the week of the year as an integer between 1 and 52
YEAR(*date*)	Returns the year as an integer

System Function

Function	**Purpose**
IFNULL(*exp*, *value*)	If *exp* is null, returns value; if not, returns exp
USER()	Returns the user authorization name
UIC()	Returns the connection id associated with the connection

Data_type

Data_type	
data_type	::= {BINARY | CHAR [*length*] | DATE | DECIMAL [(*precision* [, *scale*])] | DOUBLE PRECISION | FLOAT [(*precision*)] | INTEGER | LONG VARBINARY | LONG VARCHAR | LONG WVARCHAR | NUMERIC [(*precision* [, *scale*])] | REAL | SMALLINT | TIME | TIMESTAMP [(*timestamp precision*)] | TINYINT | VARBINARY | VARCHAR [(*length*)] } | WCHAR | WVARCHAR [*length*]

Date and Time Literals

Date/time literal	
date_literal	´YYYY-MM-DD´
time_literal	´HH:MM:SS´
timestamp_literal	´YYYY-MM-DD HH:MM:SS´

Pseudo Columns

The following pseudo columns may also be used in the select-list of a SELECT statement:

Pseudo column	**Type**	**Explanation**
ROWVER	VARBINARY(254)	Version of the row in a table.
ROWID	VARBINARY(10)	Persistent id for a row in a table.

Pseudo column	Type	Explanation
ROWNUM	DECIMAL(16,2)	Row number indicates the sequence in which a row was selected from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second row has 2, etc. ROWNUM is chiefly useful for limiting the number of rows returned by a query for example, WHERE ROWNUM < 10).

▶ **Note**

Since ROWID and ROWVER refer to a single row, they may only be used with queries that return rows from a single table.

F

Database System Views and System Tables

System Views

SOLID *SynchroNet* supports views as specified in the X/Open SQL Standard.

COLUMNS

The COLUMNS system view identifies the columns which are accessible to the current user.

Column name	Data type	Description
TABLE_CATALOG	WVARCHAR	the name of the catalog containing TABLE_NAME
TABLE_SCHEMA	WVARCHAR	the name of the schema containing TABLE_NAME
TABLE_NAME	WVARCHAR	the name of the table or view
COLUMN_NAME	WVARCHAR	the name of the column of the specified table or view
DATA_TYPE	WVARCHAR	the data type of the column
SQL_DATA_TYPE_NUM	SMALLINT	ODBC compliant data type number
CHAR_MAX_LENGTH	INTEGER	maximum length for a character data type column; for others NULL

Column name	Data type	Description
NUMERIC_PRECISION	INTEGER	the number of digits of mantissa precision of the column, if DATA_TYPE is approximate numeric data type, NUMERIC_PREC_RADIX indicates the units of measurement; for other numeric types contains the total number of decimal digits allowed in the column; for character data types NULL
NUMERIC_PREC_RADIX	SMALLINT	the radix of numeric precision if DATA_TYPE is one of the approximate numeric data types; otherwise NULL
NUMERIC_SCALE	SMALLINT	total number of significant digits to the right of the decimal point; for INTEGER and SMALLINT 0; for others NULL
NULLABLE	CHAR	if column is known to be not nullable 'NO'; otherwise 'YES'
NULLABLE_ODBC	SMALLINT	ODBC, if column is known to be not nullable '0'; otherwise '1'
REMARKS	LONG WVARCHAR	reserved for future use

SERVER_INFO

The SERVER_INFO system view provides attributes of the current database system or server.

Column name	Data type	Description
SERVER_ATTRIBUTE	WVARCHAR	identifies an attribute of the server
ATTRIBUTE_VALUE	WVARCHAR	the value of the attribute

TABLES

The TABLES system view identifies the tables accessible to the current user.

Column name	Data type	Description
TABLE_CATALOG	WVARCHAR	the name of the catalog containing TABLE_NAME
TABLE_SCHEMA	WVARCHAR	the name of the schema containing TABLE_NAME
TABLE_NAME	WVARCHAR	the name of the table or view
TABLE_TYPE	WVARCHAR	the type of the table
REMARKS	LONG WVARCHAR	reserved for future use

USERS

The USERS system view identifies users and roles.

Column name	Data type	Description
ID	INTEGER	User or role id
NAME	WVARCHAR	User or role name
TYPE	WVARCHAR	User type, either USER or ROLE
PRIV	INTEGER	Privilege information
PRIORITY	INTEGER	Reserved for future use
PRIVATE	INTEGER	Specifies whether user is private or public

System Tables

SQL_LANGUAGES

The SQL_LANGUAGES system table lists the SQL standards and SQL dialects which are supported.

Column name	Data type	Description
SOURCE	WVARCHAR	the organization that defined this specific SQL version
SOURCE_YEAR	WVARCHAR	the year the relevant standard was approved
CONFORMANCE	WVARCHAR	the conformance level at which conformance to the relevant standard
INTEGRITY	WVARCHAR	indicates whether the Integrity Enhancement Feature is supported
IMPLEMENTATION	WVARCHAR	identifies uniquely the vendor's SQL language; NULL if SOURCE is 'ISO'
BINDING_STYLE	WVARCHAR	the binding style 'DIRECT', *EMBED' or 'MODULE'
PROGRAMMING_LANG	WVARCHAR	the host language used

SYS_ATTAUTH

Column name	Data type	Description
REL_ID	INTEGER	table id
UR_ID	INTEGER	user or role id
ATTR_ID	INTEGER	column id
PRIV	INTEGER	privilege info
GRANT_ID	INTEGER	grantor id
GRANT_TIM	TIMESTAMP	grant time

SYS_CARDINAL

Column name	Data type	Description
REL_ID	INTEGER	the relation id as in SYS_TABLES
CARDIN	INTEGER	the number of rows in the table
SIZE	INTEGER	the size of the data in the table
LAST_UPD	TIMESTAMP	the timestamp of the last update in the table

SYS_CATALOGS

The SYS_CATALOGS lists available catalogs.

Column name	Data type	Description
ID	INTEGER	catalog id
NAME	WVARCHAR	catalog name
CREATIME	TIMESTAMP	create date and time
CREATOR	WVARCHAR	creator name

SYS_COLUMNS

Column name	Data type	Description
ID	INTEGER	unique column identifier
REL_ID	INTEGER	the relation id as in SYS_TABLES
COLUMN_NAME	WVARCHAR	the name of the column
COLUMN_NUMBER	INTEGER	the number of the column in the table (in creation order)
DATA_TYPE	WVARCHAR	the data type of the column
SQL_DATA_TYPE_NUM	SMALLINT	ODBC compliant data type number
DATA_TYPE_NUMBER	INTEGER	internal data type number
CHAR_MAX_LENGTH	INTEGER	maximum length for a CHAR field

Column name	Data type	Description
NUMERIC_PRECISION	INTEGER	numeric precision
NUMERIC_PREC_RADIX	SMALLINT	numeric precision radix
NUMERIC_SCALE	SMALLINT	numeric scale
NULLABLE	CHAR	are NULL values allowed (Yes, No)
NULLABLE_ODBC	SMALLINT	ODBC, are NULL values allowed (1,0)
FORMAT	WVARCHAR	reserved for future use
DEFAULT_VAL	VARBINARY	reserved for future use
ATTR_TYPE	INTEGER	user defined (0) or internal (>0)
REMARKS	LONG WVARCHAR	reserved for future use

SYS_EVENTS

Column name	Data type	Description
ID	INTEGER	unique event identifier
EVENT_NAME	WVARCHAR	the name of the event
EVENT_PARAMCOUNT	INTEGER	number of parameters
EVENT_PARAMTYPES	LONG VARBINARY	types of parameters
EVENT_TEXT	WVARCHAR	the body of the event
EVENT_SCHEMA	WVARCHAR	the owner of the event
EVENT_CATALOG	WVARCHAR	the owner of the event
CREATIME	TIMESTAMP	creation time
TYPE	INTEGER	reserved for future use

SYS_FORKEYPARTS

Column name	Data type	Description
KEY_CATALOG	INTEGER	creator name or the owner of the key

ID	INTEGER	foreign key identifier
KEYP_NO	INTEGER	keypart number
ATTR_NO	INTEGER	column number
ATTR_ID	INTEGER	column identifier
ATTR_TYPE	INTEGER	column type
CONST_VALUE	VARBINARY	possible internal constant value; otherwise NULL

SYS_FORKEYS

Column name	Data type	Description
ID	INTEGER	foreign key identifier
REF_REL_ID	INTEGER	referenced table identifier
CREATE_REL_ID	INTEGER	creator table identifier
REF_KEY_ID	INTEGER	referenced key identifier
REF_TYPE	INTEGER	reference type
KEY_SCHEMA	WVARCHAR	creator name
KEY_CATALOG	WVARCHAR	creator name or the owner of the key
KEY_NREF	INTEGER	number of referenced key parts

SYS_INFO

Column name	Data type	Description
PROPERTY	WVARCHAR	the name of the property
VALUE_STR	WVARCHAR	value as a string
VALUE_INT	INTEGER	value as an integer

SYS_KEYPARTS

Column name	Data type	Description
ID	INTEGER	unique key identifier
REL_ID	INTEGER	the relation id as in SYS_TABLES
KEYP_NO	INTEGER	keypart identifier
ATTR_ID	INTEGER	column identifier
ATTR_NO	INTEGER	the number of the column in the table (in creation order)
ATTR_TYPE	INTEGER	the type of the column
CONST_VALUE	VARBINARY	constant value or NULL
ASCENDING	CHAR	is the key ascending (Yes) or descending (No)

SYS_KEYS

Column name	Data type	Description
ID	INTEGER	unique key identifier
REL_ID	INTEGER	the relation id as in SYS_TABLES
KEY_NAME	WVARCHAR	the name of the key
KEY_UNIQUE	CHAR	is the key unique (Yes, No)
KEY_NONUNIQUE_ODBC	SMALLINT	ODBC, is the key NOT unique (1, 0)
KEY_CLUSTERING	CHAR	is the key a clustering key (Yes, No)
KEY_PRIMARY	CHAR	is the key a primary key (Yes, No)
KEY_PREJOINED	CHAR	reserved for future use
KEY_SCHEMA	WVARCHAR	the owner of the key
KEY_NREF	INTEGER	internal system specific information

SYS_PROCEDURES

Column name	Data type	Description
ID	INTEGER	unique procedure identifier
PROCEDURE_NAME	WVARCHAR	procedure name
PROCEDURE_TEXT	LONG WVARCHAR	procedure body
PROCEDURE_BIN	LONG VARBINARY	compiled form of the procedure
PROCEDURE_SCHEMA	WVARCHAR	the name of the schema containing PROCEDURE_NAME
PROCEDURE_CATALOG	WVARCHAR	the name of the catalog containing PROCEDURE_NAME
CREATIME	TIMESTAMP	creation time
TYPE	INTEGER	reserved for future use

SYS_PROCEDURE_COLUMNS

The SYS_PROCEDURE_COLUMNS defines input parameters and result set columns.

Column name	Data type	Description
PROCEDURE_ID	INTEGER	procedure id
COLUMN_NAME	WVARCHAR	procedure column name
COLUMN_TYPE	SMALLINT	procedure column type (SQL_PARAM_INPUT or SQL_RESULT_COL)
DATA_TYPE	SMALLINT	column’s SQL data type
TYPE_NAME	WVARCHAR	column’s SQL data type name
COLUMN_SIZE	INTEGER	size of the procedure column
BUFFER_LENGTH	INTEGER	column size in bytes
DECIMAL_DIGITS	SMALLINT	decimal digits of the procedure column
NUM_PREC_RADIX	SMALLINT	radix for numeric data types (2, 10, or NULL if not applicable)

NULLABLE	SMALLINT	whether the procedure column accepts a NULL value
REMARKS	WVARCHAR	a description of the procedure column
COLUMN_DEF	WVARCHAR	column's default value. Always NULL, that is, no default value is specified.
SQL_DATA_TYPE	SMALLINT	SQL data type
SQL_DATETIME_SUB	SMALLINT	subtype code for datetime. Always NULL.
CHAR_OCTET_LENGTH	INTEGER	maximum length in bytes of a character or binary data type column.
ORDINAL_POSITION	INTEGER	ordinal position of the column
IS_NULLABLE	WVARCHAR	always "YES"

SYS_RELAUTH

Column name	Data type	Description
REL_ID	INTEGER	relation id
UR_ID	INTEGER	user or role id
PRIV	INTEGER	privilege info
GRANT_ID	INTEGER	grantor id
GRANT_TIM	TIMESTAMP	grant time
GRANT_OPT	CHAR	grant option info

SYS_SCHEMAS

The SYS_SCHEMAS lists available schemas.

Column name	Data type	Description
ID	INTEGER	schema id
NAME	WVARCHAR	schema name
OWNER	WVARCHAR	schema owner name

CREATIME	TIMESTAMP	create date and time
SCHEMA_CATALOG	WVARCHAR	schema catalog

SYS_SEQUENCES

Column name	**Data type**	**Description**
SEQUENCE_NAME	WVARCHAR	sequence name
ID	INTEGER	unique id
DENSE	CHAR	is the sequence dense or sparse
SEQUENCE_SCHEMA	WVARCHAR	the name of the schema containing SEQUENCE_NAME
SEQUENCE_CATALOG	WVARCHAR	the name of the catalog containing SEQUENCE_NAME
CREATIME	TIMESTAMP	creation time

SYS_SYNONYM

Column name	**Data type**	**Description**
TARGET_ID	INTEGER	reserved for future use
SYNON	INTEGER	reserved for future use

SYS_TABLEMODES

Column name	**Data type**	**Description**
ID	INTEGER	relation id
MODE	WVARCHAR	special mode info
MODIFY_TIME	TIMESTAMP	last modify time
MODIFY_USER	WVARCHAR	last user that modified

SYS_TABLES

Column name	Data type	Description
ID	INTEGER	unique table identifier
TABLE_NAME	WVARCHAR	the name of the table
TABLE_TYPE	WVARCHAR	the type of the table (BASE TABLE or VIEW)
TABLE_SCHEMA	WVARCHAR	the name of the catalog containing TABLE_NAME
TABLE_CATALOG	WVARCHAR	the name of the catalog containing TABLE_NAME
CREATIME	TIMESTAMP	the creation time of the table
CHECKSTRING	LONG WVARCHAR	possible check option defined for the table
REMARKS	LONG WVARCHAR	reserved for future use

SYS_TRIGGERS

Column name	Data type	Description
ID	INTEGER	unique table identifier
TRIGGER_NAME	WVARCHAR	trigger name
TRIGGER_TEXT	LONG WVARCHAR	trigger body
TRIGGER_BIN	LONG VARBINARY	compiled form of the trigger
TRIGGER_SCHEMA	WVARCHAR	the name of the schema containing TRIGGER_NAME
TRIGGER_CATALOG	WVARCHAR	the name of the catalog containing TRIGGER_NAME
TRIGGER_ENABLED	CHAR	if triggers are enabled "YES"; otherwise "NO."
CREATIME	TIMESTAMP	the creation time of the trigger
TYPE	INTEGER	reserved for future use

REL_ID	INTEGER	the relation id

SYS_TYPES

Column name	**Data type**	**Description**
TYPE_NAME	WVARCHAR	the name of the data type
DATA_TYPE	SMALLINT	ODBC, data type number
PRECISION	INTEGER	ODBC, the precision of the data type
LITERAL_PREFIX	WVARCHAR	ODBC, possible prefix for literal values
LITERAL_SUFFIX	WVARCHAR	ODBC, possible suffix for literal values
CREATE_PARAMS	WVARCHAR	ODBC, the parameters needed to create a column of the data type
NULLABLE	SMALLINT	ODBC, can the data type contain NULL values
CASE_SENSITIVE	SMALLINT	ODBC, is the data type case sensitive
SEARCHABLE	SMALLINT	ODBC, the supported search operations
UNSIGNED_ATTRIBUTE	SMALLINT	ODBC, is the data type unsigned
MONEY	SMALLINT	ODBC, whether the data is a money data type
AUTO_INCREMENT	SMALLINT	ODBC, whether the data type is autoincrementing
LOCAL_TYPE_NAME	WVARCHAR	ODBC, has the data type another implementation defined name
MINIMUM_SCALE	SMALLINT	ODBC, the minimum scale of the data type
MAXIMUM_SCALE	SMALLINT	ODBC, the maximum scale of the data type

SYS_UROLE

The SYS_UROLE contains mapping of users to roles.

Column name	Data type	Description
U_ID	INTEGER	User id
R_ID	INTEGER	Role id

SYS_USERS

The SYS_USERS list information about users and roles.

Column name	Data type	Description
ID	INTEGER	User or role id
NAME	WVARCHAR	User or role name
TYPE	WVARCHAR	User type, either USER or ROLE
PRIV	INTEGER	Privilege information
PASSW	VARBINARY	Password inencrypted format
PRIORITY	INTEGER	Reserved for future use
PRIVATE	INTEGER	Specifies whether user is private or public
LOGIN_CATALOG	WVARCHAR	Reserved for future use

SYS_VIEWS

Column name	Data type	Description
V_ID	INTEGER	unique identifier for this view
TEXT	LONG WVARCHAR	view definition
CHECKSTRING	LONG WVARCHAR	possible CHECK OPTION defined for the view
REMARKS	LONG WVARCHAR	reserved for future use

G

Reserved Words

The following words are reserved in several SQL standards: ODBC 3.0, X/Open and SQL Access Group SQL CAE specification, Database Language - SQL: ANSI X3H2 (SQL-92). Some words are used by SOLID SQL. Applications should avoid using any of these keywords for other purposes. The following table contains also potential reserved words; these markings are enclosed in parenthesis.

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
ABSOLUTE	•		•	
ACTION	•		•	
ADA	•			
ADD	•	•	•	•
ADMIN				•
AFTER			(•)	•
ALIAS			(•)	
ALL	•	•	•	•
ALLOCATE	•	•	•	
ALTER	•	•	•	•
AND	•	•	•	•
ANY	•	•	•	•
APPEND				•
ARE	•		•	
AS	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
ASC	•	•	•	•
ASSERTION	•		•	
ASYNC			(•)	•
AT	•		•	
AUTHORIZATION	•		•	•
AVG	•	•	•	
BEFORE			(•)	•
BEGIN	•	•	•	•
BETWEEN	•	•	•	•
BINARY				•
BIT	•		•	
BIT_LENGTH	•		•	
BOOKMARK				•
BOOLEAN			(•)	
BOTH	•		•	
BREADTH			(•)	
BY	•	•	•	•
CALL			(•)	•
CASCADE	•	•	•	•
CASCADED	•		•	•
CASE	•		•	•
CAST	•		•	•
CATALOG	•		•	•
CHAR	•	•	•	•
CHAR_LENGTH	•		•	
CHARACTER	•	•	•	
CHARACTER_LENGTH	•		•	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
CHECK	•	•	•	•
CLOSE	•	•	•	•
COALESCE	•		•	•
COLLATE	•		•	
COLLATION	•		•	
COLUMN	•		•	•
COMMIT	•	•	•	•
COMMITBLOCK				•
COMMITTED				•
COMPLETION			(•)	
CONNECT	•	•	•	
CONNECTION	•	•	•	
CONSTRAINT	•		•	•
CONSTRAINTS	•		•	
CONTINUE	•	•	•	
CONVERT	•		•	
CORRESPONDING	•		•	•
COUNT	•	•	•	
CREATE	•	•	•	•
CROSS	•		•	•
CURRENT	•	•		•
CURRENT_DATE	•		•	
CURRENT_TIME	•		•	
CURRENT_TIMESTAMP	•		•	
CURRENT_USER	•		•	
CURSOR	•	•	•	•
CYCLE			(•)	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
DATA			(•)	•
DATE	•		•	•
DAY	•		•	
DEALLOCATE	•	•	•	
DEC	•	•	•	•
DECIMAL	•	•	•	•
DECLARE	•	•	•	•
DEFAULT	•	•	•	•
DEFERRABLE	•		•	
DEFERRED	•		•	
DELETE	•	•	•	•
DENSE				•
DEPTH			(•)	
DESC	•	•	•	•
DESCRIBE	•	•	•	
DESCRIPTOR	•	•	•	
DIAGNOSTICS	•	•	•	
DICTIONARY			(•)	
DISCONNECT	•	•	•	
DISTINCT	•	•	•	•
DOMAIN	•		•	•
DOUBLE	•	•	•	•
DROP	•	•	•	•
EACH			(•)	
ELSE	•		•	•
ELSEIF			(•)	•
ENABLE				•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
END	•	•	•	•
END-EXEC	•		•	
EQUALS			(•)	
ESCAPE	•		•	•
EVENT				•
EXCEPT	•		•	•
EXCEPTION	•	•	•	
EXEC	•	•	•	•
EXECUTE	•	•	•	•
EXISTS	•	•	•	•
EXPLAIN				•
EXPORT				•
EXTERNAL	•		•	•
EXTRACT	•		•	•
FALSE	•		•	
FETCH	•	•	•	•
FIRST	•		•	
FIXED				•
FLOAT	•	•	•	•
FOR	•	•	•	•
FOREIGN	•	•	•	•
FOREVER				•
FORTRAN	•			
FORWARD				•
FOUND	•	•	•	
FROM	•	•	•	•
FROMFIXED				•
FULL	•		•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
GENERAL			(•)	
GET	•	•	•	•
GLOBAL	•		•	
GO	•		•	
GOTO	•	•	•	
GRANT	•	•	•	•
GROUP	•	•	•	•
HAVING	•	•	•	•
HINT				•
HOUR	•		•	
IDENTIFIED				•
IDENTITY	•		•	
IF			(•)	•
IGNORE	•		(•)	
IMMEDIATE	•	•	•	
IMPORT				•
IN	•	•	•	•
INCLUDE	•	•		
INDEX	•	•		•
INDICATOR	•		•	
INITIALLY	•		•	
INNER	•		•	•
INPUT	•		•	
INSENSITIVE	•		•	
INSERT	•	•	•	•
INT	•	•	•	•
INTEGER	•	•	•	•
INTERNAL				•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
INTERSECT	•		•	•
INTERVAL	•		•	
INTO	•	•	•	•
IS	•	•	•	•
ISOLATION	•		•	•
JAVA				•
JOIN	•		•	•
KEY	•	•	•	•
LANGUAGE	•		•	
LAST	•		•	
LEADING	•		•	
LEAVE			(•)	•
LEFT	•		•	•
LESS			(•)	
LEVEL	•		•	•
LIKE	•	•	•	•
LIMIT			(•)	
LOCAL	•		•	•
LOCK				•
LONG				•
LOOP			(•)	•
LOWER	•		•	
MAINMEMORY				•
MASTER				•
MATCH	•		•	
MAX	•	•	•	
MERGE				•
MESSAGE				•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
MIN	•	•	•	
MINUTE	•		•	
MODIFY			(•)	•
MODULE	•		•	
MONTH	•		•	
NAMES	•		•	
NATIONAL	•		•	
NATURAL	•		•	•
NCHAR	•		•	
NEW			(•)	•
NEXT	•		•	•
NO	•		•	•
NONE	•		(•)	
NOT	•	•	•	•
NULL	•	•	•	•
NULLIF	•		•	•
NUMERIC	•	•	•	•
OBJECT			(•)	
OCTET_LENGTH	•		•	
OF	•	•	•	•
OFF	•		(•)	
OID			(•)	
OLD			(•)	•
ON	•	•	•	•
ONLY	•		•	•
OPEN	•	•	•	
OPERATION			(•)	
OPERATORS			(•)	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
OPTIMISTIC				•
OPTION	•		•	•
OR	•	•	•	•
ORDER	•	•	•	•
OTHERS			(•)	
OUTER	•		•	•
OUTPUT	•		•	
OVERLAPS	•		•	
PARAMETERS			(•)	
PARTIAL	•		•	
PASCAL	•			
PENDANT			(•)	
PESSIMISTIC				•
PLAN				•
PLI	•			
POSITION	•		•	
POST				•
PRECISION	•	•	•	•
PREORDER			(•)	
PREPARE	•	•	•	•
PRESERVE	•		•	
PRIMARY	•	•	•	•
PRIOR	•		•	
PRIVATE			(•)	
PRIVILEGES	•		•	•
PROCEDURE	•		•	•
PROPAGATE				•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
PROTECTED			(•)	
PUBLIC	•	•	•	•
PUBLICATION				•
READ			•	•
REAL		•	•	•
RECURSIVE			(•)	
REF			(•)	
REFERENCES	•	•	•	•
REFERENCING			(•)	•
REGISTER				•
RELATIVE	•		•	
RENAME				•
REPEATABLE				•
REPLACE			(•)	
REPLICA				•
REPLY				•
RESIGNAL			(•)	
RESTART				•
RESTRICT	•	•	•	•
RESULT				•
RETURN			(•)	•
RETURNS			(•)	•
REVERSE				•
REVOKE	•	•	•	•
RIGHT	•		•	•
ROLE			(•)	•
ROLLBACK	•	•	•	•
ROUTINE			(•)	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
ROW			(•)	
ROWID				•
ROWNUM				•
ROWSPERMESSAGE				•
ROWVER				•
ROWS	•		•	
SAVEPOINT			(•)	•
SCAN				•
SCHEMA	•		•	•
SCROLL	•		•	
SEARCH			(•)	
SECOND	•		•	
SECTION	•	•	•	
SELECT	•	•	•	•
SENSITIVE			(•)	
SEQUENCE			(•)	•
SERIALIZABLE				•
SESSION	•		•	
SESSION_USER	•		•	
SET	•	•	•	•
SIGNAL			(•)	
SIMILAR			(•)	
SIZE	•		•	
SMALLINT	•	•	•	•
SOME	•		•	•
SORT				•
SPACE	•			
SQL	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
SQLCA	•	•		
SQLCODE	•		•	
SQLERROR	•	•	•	•
SQLEXCEPTION			(•)	
SQLSTATE	•		•	
SQLWARNING	•		(•)	
START				•
STRUCTURE			(•)	
SUBSCRIBE				•
SUBSCRIPTION				•
SUBSTRING	•		•	
SUM	•	•	•	
SYNC_CONFIG				•
SYSTEM	•			
SYSTEM_USER			•	
TABLE	•	•	•	•
TEMPORARY	•		•	
TEST			(•)	
THEN	•		•	•
THERE			(•)	
TIME	•		•	•
TIMEOUT				•
TIMESTAMP	•		•	•
TIMEZONE_HOUR	•		•	
TIMEZONE_MINUTE	•		•	
TINYINT				•
TO	•	•	•	•
TRAILING			•	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
TRANSACTION	•		•	•
TRANSACTIONS				•
TRANSLATE	•		•	
TRANSLATION	•		•	
TRIGGER			(•)	•
TRIM	•		•	
TRUE	•		•	
TYPE			(•)	
UNDER			(•)	
UNION	•	•	•	•
UNIQUE	•	•	•	•
UNKNOWN	•		•	
UNREGISTER				•
UPDATE	•	•	•	•
UPPER	•		•	
USAGE	•		•	
USER	•	•	•	•
USING	•	•	•	•
VALUE	•	•	•	•
VALUES	•	•	•	•
VARBINARY				•
VARCHAR	•	•	•	•
VARIABLE			(•)	
VARWCHAR				•
VARYING	•	•	•	
VIEW	•	•	•	•
VIRTUAL			(•)	
VISIBLE			(•)	

Reserved word	ODBC	X/Open SQL	ANSI SQL2	SOLID SQL
WAIT			(•)	•
WCHAR				•
WHEN	•		•	•
WHENEVER	•	•	•	
WHERE	•	•	•	•
WHILE			(•)	•
WITH	•	•	•	•
WITHOUT			(•)	
WORK	•	•	•	•
WRITE			•	•
WCHAR				•
WVARCHAR				•
YEAR	•		•	
ZONE			•	

H

SOLID *SynchroNet* Command Line Options

General Options

Option	Description	Examples
-c*dir*	Changes working directory.	
-f	Starts server in foreground.	
-h	Displays help.	
-m	Monitors users' messages and SQL statements.	
-n*name*	Sets server name.	

--**s**{**start** | **install** | **remove**}, *name*, *fullexepath*, [**autostart**]	The Windows NT version of SOLID *SynchroNet* is by default an icon exe version. Using the option **-sstart**, SOLID *SynchroNet* can be started as a service executable and started and stopped from the service manager. If SOLID *SynchroNet* is started without the **-sstart** option, it starts as an icon exe like the w95, w98, and w2000 versions. The service version of SOLID *SynchroNet* cannot interact with the display and cannot create a new database. The service version writes warning and error messages also to the NT event log. SOLID *SynchroNet* can also install and remove services using this command line option.	`SOLID.EXE -s"install,SOLID, D:\SOLID\SOLID.EXE -sstart -cd:\SOLID"`
		
		`SOLID.EXE -s"install,SOLID, D:\SOLID\SOLID.EXE -sstart -cd:\SOLID,autostart"`
		
		`SOLID.EXE -s"remove,SOLID"`

-U*username*	See option **-x** execute or **-x** exit. If used without the **-x** option, specifies the username for the database being created.	

Option	Description	Examples
-P*password*	See option **-x** execute or **-x** exit. If used without the **-x** option, specifies the given password for the database being created.	
-x autoconvert -C*catalogname*	Converts database format to version 3.5 and starts server process. -C*catalogname* is required to specify the default system catalog name for the database.	
-x convert -C*catalogname*	Converts database format to version 3.5 and exits. -C*catalogname i*s required to specify the default system catalog name for the database.	
-x execute:*input file*	Prompts for the database administrator's user name and password, creates a new database, executes SQL statements from a file, and exits. The options -U and -P can be used to give the database the administrator's user name and password.	`solid.exe -x execute:init.sql` `solid.exe -x execute:init.sql -Udba -Pdba`
-x exit	Prompts for the database administrator's user name and password, creates a new database, and exits. Options -U and -P can be used to give the database administrator's user name and password.	`solid.exe -x exit` `solid.exe -x exit -Udba -Pdba`
-x forcerecovery	Does a forced roll-forward recovery.	
-x hide	Hides server icon.	
-x ignoreerrors	Ignores index errors.	
-x testblocks	Tests database blocks.	
-x testindex	Tests database index.	
-?	Help = Usage.	

Glossary

This glossary gives you a description of the terminology used in this guide.

Asynchronous store and forward messaging

The messaging architecture of SOLID *SynchroNet*. All messages are stored to the SOLID database prior to sending them to the other database. This ensures that no messages are ever lost in the synchronization process.

Binary Large Object (BLOb)

A BLOb is a large block of binary information such as a picture, video clip, sound excerpt, or a formatted text document. BLObs can be saved to and retrieved from SOLID *SynchroNet*.

Catalog

A catalog logically partitions a database so that data is organized in ways that meet business or application requirements. 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 a 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* in DML statements.

Checkpoint

Checkpoints are used to store a consistent state of the database quickly onto the disk. After a system crash, the database will start recovering transactions from the latest checkpoint. The more frequently checkpoints are made, the fewer transactions need to be recovered from the log file.

Client/server computing

Client/server computing divides a large piece of software into modules that need not all be executed within the same memory space nor on the same processor. The calling module becomes the ‘client’ that requests services, and the called module becomes the ‘server’ that provides services. Client and server processes exchange information by sending messages through a computer network. They may run on different hardware and software platforms as appropriate for their special functions.

Two basic client/server architecture types are called two-tier and three-tier application architectures.

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.

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

Database management system (DBMS)

A DBMS is a system that stores information in and retrieves information from a database. A DBMS typically consists of a database server, administration utilities, an application interface, and development tools.

Database procedures (Stored procedures)

Database procedures allow programmers to split the application logic between the client and the server. These procedures are stored in the database, and they accept parameters in the activation call from the client application. This arrangement is used by intelligent transactions that are implemented with calls to stored procedures.

Database procedures allow programmers to split the application logic between the client and the server.

Event Alerts

Events are objects with a name and parameters. Event alerts are used to signal an event in the database. The signal is sent from an application using the `POST EVENT` command. The signal is received by one or more client applications waiting for the event. The use of event alerts removes resource consuming database polling from applications.

Full publication

A full publication returns all subscribed rows of the publication from the master to a replica database. The initial subscription is always a full publication. If the tables in the publication are set for *incremental publication,* subsequent subscriptions for the same publication contain only the data that has been changed since the prior subscription.

Incremental publication

A subscription that returns only those rows of a publication that have been inserted, updated or deleted in the master database since the previous subscription.

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.

Log file (Transaction log)

This file holds a log of all committed operations executed by the database server. If a system crash occurs, the database server uses this log to recover all data inserted or modified after the latest checkpoint.

Master database (master)

A *SynchroNet* database that contains the official version of all data and provides publications for replicas to subscribe.

Messages (see Asynchronous Store and Forward Messaging and Synchronization message)

Multi-master synchronization model

In a multi-master synchronization model, a local database can contain replica databases from multiple masters. For each replica database, a catalog is created in the local database. This keeps replica data from different masters separated. A local database can also contain master data in one or multiple catalogs. This keeps local data separate from shared data.

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 the hierarchy.

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

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.

SOLID 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 Windows operating systems' registry settings.

Official data (official version)

In a *SynchroNet* 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.

Open Database Connectivity (ODBC)

ODBC is a programming interface standard for SQL database programs. SOLID *SynchroNet* offers a native ODBC programming interface.

Optimizer Hints

Optimizer hints (which is an extension of SQL) are directives specified through embedded pseudo comments within query statements. The Optimizer detects these directives or *hints* and bases its query execution plan accordingly. Optimizer hints allow applications to be optimized under various conditions to the data, query type, and the database. They not only provide solutions to performance problems occasionally encountered with queries, but shift control of response times from the system to the user.

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.

Publication

A predefined, consistent subset of master data that a replica an request (subscribe). A publication can be used to subscribe different subsets within the publication.

Relational database management system (RDBMS)

SOLID *SynchroNet* is an RDBMS, which stores and retrieves information that is organized into two-dimensional tables. This name derives from the relational theory that formalizes the data manipulation requests as set operations and allows mathematical analysis of these sets. RDBMSs typically support the SQL language for data manipulation requests.

Replica database (replica)

A *SynchroNet* database that contains a subset of master data and some tentative local transaction data.

Schema

All tables are contained in a higher level construct called schema. It is a place where tables and related objects are gathered together under one qualifying name. For each schema there are zero or more tables, and for each table, there is exactly one schema to which it belongs. The relationship between a schema and its tables is similar to that of an operating system directory and the files contained within that directory.

Sequence objects

Sequence objects generate number sequences for objects stored in databases. Sequences have an advantage over separate tables. They are specifically fine-tuned for fast execution and result in less overhead than normal update statements.

SOLID directory

The default directory for storing SOLID DBMS database files. This is the server program’s working directory.

Structured Query Language (SQL)

SQL is a standardized query language designed for handling database requests and administration. The SQL syntax used in SOLID *Embedded Engine* is based on the ANSI X3H2-1989 Level 2 standard including important ANSI X3H2-1992 (SQL2) extensions. For a more formal definition of the syntax, refer to *Appendix D SOLID SQL Syntax* of **SOLID Administrator Guide.**

SQL Access Group's Call Level Interface (SAG CLI)

SAG CLI is a programming interface standard that defines the functions that are used to submit dynamic SQL clauses to a database server for execution. The ODBC interface is also based on SAG CLI. The SOLID *SQL API* conforms to the SAG CLI standard.

Subscription

The definition of a specific refreshable subset of master data in the replica; that is, an incarnation of publication in one replica. "Subscription" also refers to the event of requesting the data from the master database and/or to the resulting data in a replica database.

Synchronet Structured Query Language (SQL)

SynchroNet commands are extensions to SOLID SQL. They allow the management of distributed data through command statements that specify synchronization operations. *SynchroNet* commands are executed using the SOLID *SQL Editor* or SOLID *DBConsole*, in either batch or interactive mode.

Synchronization history

A set of data containing change history of a particular table. 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 subscriptions to publications).

Synchronization process

A process that controls the assembly and sending of synchronization messages at a replica database.

Tentative data

All data in a replica database that is set up 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 be processed. 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.

Triggers

Triggers are pieces of logic, which a Solid server automatically executes when a user attempts to change the data in a table. When a user modifies data within the table, the trigger that corresponds to the command (such as insert, delete, or update) is activated.

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 redundancy 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