# Table of Contents

**About This Document**  
ix  
Purpose of This Document  
ix  
Audience  
ix  
How This Document Is Organized  
ix

**PART ONE**  
**Cloudsync User’s Guide**

1. **Introducing Cloudsync**  
1-1  
Process Overview  
1-1  
Advantages of Cloudsync  
1-3  
Operating Environment  
1-3  
Cloudsync Capabilities  
1-4  
  Creating Targets  
1-5  
  Synchronizing Databases  
1-5  
  Handling Conflicts  
1-5  
Writing and Deploying Cloudsync Applications  
1-6  
SQL-J Extensions for Cloudsync  
1-7

2. **Publications**  
2-1  
Publication Basics  
2-1  
What a Publication Contains  
2-2
# Cloudscape Synchronization Guide

## 3 Creating a Synchronized System

Steps for Creating a Synchronized System

- Determining the Database Type
- Sample Code for Creating a Synchronized System
- Configuring Network Addresses
  - Servlet API
    - Cloudscape Class Name
    - Refresh Request URL
  - Built-in HTTP Support
    - Port Numbers
    - Network Addresses
  - Moving the Source Machine

## 4 Synchronizing Databases

- Agreement and Synchronization
- The Refresh Operation
- Provisional Transactions
- Steps of the Refresh Operation
  - Applying Provisional Transactions at the Source
    - Scenario 1: The transaction applies successfully
    - Scenario 2: A consistency error occurs
    - Removing records from SYSERRORS
      - Scenario 3: An environmental error occurs and the transaction remains provisional
  - Applying Source Data to the Target
  - Target Refresh Failures
    - Connection Failure
    - Other Failure
  - Concurrency During a Refresh

---

*Cloudscape Synchronization Guide*
Refresh Frequency 4-6
Refreshes and Jar Files 4-7
Triggers and the Refresh Operation 4-7
Refreshing Copied Sources and Targets 4-8
Refresh Security 4-8
  User Authentication and User Authorization 4-9
  Refresh Security Properties 4-9
  Refreshing Using SSL 4-10
  Limitations of the Security Model 4-11
  Security Configuration Example 4-11

5 Updates and Work Units 5-1
Updates from Multiple Targets 5-1
Controlling Update Behavior 5-2
Changed-Value Updates 5-3
Work Units 5-3
  Understanding Work Units 5-4
  Work Units are Applied Provisionally 5-4
  Work Units Operate from Target to Source 5-5
  Work Units are Method Aliases 5-5
  Benefits of Work Units 5-5
SQL-J Extensions for Work Units 5-6
Creating a Work Unit 5-6
Invoking a Work Unit 5-7
Specifying the Connection in a Work Unit 5-8
Transactions and Work Units 5-9
Error Handling in Work Units 5-9
Altering Work Unit Methods 5-10

6 Designing Synchronized Applications 6-1
Application Design Overview 6-1
Design Guidelines 6-2
Steps for Designing a Distributed System 6-2
  Step 1: Defining the Application 6-3

Cloudscape Synchronization Version 3.0
Step 2: Listing the Operations 6-4
Step 3: Describing the System-Wide Data 6-4
Step 4: Describing the Target Data 6-5
Step 5: Listing the Inputs 6-6
Step 6: Listing the Outcomes 6-6
Step 7: Representing the Outcomes in the Database 6-7

Work Unit Examples 6-7
The placeOrder Work Unit 6-8
The Target-side Method 6-8
The Source-side Method 6-8
Using the placeOrder Work Unit 6-9
The sendLead Work Unit 6-9
The Target-side Method 6-9
The Source-side Method 6-9
Using the sendLead Work Unit 6-10

Work Unit Example Code 6-10

Publication Design Issues 6-15
Making Targets Small and Efficient 6-15
Target-Side Validation 6-16
Handling Published Foreign Keys 6-17
Creating Universal Keys 6-18

7 Deploying Synchronized Systems 7-1
Configuring Source and Target Machines 7-1
What the Server Needs 7-1
What the Target Machine Needs 7-2
Distributing Application Code Within the Database 7-2
Example of Jar Files in a Database 7-3
Locally Configuring Target Database Properties 7-4
Setting Properties 7-5
Checking Property Values 7-6

8 Debugging Synchronized Systems 8-1
The printFailedTransaction Method 8-1
Creating a Method Alias 8-2
Using printFailedTransaction 8-2
Note to Cloudconnector Users 8-2
Example Queries 8-3
Understanding SYSERRORS Query Results 8-4
printFailedTransaction Output 8-4
For each failed transaction:

PART TWO
Cloudsync Reference

9 Cloudsync Commands Reference 9-3
ALTER PUBLICATION 9-4
COPY PUBLICATION 9-6
CREATE PUBLICATION 9-8
CREATE WORK ALIAS 9-13
DROP PUBLICATION 9-14
DROP WORK ALIAS 9-15
REFRESH 9-16

10 Cloudsync System Tables 10-1
SYSPUBITEMS 10-2
SYSPUBS 10-3
SYSSYNCRINSTANTS 10-4

11 URL Attributes for Synchronization 11-1
convertToSource=true 11-2
createSource=true 11-3
createTargetFrom=copyFileName 11-4

12 Cloudsync Properties 12-1
cloudscape.database.readAccessTarget 12-3
cloudscape.database.refresh 12-5
cloudscape.database.sourceUser 12-6
cloudscape.database.synchronizationCleanup 12-8
cloudscape.DataComm.listenType 12-9
cloudscape.DataComm.port 12-10
cloudscape.DataComm.url 12-11
cloudscape.listener.synchronization.address 12-14
cloudscape.refresh.guaranteedRefreshInterval 12-15
cloudscape.synchronization.workUnitOnly 12-16
sysinfo.publicationID 12-18
sysinfo.publicationName 12-19
sysinfo.sourceDatabaseID 12-20
sysinfo.sourceDatabaseName 12-21
sysinfo.sourceServerAddress 12-22

PART THREE
Appendixes

Appendix A
Illustrations of the Refresh Operation A-1
Possible Outcomes of a Provisional Transaction A-1
Timelines A-3
A Simple Scenario A-3
A Slightly More Complicated Synchronization Scenario A-5

Index IN-1
About This Document

- “Purpose of This Document” on page ix
- “Audience” on page ix
- “How This Document Is Organized” on page x

Purpose of This Document

This book is for developers using the Cloudsync option. It is assumed that you are already familiar with using Cloudscape in embedded and server environments.

Audience

The Cloudscape Synchronization Guide is for Cloudscape users who have installed the Cloudscape synchronization product. It provides both how-to and reference material for users of Cloudscape synchronization. This guide is to be used in conjunction with the Cloudscape Developer's Guide and the Cloudscape Reference Manual.
How This Document Is Organized

This document contains two parts: a user’s guide and a reference section. The document includes the following chapters:

Part One (User’s Guide):

- Chapter 1, “Introducing Cloudsync”
  A high-level overview of Cloudscape synchronization.
- Chapter 2, “Publications”
  Explains publications, which define the shared part of a database.
- Chapter 3, “Creating a Synchronized System”
  Shows the steps for setting up a basic synchronized system and provides an example.
- Chapter 4, “Synchronizing Databases”
  Explains the refresh operation, which synchronizes databases.
- Chapter 5, “Updates and Work Units”
  Shows how to update data given the possibility of conflicting updates from elsewhere in the system.
- Chapter 6, “Designing Synchronized Applications”
  Provides an approach to designing a distributed application with Cloudsync, using a simple example.
- Chapter 7, “Deploying Synchronized Systems”
  Shows how to deploy a distributed application.
- Chapter 8, “Debugging Synchronized Systems”
  Explains some of the tools provided for debugging distributed applications.

Part Two (Reference):

- Chapter 9, “Cloudsync Commands Reference”
  Reference pages for the Cloudscape synchronization SQL-J statements.
- Chapter 10, “Cloudsync System Tables”
  Descriptions of the Cloudscape synchronization system tables.
- Chapter 11, “URL Attributes for Synchronization”
  Descriptions of the Cloudscape synchronization URL attributes.
- Chapter 12, “Cloudsync Properties”
  Descriptions of Cloudscape synchronization properties.

Part Three (Appendixes):

- Appendix A, “Illustrations of the Refresh Operation”
  Graphically illustrates the synchronization process.
PART ONE

Cloudsync User’s Guide
1 Introducing Cloudsync

Cloudsync is an extension to Cloudscape that provides an easy way to share data and application code among different devices. It is specially designed for distributed applications that do one or both of the following:

- run on intermittently connected computing devices
- require the performance and convenience of a local database

Using Cloudsync, you simply provide the application logic and data, and Cloudsync distributes the logic and synchronizes the data almost transparently.

This chapter introduces Cloudsync. Topics in this section are covered in greater detail in later chapters. This chapter includes these sections:

- “Process Overview” on page 1-1
- “Advantages of Cloudsync” on page 1-3
- “Operating Environment” on page 1-3
- “Cloudsync Capabilities” on page 1-4
- “Writing and Deploying Cloudsync Applications” on page 1-6
- “SQL-J Extensions for Cloudsync” on page 1-7

Process Overview

This section provides a high-level summary of the steps involved in creating a synchronized system. The remaining chapters of this guide discuss aspects of synchronization more specifically, and should be consulted for in-depth information about each phase of this process.
Introducing Cloudsync

A synchronization system consists of one source database and one or more target databases. Basically, setting up the source and target databases consists of these steps:

1. Create the source database, containing all objects you want in the target (including work units).
2. Create/generate the objects necessary to create the target (a publication and a copy file).
3. Create a target database using the copy file.

Following is a brief description of the steps:

- **Create the source database**
  The source database is the master database for a synchronized system, and is designed to receive updates from one or more external target databases. Source databases are created using the `createSource=true` attribute (or from Cloudview). Standard databases can be converted to Source databases using the `convertToSource=true` attribute.

- **Optionally, write update logic and create a Work Unit alias**
  A work unit alias references a method containing logic to apply during refreshes. Work unit aliases are created at the source database, but they can specify different logic for the source and target. If you want to use a work unit at the target, it must be have been created prior to the publication (next). To create a work unit alias, use the `CREATE WORK ALIAS` statement (or use Cloudview).

- **Create a Publication object**
  A Cloudsync publication is created at the source database, and can be thought of as a kind of database-level view. Publications are treated as database objects, and are stored with the source database. A publication specifies which source database objects and data (and, optionally work unit update logic) you want to include in the target database. Use the `CREATE PUBLICATION` statement to create a publication.

- **Generate a copy file (from the Publication)**
  The copy file will be used to create the target database in the next step. After you have created a publication, use the `COPY PUBLICATION` statement to generate a copy file from it.

- **Create a target database**
  A target database is always created using a copy file. In fact, a database created as a target does not allow explicit DDL statements. To create a database, use the `createTargetFrom=copyFileName` attribute on the database connection URL (or use Cloudview). You can create multiple
target databases from a single copy file, or you can use a different copy file for each target database.

For detailed discussions of setting up synchronized systems, see Chapter 3, “Creating a Synchronized System”, and Chapter 6, “Designing Synchronized Applications”.

Advantages of Cloudsync

Cloudsync:

- is easy to use.
- synchronizes all updates, deletions, and insertions to all tables, indexes, and views.
- is efficient: only changes to the data, schema, or application code are copied to other databases. Unchanged elements are not copied.
- guarantees that databases in the system remain internally consistent and reflect each other’s changes.
- provides an easy, robust framework for handling conflicting updates.
- can distribute application changes as well as data and schema changes.

Operating Environment

Cloudsync uses a hub-and-spoke topology. The source database, at the “hub,” holds the master copy of the database objects. All administration is handled at the source. The source typically runs as a server. See the Cloudscape Server and Administration Guide for more details.

A target database, on a “spoke,” holds a copy of the source database (or a subset of the database). Targets are designed for non-technical end users and require no local administration. Targets are typically embedded in an application; the user of the target application may not even be aware of the database. Cloudsync can also distribute and update the target application itself. A target can run on almost any computing device that contains a Java Virtual Machine (JVM).
Cloudsync Capabilities

Cloudsync provides the usual features that support client/server database applications. It also includes additional capabilities to support distributed applications:

- **Creating Targets**
  Publications are used to create and define target databases.

- **Synchronizing Databases**
  The Refresh Operation synchronizes a Target database with the Source database.

*Cloudscape Synchronization Guide*
• Handling Conflicts
  Server-driven arbitration, provisional transactions, and work units help maintain data consistency.

Creating Targets

A publication defines the part of the source database to be shared with a certain set of target databases. A publication can also contain target application code. A single SQL-J command at the source creates a publication that can create and manage any number of targets. A publication restricts the target to only the data the user needs, thus enhancing application performance, security, and transfer time across the network.

See Chapter 2, “Publications”, for more information on creating and using publications.

Synchronizing Databases

Disconnected databases in a distributed system can change independently. The process that causes each database to reflect changes in the other databases is called synchronization. In Cloudscape synchronization, the refresh operation, invoked by a single SQL-J command at a target, synchronizes data automatically between the target and the source. Changes initiated at any target (or directly at the source) are propagated to other targets as they refresh. When the refresh operation completes, the target is in agreement with the source and therefore has consistent data.

See Chapter 4, “Synchronizing Databases”, for more details on the refresh operation.

Handling Conflicts

Different databases may attempt to change data in conflicting ways, and the refresh operation must handle this. For example, in an order entry application, the system must avoid corruption if a user at one target orders eight items and a user at another target orders seven items, when only ten items are available.

Cloudsync provides a simple framework for handling conflicts:
Introducing Cloudsync

- Source Arbitration
  The source is the final arbiter of all transactions on the system. All changes in the synchronized system must apply at the source before being propagated to targets. This makes it unnecessary to resolve conflicts directly between targets. Since targets need not be available to resolve conflicts, they can disconnect at any time.

- Provisional Transactions
  All transactions originating at a target are provisional until a refresh operation completes. If a transaction is applied successfully at the source during the refresh operation, it becomes durable. If it is rejected at the source, it is rolled back on the target after the refresh.

- Work Units
  Cloudsync provides a simple, flexible means, the work unit, to specify how to handle a potential conflict when a provisional target data change is applied at the source. Work units are optional. If no conflict is anticipated for a given transaction, the data can synchronize automatically at refresh time without a work unit.

  Work units are ordinary Java methods that describe logical operations on both target and source databases. Cloudsync automatically executes a work unit at the target and the source at the appropriate time.

  See Chapter 5, “Updates and Work Units”, for more information on work units.

Writing and Deploying Cloudsync Applications

Designing and deploying applications using Cloudsync is straightforward.

- Design the application to run against the target while it is disconnected from the source.
- Optionally enclose database updates in work units to ensure that potential conflicts are handled properly at refresh time. Distribute the application and its work units to the targets when they are created.

Chapter 6, “Designing Synchronized Applications”, provides step-by-step instructions, along with an example, showing how to design a distributed application using Cloudsync. Chapter 7, “Deploying Synchronized Systems”, shows how to distribute applications to targets using Cloudsync.
SQL-J Extensions for Cloudsync

Cloudsync provides additional SQL-J statements to support synchronization:

- CREATE PUBLICATION creates a publication.
- ALTER PUBLICATION alters an existing publication.
- DROP PUBLICATION drops a publication.
- COPY PUBLICATION creates the file from which a target database is created.
- CREATE WORK ALIAS registers a work unit.
- DROP WORK ALIAS removes a work unit.
- REFRESH synchronizes a target with the source.

See Chapter 9, “Cloudsync Commands Reference”, for details about these statements.

The SQL-J CALL statement and VALUES clause have been enhanced to support Cloudsync. See the Cloudscape Reference Manual for more information.

The database connection URL has also been enhanced to support Cloudsync. See Chapter 11, “URL Attributes for Synchronization”, for information.

Some properties are used only with Cloudsync. See Chapter 12, “Cloudsync Properties”, for information.
Introducing Cloudsync
This section covers the following topics:

- “Publication Basics” on page 2-1
- “What a Publication Contains” on page 2-2
- “Altering Publications” on page 2-3
- “Altering Published Tables” on page 2-3
- “Publishing Foreign Keys” on page 2-4
- “Publishing Triggers” on page 2-4

Publication Basics

A publication defines the part of a source database to be shared with a certain set of targets. This can include application logic stored in the source database. A source database may contain any number of publications, each used with a different set of target databases.

A publication is created with a CREATE PUBLICATION statement on the source database. The COPY PUBLICATION statement, also run at the source, uses an existing publication to create a copy file. A target database is created from the copy file.

You can use a single publication to create any number of copy files, and you can use a single copy file to create any number of target databases. Once you have created all the targets you want from a copy file, you can delete it.

NOTE: Since a publication defines a target’s dictionary objects, no DDL (e.g., CREATE TABLE) statements are permitted at a target.
A publication may contain parameters. You can specify the values for the parameters when the copy file is created. This allows a single publication to be used to create different target databases, each containing the same dictionary objects but only the subset of the data relevant to that target.

A publication can also contain jar files and target database properties. This lets you distribute and manage target application code as well as the Java objects stored in target database user tables. Like changes to data, changes to application code can be distributed to any target in the synchronized system.

You can alter a publication by issuing ALTER PUBLICATION at the source database. The changes you make with ALTER PUBLICATION are propagated to a subscribing target when it synchronizes with the source.

See “CREATE PUBLICATION” on page 9-8, “COPY PUBLICATION” on page 9-6, “DROP PUBLICATION” on page 9-14, and “ALTER PUBLICATION” on page 9-4 for the complete specifications of the statements that create, copy, drop, and alter publications.

See “Sample Code for Creating a Synchronized System” on page 3-3 for an example of creating and copying a publication.

**What a Publication Contains**

The following dictionary items can be explicitly published:

- tables (other than system tables)
- indexes
- views
- method aliases
- work unit aliases (see “Work Units” on page 5-3)
- class aliases
- stored prepared statements
- foreign keys
- triggers
- user-defined aggregates
- jar files (typically containing application code or Java classes and resources)
- target database properties

*Cloudscape Synchronization Guide*
The following database items are implicitly published:

- primary keys
- unique and check constraints (if all columns in the constraint are published)

**Altering Publications**

The ALTER PUBLICATION statement lets you change a publication. You can add or drop any database object (except some parameters) that you can include in the publication. In addition, you can add a parameter to an existing publication.

See “ALTER PUBLICATION” on page 9-4 for details about the ALTER PUBLICATION command.

**Altering Published Tables**

The ALTER TABLE command allows you to add a column to a table, add a constraint, or drop a constraint. You can use ALTER TABLE even if the table has already been published.

If ALTER TABLE adds a column, and if the column’s table was published without a column list, the column automatically appears in the publication. If the table was published with a column list, you will have to drop and re-add the table to the publication using ALTER PUBLICATION.

If ALTER TABLE adds a unique or check constraint, and if all columns mentioned in the constraint are in the publication, the constraint is automatically added to the publication.

If ALTER TABLE drops a unique or check constraint (other than a primary key) that is implicitly published, that constraint is removed from the publication.

ALTER TABLE fails with an error if you attempt to drop:

- an explicitly published item
- the primary key of a published table
**Publishing Foreign Keys**

Foreign key constraints are not implicitly included in a publication. However, even if a foreign key on the source is not published to a target, the source still rejects all changes from that target that violate the foreign key.

To enforce a foreign key at the target as well as the source, you must publish the foreign key. To do this, use the ADD FOREIGN KEY clause of CREATE PUBLICATION or ALTER PUBLICATION.

A foreign key that is valid for one set of data in a database may be invalid for a subset of that data. Because a target is typically a subset of the source, a foreign key may be invalid on a target even if it is valid on the source. Therefore, if a publication contains a foreign key, and if the target data would violate the foreign key if the target database were created, COPY PUBLICATION fails. If COPY PUBLICATION succeeds, the foreign key is enforced on the target until the foreign key is dropped from the publication.

Be careful when publishing foreign keys, or you may create a target that cannot refresh. See “Handling Published Foreign Keys” on page 6-17 for information on how to safely design publications containing foreign keys.

**Publishing Triggers**

By default, triggers are not published. This means they fire only on the source database, not on the target.

If you want updates on the target to more closely follow the expected behavior at the source, you can publish a trigger. To do this, use the ADD TRIGGER clause of CREATE PUBLICATION, or the ADD TRIGGER or DROP TRIGGER clauses of ALTER PUBLICATION.

If you try to publish a trigger that cannot function on a target because of a database object discrepancy (for example, it modifies an unpublished table), the CREATE PUBLICATION or ALTER PUBLICATION will not succeed. However, there are some scenarios in which the CREATE PUBLICATION statement for a trigger may succeed, but the trigger will still fail on the target. For example, as with any other class reference, your trigger may fail at the target if it references a class in the source classpath that isn’t in the target’s classpath. Please verify that triggers work at the target.
If the published trigger operation is a Java method call, the target looks for a Java class with the given class name. In this scenario, verify the desired classes are available at the target before firing the trigger.

See “Triggers and the Refresh Operation” on page 4-7 for details on how changes caused by a trigger at the target are handled at the source.
Creating a Cloudscape synchronization system is very simple. Basically, you

- create a database
- create a publication
- copy the publication to a file
- create a target database using the copy file

Each of these steps take a single line of code. See “Sample Code for Creating a Synchronized System” on page 3-3 for an example.

Typically, you configure source and target databases for synchronization at the time they are created.

**NEW:** In version 3.0 you can convert non-synchronization-enabled databases to source databases. For more information, see “URL Attributes for Synchronization” on page 11-1.

Typically, the source database runs as a server. For more information on servers, please consult the *Cloudscape Server and Administration Guide*. A target can run on almost any device that includes a Java Virtual Machine (JVM).

This section shows how to create synchronized databases. It covers the following topics:

- “Steps for Creating a Synchronized System” on page 3-2
- “Determining the Database Type” on page 3-3
- “Sample Code for Creating a Synchronized System” on page 3-3
- “Configuring Network Addresses” on page 3-5
NOTE: You can also use Cloudview to create a synchronized system. For more information, see the Cloudview online help.

Steps for Creating a Synchronized System

Follow these steps to create a basic synchronized system:

1. Create a source database by setting the database connection URL attribute `createSource` to “true”. (You can also convert a non-source database to a source database using the `convertToSource` attribute).
   See “URL Attributes for Synchronization” on page 11-1 for details on `createSource` and `convertToSource`.

2. Connect to the source database. (Steps 1 and 2 are typically combined in a single line of code, since when you create the database you start a connection to it.)

3. Create tables in the source database and populate them with data.

4. Create a publication in the source database, using the `CREATE PUBLICATION` statement. See “CREATE PUBLICATION” on page 9-8 for details.

5. Still at the source database, copy the published dictionary objects and data to a file, using the `COPY PUBLICATION` statement. See “COPY PUBLICATION” on page 9-6 for details.

6. If necessary, move the copy file to the machine where you will create the target.

7. Create a target database, most likely in a different JVM from the source, by connecting to a URL in which the `createTargetFrom` property is set to the path or URL of the copy file.
   NOTE: Do not enclose the copy file name in single quotes (').
   See “URL Attributes for Synchronization” on page 11-1 for more details.

Once you complete these steps, you have created a source database and an associated read-write, refreshable target database.

NOTE: You can also create source and target databases, tables, and publications using the graphical Cloudview environment. Consult Cloudview’s online help system for more information.
Determining the Database Type

You can determine whether an existing database is a source, target, or standard (non synchronization-enabled) database. To determine the type of database, use the `COM.cloudscape.database.PropertyInfo.getDatabaseProperty` method.

**NEW:** Cloudscape now provides a class alias for this method: `PropertyInfo`.

For example, for a target database, the following query returns `true`:

```java
VALUES PropertyInfo.getDatabaseProperty('cloudscape.database.target')
```

For a source database, the following query returns `true`:

```java
VALUES PropertyInfo.getDatabaseProperty('cloudscape.database.source')
```

Sample Code for Creating a Synchronized System

The following sample code creates a simple synchronized system.

**NOTE:** This example is an altered and simplified subset of the application described in Chapter 6, “Designing Synchronized Applications”.

This small piece of code creates a complete synchronized system consisting of one source database and one target database. The target database is designed to contain a subset of all orders: just those placed by sales representative number two.

After these databases are created, a target user can issue a single command (REFRESH) to synchronize any target changes to the source, or source changes to the target. (Chapter 4, “Synchronizing Databases”, explains how.)

In this example, the source is on a Cloudconnector server. For simplicity’s sake, the source and target are assumed to be on the same network. This example uses Windows-style path names.

```java
//Start the Cloudconnector driver
Class.forName("COM.cloudscape.core.WebLogicDriver").newInstance();
```
//Create the source database and connect to it using a Cloudconnect server
Connection conn = DriverManager.getConnection
("jdbc:cloudscape:weblogic:toysSourceDB;createSource=true");

//Create a statement in the source database
Statement s = conn.createStatement();

/* Create TOYS and ORDERS tables in the source database.
(Note that tables to be published must contain a primary key).*/
s.execute("CREATE TABLE toys " +
"(toy VARCHAR(30) CONSTRAINT toys_pk PRIMARY KEY, " +
"price DECIMAL(5,2), " +
"quantity INT)\n");

s.execute("CREATE TABLE orders " +
"(orderID INT CONSTRAINT orders_pk PRIMARY KEY, " +
"salesRepID INT, " +
"customer VARCHAR(50), " +
"toy VARCHAR(50), " +
"quantity INT CONSTRAINT enoughToys CHECK (quantity >= 0)\n");

//Populate the TOYS table
s.execute("INSERT INTO toys VALUES ('barbie', 9.50, 1)\n";

s.execute("INSERT INTO toys VALUES ('beanie baby elephant', 6.50, 0)\n";

/*Create a publication named ToyOrders that contains TOYS and ORDERS
(primary keys are implicitly published). All columns from both
tables are published, and a row filter (WHERE clause) is specified for
the ORDERS table.*/
s.execute("CREATE PUBLICATION ToyOrders (salesRepID INT) " +
"ADD TABLE toys " +
"ADD TABLE orders " +
"WHERE (salesRepID = ?salesRepID)\n";

//Copy the publication to a file, specifying parameters entered by user
s.execute("COPY PUBLICATION ToyPub (salesRepID = 2) " +
"TO FILE 'c:\MyDBs\toysFile.copy'\n";

//Close the connection to the source database
conn.close();

//Start the JDBC embedded driver
Class.forName("COM.cloudscape.core.JDBCDriver").newInstance();

/* Create a target database using the copy file, and connect to the
 * target database.
 * Make sure not to introduce blank spaces in the URL.
 */
Connection conn2 = DriverManager.getConnection("jdbc:cloudscape:toysTargetDB;" +
"createTargetFrom=c:\MyDBs\toysFile.copy");

//Close the connection to the target database
conn2.close();

//You now have a synchronizable, read-write target database.

Because the toysFile copyfile was created using the parameter ‘salesRepID=2’,
refreshes at the resulting ToysTargetDB database will receive only those ORDERS
rows from the source that contain a value of ‘2’ in the SalesRepID column.

Configuring Network Addresses

This section explains how to configure your system so that source and target
databases can communicate with each other.

• “Servlet API” on page 3-6
• “Built-in HTTP Support” on page 3-7
• “Moving the Source Machine” on page 3-8

Cloudscape uses the HTTP protocol to send and receive synchronization (refresh)
messages between source and target databases. Since the target initiates refresh
requests in Cloudscape synchronization, the target must specify the source
database’s network address, which includes a host name (or host IP address) and
port number.

The target database obtains the host name and port number of the source database
from the copy file at the time the target is created. To ensure the correct source
network address is copied to the targets, you should specify the source’s network

Cloudscape Synchronization Version 3.0
address before any targets are created. For information about moving the source after targets are created, see the section “Moving the Source Machine” on page 3-8.

There are two ways to set up the source database to “listen” for refresh requests from a target:

- Servlet API
- Built-in HTTP Support

**NEW:** In version 3.0, built-in HTTP support can be JVM-native HTTP support, as well as native HTTPS support.

Network address configurations are set only at the source, not at the target.

The `cloudscape.DataComm.listenType` property determines which means of communication to use. The two values of `cloudscape.DataComm.listenType` are `servlet` and `builtinHttp`. The default is `builtinHttp`.

### Servlet API

Use the servlet API if the source database is running on a server framework that supports it. Cloudconnector supports servlets. See the *Cloudscape Server and Administration Guide* for more information on server frameworks.

To use the servlet API, set the `cloudscape.DataComm.listenType` property to `servlet`. You must also:

- tell the server framework the Cloudscape Class Name that implements the refresh service
- tell Cloudscape the Refresh Request URL (the URL to which targets should send refresh requests)

### Cloudscape Class Name

The Cloudscape class name to use with the servlet API is `COM.cloudscape.core.ServletHandler`. How you set this class name depends on the server framework. For example, in the case of Cloudconnector, insert a line like the following in the `weblogic.properties` file:

```
weblogic.httpd.register.sync=COM.cloudscape.core.ServletHandler
```

**NOTE:** The word “sync” could be replaced by any other word. However, the word specified here must be the same as that used in the refresh request URL.
Refresh Request URL

The property to set in `cloudscape.properties` for the URL is `cloudscape.listener.synchronization.address`. The exact value of this property depends on the server framework. For Cloudconnector, it would be something like the following for a machine named `myhost`:

```
cloudscape.listener.synchronization.address=http://myhost:7001/sync
```

**NOTE:** The `cloudscape.listener.synchronization.address` property specifies the address targets use to refresh. This property does not affect the source. To specify the port number that the source listens on, use the `cloudscape.DataComm.port` property.

Built-in HTTP Support

If you use built-in HTTP support, you don’t have to set any configuration properties as long as the default port number and address are acceptable.

Port Numbers

By default, the source listens on port 2001. To set the source to listen on a different port number, assign a different value to the `cloudscape.DataComm.port` property in the source database system before running COPY PUBLICATION.

If there is any chance that another source database will run concurrently on the same host computer in a different Cloudscape system, please be sure to specify unique port numbers for each source database.

Network Addresses

Typically, you don’t need to set the network address when using built-in HTTP support. By default, Cloudscape assumes that the URL to which targets send refresh requests is:

```
http://hostname:port/application
```

To set a different network address from the default, alter this address as follows:

- replace `hostname` with the hostname you want
- specify the port number, even if it is the default (2001)
- be sure to include `/application` at the end of the address
For example, the following URL may be a valid network address using built-in HTTP support:

```
http://myhost:2003/application
```

See Chapter 12, “Cloudsync Properties”, for information on the synchronization properties. Also see *Tuning Cloudscape* for more information on using properties.

**Moving the Source Machine**

If you change the network address of the source after a target is created, the target can no longer connect to the source. If you move the source to another machine, the new machine must use the same network address as the old source machine.

However, if you create a special DNS name for the source’s IP address, and configure the source to use this name in its listen address, you can move the source to another machine, correct the DNS entry at the source, and still synchronize with targets.
4 Synchronizing Databases

This section covers the following topics:

- “Agreement and Synchronization” on page 4-1
- “The Refresh Operation” on page 4-2
- “Provisional Transactions” on page 4-2
- “Steps of the Refresh Operation” on page 4-2
- “Concurrency During a Refresh” on page 4-6
- “Refresh Frequency” on page 4-6
- “Refreshes and Jar Files” on page 4-7
- “Triggers and the Refresh Operation” on page 4-7
- “Refreshing Copied Sources and Targets” on page 4-8
- “Triggers and the Refresh Operation” on page 4-7
- “Refresh Security” on page 4-8

Agreement and Synchronization

In a synchronized system, one database commits changes, then another database changes accordingly. The lag between the two commits is called latency. If latency accounts for all of the difference between the data in two databases, the databases are said to be in agreement, even if they do not actually match. The process that copies data to maintain agreement is called synchronization. The Cloudsync operation that synchronizes databases is called the refresh operation.
Cloudsync guarantees that a source database and its targets are always in agreement.

**The Refresh Operation**

The *refresh* operation synchronizes source and target data. A target database initiates the refresh operation. Changes from one target that apply at the source are propagated to other targets in the synchronized system when those targets refresh.

The source database cannot directly initiate a refresh operation (although it is possible to write an application which does so).

To issue a refresh, use the SQL-J command REFRESH on the target database. For more details on the REFRESH statement, see “REFRESH” on page 9-16.

**Provisional Transactions**

All transactions completed at the target since the target’s last refresh (or if there has not yet been a refresh, since the database was created) are provisional. Cloudsync may alter or reject a provisional transaction at the source during the refresh operation to preserve database consistency. Transactions become durable after the refresh operation completes. Cloudsync never alters transactions after they become durable.

**Steps of the Refresh Operation**

The refresh operation goes through the following steps:

1. Provisional transactions are applied at the source in the order in which they were applied at the target. See “Applying Provisional Transactions at the Source” on page 4-3 for possible outcomes of this step.

2. All changes to the source since this target’s last refresh, including successful transactions just applied from this target, are copied to the target.

If the refresh completes successfully:
Steps of the Refresh Operation

- Target data is in agreement with source data.
- Transactions that committed at the target before this refresh are no longer provisional. A provisional change becomes durable after it applies successfully at the source. A change is removed from the target if it is rejected at the source.

Applying Provisional Transactions at the Source

When a provisional transaction is sent to the source, one of the following occurs:

- Scenario 1: The transaction applies successfully, and it becomes durable at the source and target at the end of the refresh
- Scenario 2: A consistency error occurs, and any changes the transaction has made to user tables are rolled back on both the source and target
- Scenario 3: An environmental error occurs and the transaction remains provisional

Cloudscape synchronization keeps databases internally consistent and in agreement even if errors occur.

NOTE: By using work units, you can control when each type of error is raised. See “Error Handling in Work Units” on page 5-9 for more details.

NEW: In version 3.0, you can easily retrieve failed transaction information from SYSERRORS using the printFailedTransaction method. For more information, see “Debugging Synchronized Systems” on page 8-1.

Scenario 1: The transaction applies successfully

Since source and target data are similar, provisional target transactions typically apply successfully at the source. If they do:

1. The transaction is applied durably at the source.
2. Changes from this transaction are written durably at the target at the end of the refresh cycle.

Scenario 2: A consistency error occurs

A consistency error occurs when the provisional transaction, if applied at the source, would cause the source to become inconsistent. For example, a consistency
error occurs if the transaction violates a source check constraint. Consistency errors typically occur because the source data has changed since this target’s last refresh.

When consistency errors are encountered during refresh, the offending provisional transaction fails, but the refresh continues.

If a provisional transaction is rejected at the source to avoid a consistency violation:

1. The offending provisional transaction aborts.
2. The failed transaction is recorded in the SYSERRORS table.
3. The refresh resumes with the next provisional transaction.
4. Changes attempted by the failed transaction are rolled back from the target’s user tables after the refresh.

Cloudscape provides utilities to show descriptions of consistency errors. These utilities are in `COM.cloudscape.types.ErrorInfo`.

**Removing records from SYSERRORS**

In some cases, you may want to remove records from the SYSERRORS table. The `deleteError` method in `COM.cloudscape.database.Database` can be used to delete a specific error using its errorID.

For example, the following ij statement would remove the error from SYSTABLES whose errorID is `a352c053`.

```sql
SELECT Factory.getDatabaseOfConnection().deleteError(COM.cloudscape.database.Factory.getSystemOfConnection().getUUIDFactory().createUUID('a352c053')) FROM SYS.SYSERRORS;
```

See the *Cloudscape Reference Manual* for more information about the SYSERRORS table.

**Scenario 3: An environmental error occurs and the transaction remains provisional**

When a refresh encounters environmental errors at the source, the transaction remains provisional.

Examples of environmental errors are:

- out of memory
- method alias name not found
- unresolvable deadlock
Steps of the Refresh Operation

- source machine crash
- severe coding error

When an environmental error occurs.

1. The refresh halts.
2. The provisional transaction remains provisional. (Even ‘successful’ transactions that have not yet applied at the source remain provisional.)
3. The transaction is re-sent to the source at the next refresh.

If an environmental error occurs, the system administrator should address the problem immediately.

Applying Source Data to the Target

At the end of a successful refresh operation, the target data is replaced with a copy of the source data. (This is done efficiently; only data changes are actually sent through the wire.) Since source data changes simply overwrite target data, agreement between source and target is assured, no matter what errors occurred during the refresh or how they were handled.

Target Refresh Failures

Rarely, a refresh may complete successfully at the source, but fail on the originating target during the refresh reply. This can occur when the connection fails between the target and source during refresh, or when a publication has been poorly designed.

Connection Failure

If a refresh operation completes at the source but the connection fails during the refresh reply, the refresh fails with an I/O error and the target is left in the same state as before the refresh. Changes made at the source during this refresh endure. The user should refresh again later, and the changed data will reach the target at the target’s next successful refresh. The source recognizes changes that were previously applied and does not reapply them.
Other Failure

If a refresh operation completes at the source but fails to apply at the target for a reason other than connection failure, the user should contact the system administrator. The target error message indicates this situation. A likely cause of this situation is a foreign key violation at the target, which can be avoided with proper publication design. See “Handling Published Foreign Keys” on page 6-17 for more details.

Concurrency During a Refresh

Cloudsync permits multiple connections to a target database. While target data is being handled at the source during a refresh, the current target connection waits, but other connections may still modify the target data. If transactions commit at the target before the refresh reply returns, these transactions remain provisional at the target at the end of the refresh. Therefore, these transactions are not lost; they will be applied at the source at the next refresh.

While source data is being written back to the target, all target connections (except the one performing the refresh) are suspended.

Refresh Frequency

The target must refresh at least occasionally. If a target does not issue a refresh for a certain period of time (a week by default), it can never issue another refresh. This time limit can be configured by using the cloudscape.refresh.guaranteedRefreshInterval property at the source. Increasing the refresh interval raises storage space requirements at the source; decreasing the interval reduces these requirements. See “cloudscape.refresh.guaranteedRefreshInterval” on page 12-15 for more details.

Of course, even if a target can no longer refresh, it is quite easy to create a new target. See Chapter 3, “Creating a Synchronized System” for details.
Refreshes and Jar Files

If a target contains a jar file, and the corresponding jar file at the source has been replaced since the last refresh, a refresh returns a warning. When it receives this warning, the application should remove all references to objects defined in classes in the refreshed jar files. A simple way to do this is to terminate and restart the application.

Triggers and the Refresh Operation

NEW: The ability to create triggers is new in version 3.0.

In a Cloudsync application, you can handle a trigger in one of three ways:

- *The trigger is not published.*
  The default.

- *The trigger is published to the target, but changes made by the trigger are not sent to the source at refresh time.*
  The usual situation when you publish a trigger.

- *The trigger is published to the target, and some or all changes made by the trigger are sent to the source at refresh time.*
  Usually unnecessary, and strongly discouraged.

By default, triggers are not published to the target. If you want the target to better imitate source behavior, you can publish triggers. Another advantage of publishing a trigger is that if a target trigger causes the transaction to fail, the transaction is never sent to the source, which may improve source performance. See “Publishing Triggers” on page 2-4 for more information on publishing triggers.

Even if you publish a trigger, target data changes caused by the trigger on the current connection are not sent to the source at refresh time. (The exception is the inadvisable scenario in which the data changes are in a different connection from the refresh operation.) The synchronized system relies on the source trigger to change data appropriately. Changes caused by the source trigger are then sent back to the target at the end of the refresh operation. Note that this implies that if a trigger acts differently on the source than on the target, only the source trigger actions persist after the refresh operation completes. Additionally, not sending trigger-initiated data on refresh helps reduce the amount data that must be sent to the source.
Synchronizing Databases

While typically, target trigger actions are not sent to the source at refresh time, if the trigger performs work on a different connection than the default at the time the trigger fired, that work is sent to the source at the next refresh.

**NOTE:** Unless it is absolutely necessary, do not force your application to send target trigger actions to the source. It is best to preserve the atomicity of triggers and work units by not allowing them to open separate connections. Among other potential problems, multiple connections to the same database may cause a deadlock.

**Refreshing Copied Sources and Targets**

**NEW:** Cloudsync version 3.0 prevents refreshes between copied sources and targets that could cause the system to become inconsistent.

A synchronization system includes a single source database, a set of target databases, and a set of copy files. The Cloudscape synchronization software automatically maintains the necessary relationships between these components. To ensure proper operation, follow these simple rules:

- Do not clone a target database and refresh both the original target and the cloned target. To make a new target database, always start from a copy file. It is okay to move a target database.
- Do not clone a source database and refresh targets against both the original source and the cloned source. It is okay to move the source database.

If either rule is violated, Cloudscape detects the situation and reports an error during refresh.

When you recover a source database from a backup, any transactions that occurred after the backup are lost. Therefore, a target that performed a refresh during this window may not refresh again. Therefore, to minimize the impact of a source failure, we strongly recommend you make frequent backups of the source.

**Refresh Security**

Cloudsync allows you to protect the source from unwanted access or modification by targets. You can control a target’s ability to access and modify the source. You
can also limit access to the targets themselves. See these sections for more information:

- “User Authentication and User Authorization” on page 4-9
- “Refresh Security Properties” on page 4-9
- “Refreshing Using SSL” on page 4-10
- “Limitations of the Security Model” on page 4-11
- “Security Configuration Example” on page 4-11
- “Work Units” on page 5-3

User Authentication and User Authorization

User authentication means using an authentication service to validate a user’s name and password before permitting that user access to the system. Cloudscape provides a built-in authentication service, and it also allows you to use an external service such as LDAP. You can also build your own authentication service. User authorization means the degree of database access granted a user once she has entered the system. Before setting up user authentication and user authorization in a synchronized system, please become thoroughly familiar with the user authentication and user authorization sections of the Cloudscape Developer’s Guide.

You can set user authentication and user authorization on both source and target databases. Direct connections to either a source or a target database are subject to the user authentication and user authorization requirements you have set. However, in the current release, the source database does not authenticate users during the refresh operation. The source must “trust” the authentication at the target. User authorization does apply at the source during the refresh, based on the name of the user executing the refresh.

Refresh Security Properties

At both source and target databases, you can specify the database’s default authorization connection mode: no access, read-only access, or full access. The property that specifies this is cloudscape.database.defaultConnectionMode, which is set to full access by default. If the default connection property does not authorize full access, you can specify a list of users authorized for full access. The property that specifies the list of full access users is cloudscape.database.fullAccessUsers. The user names can be either delimited or non-delimited.
For the source database, you can set these properties in the *cloudscape.properties* file, but it is preferable to set them directly in the database using the *COM.cloudscape.database.PropertyInfo* class. To set these properties in the target, publish them using the ADD TARGET DATABASE PROPERTY clause of CREATE PUBLICATION or ALTER PUBLICATION.

**NEW:** In version 3.0, a full-access user at the target can directly set target properties using *PropertyInfo.SetDatabaseProperty*.

See “Security Configuration Example” on page 4-11 for examples of setting these properties.

By default, the refresh operation connects to the source using the same user name as the target connection. If full access is not the source default, the source checks to see whether the target user name is on the list of full access users. If it is not, the refresh is not allowed to change source data (though users at the target may still be able to read source changes if the source and target are configured to allow it).

It may be impractical for the source to maintain a list of every user name. If this is the case, you can make the target always connect to the source using a given user name, regardless of the name used to connect to the target. To do this, publish the *cloudscape.database.sourceUser* property to the target. A full-access user at the target can modify this property for that target.

You can specify a target user authentication service by publishing the *cloudscape.authentication.provider* property to the target or by having a full-access user set it at the target.

### Refreshing Using SSL

If you use a server framework that supports SSL, you can configure target databases to enable the standard JDK URL support for POST messages and thus make SSL use possible. To do this:

- Set the *cloudscape.DataComm.url* property to *true* for target databases (easiest done through publishing this property) or by setting *cloudscape.listener.synchronization.address* to an address that starts with *https*.
- Run the source database in a server framework that supports SSL and configure the framework for SSL. Cloudconnector is an example of a server framework that supports SSL.
- Run target databases in a JVM that supports SSL.
NOTE: Currently only browser JVMs support SSL.

- Set cloudscape.listener.synchronization.address to the correct value for SSL for the server framework.

Limitations of the Security Model

Cloudsync’s current security model has some important limitations:

- **The source must trust the target to authenticate the user.**
  Passwords are not sent from target to source during the refresh, since they cannot currently be encrypted. Therefore, the source does not perform user authentication during the refresh procedure.

- **Published Target passwords set at the database level are not secure.**
  Passwords published as database properties are stored as clear text in the system tables.

Here are some suggestions for enhancing security given Cloudsync’s current security model:

- If there is no administrator at the targets, then either:
  - Use LDAP authentication at the targets. (This is practical only if target users can connect to the same LDAP server as the source when logging on to the target.)
  - Develop your own user authentication scheme for targets. See the *Cloudscape Developer’s Guide* for information on authentication schemes.
  - Do not authenticate users at the target.

- If targets have a local administrator and permit multiple connections, then it may be safe to use Cloudscape’s built-in authentication service, provided the local administrator maintains close tabs on the targets.

- Use workUnitOnly publications. For more information, see “cloudscape.synchronization.workUnitOnly” on page 12-16.

- Encrypt target databases.

Security Configuration Example

This section describes a scenario of configuring user authentication and user authorization at source and target. The scenario assumes that:
Synchronizing Databases

- you are using LDAP user authentication at both source and target.
- The same LDAP server is available to both source and target when the target user connects to the local database.
- there is a user “dba” at the source that already exists in the LDAP server.
- the user authentication server is called appserver:389.
- the LDAP search base is oakland.cloudscape.com.
- the following files are in the source’s class path:
  - jndi.jar
  - ldap.jar
  - providerutil.jar
- The following lines are in the source’s cloudscape.properties file:

  ```properties
  cloudscape.system.bootAll=true
  cloudscape.connection.requireAuthentication=true
  cloudscape.authentication.provider=LDAP
  cloudscape.authentication.server=appserver:389
  cloudscape.authentication.ldap.searchBase=ou=*
oakland.cloudscape.com
  ```

  This scenario allows anyone approved by the LDAP user authentication service to connect to the target. It assumes the same LDAP server and search base at both source and target. Any target created from the publication can modify the source during the refresh, regardless of which user connected to the target. Only user “dba” and users at targets created from this publication (or other publications that set sourceUser to “safeUser”) can see or alter source data.

  At the source, set the user authorization properties as follows (lines are broken for readability):

  ```java
  CALL (CLASS COM.cloudscape.database.PropertyInfo).
  setDatabaseProperty('cloudscape.database.defaultConnectionMode', 'noAccess')
  CALL (CLASS COM.cloudscape.database.PropertyInfo).
  setDatabaseProperty('cloudscape.database.fullAccessUsers','dba,safeUser')
  ```

  Add the properties to the publication that creates the target (you could also add them to a CREATE PUBLICATION statement).

  **NOTE:** Lines are broken for readability. Type each statement on its own line.

  ```sql
  ALTER PUBLICATION securePub
  ADD TARGET DATABASE PROPERTY
  cloudscape.database.propertiesOnly='true'
  ```
ADD TARGET DATABASE PROPERTY
cloudscape.connection.requireAuthentication='true'

ADD TARGET DATABASE PROPERTY
cloudscape.database.sourceUser='safeUser'

ADD TARGET DATABASE PROPERTY
cloudscape.database.defaultConnectionMode='fullAccess'

ADD TARGET DATABASE PROPERTY
cloudscape.authentication.provider='LDAP'

ADD TARGET DATABASE PROPERTY
cloudscape.authentication.server='appserver:389'

ADD TARGET DATABASE PROPERTY
cloudscape.authentication.ldap.searchBase= 'o=oakland.cloudscape.com'

You can see the property settings at the target using the following command:

VALUES PropertyInfo.getDatabaseProperties().toString()

Note that this does not verify that the properties were set correctly.
This chapter covers the following topics:

- “Updates from Multiple Targets” on page 5-1
- “Controlling Update Behavior” on page 5-2
- “Changed-Value Updates” on page 5-3
- “Work Units” on page 5-3
- “SQL-J Extensions for Work Units” on page 5-6
- “Creating a Work Unit” on page 5-6
- “Invoking a Work Unit” on page 5-7
- “Specifying the Connection in a Work Unit” on page 5-8
- “Transactions and Work Units” on page 5-9
- “Error Handling in Work Units” on page 5-9
- “Altering Work Unit Methods” on page 5-10

Updates from Multiple Targets

All durable data changes in a Cloudscape synchronization system are handled at the source. Since source data replaces target data after the refresh operation, the normal multi-user concurrency and transaction control mechanisms of the source database prevent the synchronized system from becoming inconsistent.

The target “screens” the transactions sent to the source: if a transaction fails at the target, it is not sent to the source at refresh time. Since target and source data are
usually not very different, updates that succeed at the target usually succeed at the source as well.

Of course, provisional transactions that succeed at the target can still fail at the source. This typically occurs because another target has changed the source data since this target’s last refresh. For example, an update from a different target may have reduced inventory enough to cause the current transaction to violate a source check constraint. (An update from a target can also fail if the system administrator or other user has updated the source database directly.) Transactions are in conflict if applying both of them at the source would violate a database consistency rule.

Conflicts are simpler to manage in Cloudsync than in many other synchronized systems. Cloudsync’s hub-and-spoke topology allows conflicts to be resolved between a single target and the source, rather than among multiple databases. This is similar to how updates are handled in a centralized client/server application.

Transaction failure is not the only issue when data changes come from many targets. You may want to specify how changes should be handled even when they don’t fail. For example, you may want to send a message when an update causes inventory to drop too low, even if no transaction has yet failed.

The rest of this chapter shows the simple, robust way that Cloudsync provides to handle updates that come from different parts of the system.

**Controlling Update Behavior**

Cloudsync provides two types of updates:

- **Changed-Value Updates**
  The default strategy, requiring no additional coding on your part. Use only when conflicts are not expected.

- **Work Units**
  Let you determine exactly how you want the update to be handled. Use when there are possible conflicts or when you want to take special action, such as sending an email message.
Changed-Value Updates

By default, the data changed by a target transaction simply replaces the current source data during the refresh unless a constraint prevents it. The value of a column in a given row in the target database replaces the value in the corresponding row in the source database.

There are situations in which Cloudsync’s default update strategy works well; for example:

- All changes are made directly at the source; the targets are all read-only.
- There is only one read-write target and the source is never updated except by that target.
- No two targets change the same data.

In these conflict-free cases, simply make changes to the target database (if appropriate), then run a refresh to send changes to the source or to receive updated changes from the source. When deciding whether to use changed-value updates, consider whether the source database will be upgraded. In these situations, work unit updates are preferred.

Work Units

A work unit is a Cloudsync entity consisting of logical operations that update data (or perform other operations) upon refresh. A work unit consists of one method at the target and one method at the source. These source- and target-side methods may be identical, or they may differ because the target database may not contain all of the source database’s dictionary objects or data, or for any other reason useful to the application. Work units provide a way to handle conflicting updates from multiple targets using logical operations rather than just blindly copying data.

This section discusses the following topics:

- “Understanding Work Units” on page 5-4
- “Work Units are Applied Provisionally” on page 5-4
- “Work Units Operate from Target to Source” on page 5-5
- “Work Units are Method Aliases” on page 5-5
- “Benefits of Work Units” on page 5-5

NEW: To increase security, you can publish the cloudscape.synchronization.workUnitOnly property which forces a target
Updates and Work Units

to accept only work-unit updates. Specifying work unit-only updates also provides more flexibility when upgrading the source database. See “cloudscape.synchronization.workUnitOnly” on page 12-16 for details.

Understanding Work Units

When conflicts are possible, you may not want a target value to simply replace the source value. For example, consider a Web-based toy sales application. Say that customers at Target A and Target B each issue a refresh at the beginning of the business day. At 9:00 a.m., both targets show that there are 10 available Barbie dolls. At 9:05, a customer using Target A orders three Barbies, reducing the number of Barbies known to the target to seven, then issues a refresh. The source database then shows that seven Barbies are available. At 9:10, a customer using Target B orders one Barbie, changing the apparent availability of this doll on Target B to nine. Target B then refreshes, causing the source database to incorrectly show the number of available Barbies as nine, rather than reducing it to six. This update causes the source to incorrectly record inventory. Changed-value updates from the target cannot handle this situation correctly; for example, using changed-value updates, you cannot tell the source to reduce the inventory of Barbie dolls by three.

To handle conflicting updates from multiple targets, Cloudsync allows you to use logic rather than forcing you to copy data. In the case of the toy sales application, each target can invoke a procedure telling the source to reduce its inventory by a certain amount, rather than just copying the value indicating the number of Barbie dolls in stock. This procedure could also tell the source to back-order an item if there were none available, and it could say what to do if an application-defined exception occurred.

Work Units are Applied Provisionally

Like changed-value updates, the target-side work unit method is applied provisionally to the target at the time it is invoked. Also like changed-value updates, the target-side work unit method is under transactional control and thus either provisionally commits or rolls back. Unlike changed-value updates, however, the source-side method, rather than the provisional target data, is applied durably at the source at refresh time.

For more information, see the section “Transactions and Work Units” on page 5-9.
**Work Units Operate from Target to Source**

Work units determine how target data is written to the source, but they have no effect on how source data is written back to the target. The refresh operation uses changed-value updates to replace the target with a more recent version of the source. This preserves agreement even if the work unit makes different changes each time it is run. Because refresh sends work unit logic from target to source, and consistent data from source to target, Cloudsync’s synchronization strategy is called LUCID—logic up, consistent information down.

Like other target changes, the work unit’s target-side method has only a temporary effect on the target data. The results of the target-side method remain on the target only until the end of the refresh cycle, since the results of the source-side method are written back to the target at that time.

Since work unit methods are normal Java code, they are not restricted to database updates: for example they can also send an email message.

**Work Units are Method Aliases**

Work units are essentially synchronization-enabled method aliases. (See the *Cloudscape Reference Manual* for more information on method aliases.) When you call a non-synchronization method alias, a single method is called. When you call a work unit, two methods are called: the target method is called when the work unit is invoked at the target, and the source method is called when the transaction is applied at the source during the refresh. Parameters invoked with the target work unit method are also available to the source method.

Like method aliases, work units:

- are registered with a CREATE WORK ALIAS statement
- are invoked using the CALL or VALUES statements
- are implemented as public static Java methods
- can return values

Unlike other static methods called by SQL-J, work units are atomic, meaning that commits and rollbacks are not permitted in either method of the work unit.

**Benefits of Work Units**

Work units provide the following benefits:
• They allow you to apply application- or business-level logic rather than database-level logic during a refresh.
• They allow you to handle conflicts between target changes and source data.
• They provide transactional atomicity to the actions in the work unit.
• They provide flexible error handling.
• They provide increased flexibility for upgrades to the source database.

SQL-J Extensions for Work Units

Work units are registered using CREATE WORK ALIAS at the source. They are unregistered using DROP WORK ALIAS at the source. You must publish a work unit to any target that will use it.

Work units are invoked using one of the following statements:

• CALL
  if the work unit does not return a value or if the return value is not important
• VALUES
  to use the work unit’s return value

Work units for a target must have been included (using the ADD WORK ALIAS clause of CREATE PUBLICATION or ALTER PUBLICATION) in the publication that created that target. As with other explicitly published dictionary objects, you cannot drop a work unit unless you drop all publications that contain it.

See Chapter 9, “Cloudsync Commands Reference”, for details on these statements.

Creating a Work Unit

Work units are registered at the source using CREATE WORK ALIAS, then are put in the publication that creates the target.

The syntax for CREATE WORK ALIAS is:

```
CREATE WORK ALIAS WorkAliasName
     FOR SourceMethodName [, TargetMethodName ]
```

The SourceMethodName and TargetMethodName are of the form classname.methodname (class aliases may be used).
If you do not specify the target method name, it is assumed to be the same as the source method name. Even if the source and target methods have the same name, the code in the methods may be different.

Because the work unit is an alias, publishing the work unit does not automatically copy any classes to the target. (To automatically update target classes when source classes are updated, put the classes into jar files in the source database. See “Distributing Application Code Within the Database” on page 7-2 for details.)

If the source and target methods do not have the same number of parameters, the work unit will fail when applied at the source during the refresh.

Work units, method aliases, and user-defined aggregates (which are a type of method alias) occupy the same name space. You cannot create a work unit using an existing method alias name.

The following example registers a work unit:

```
CREATE WORK ALIAS recordSalesOrder
FOR COM.SalesApplication.HubProcessor.recordSalesOrder,
    COM.SalesApplication.TargetProcessor.recordSalesOrder
```

See “Work Unit Examples” on page 6-7 for examples of work units.

---

**Invoking a Work Unit**

To invoke a work unit, use the CALL statement or a VALUES statement at either the target or the source. Calling a work unit is syntactically similar to calling other methods in Cloudscape. Calling a work unit at the source is just like making source changes directly.

Use CALL if the work unit does not return a value or if you don’t use the return value. Use VALUES to see the work unit’s return value.

The syntax of CALL with a work unit is:

```
CALL WorkUnitName
    ( [ MethodParameter [, MethodParameter ]* ] )
```

The syntax of VALUES with a work unit is:

```
VALUES ( WorkUnitName
    ( [ MethodParameter [, MethodParameter ]* ] )
```

To see a work unit’s return value, fetch the first value of the statement’s result set.

For example:
Connection conn;
PreparedStatement ps;
ResultSet rs;

ps = conn.prepareStatement("VALUES (myWorkUnit())");
rs = ps.executeQuery();
rs.next();
int successIndicator = rs.getInt(1);

You can specify any number of parameters in a CALL or VALUES statement. The same parameter values are passed to both the target-side and source-side methods of the work unit. The rules for parameters are the same as for other SQL-J expressions. See the Cloudscape Reference Manual for details about parameter syntax.

If CALL or VALUES cannot find the work unit name or the Java method for the work unit, an error is raised. If the failure occurs at the target, a statement exception is raised. If the failure occurs at the source, an environmental error is raised and the refresh halts.

You can call one of the work unit’s methods directly, without using the work unit alias. If you do, it will behave like a non-synchronized method.

Specifying the Connection in a Work Unit

A work unit typically executes a query. To do that, it must know which connection to execute the query on.

To obtain a connection in a work unit, put a line like:

Connection c =
    DriverManager.getConnection("jdbc:cloudscape:;current=true");

within the work unit.

It is also permissible to pass getCurrentConnection() to CALL or VALUES as a parameter. SQL-J can correctly interpret getCurrentConnection() at refresh time.

When invoking work units, avoid using “?” parameters to represent connections. If the parameter does not represent the current connection, a runtime error is raised at the target.
Transactions and Work Units

Transaction control inside work units is the same as in methods issued inside queries. You cannot commit a transaction inside a method or a work unit invoked in an SQL-J statement.

By default, target work unit methods run at the target database’s isolation level, and source work unit methods run at the source database’s isolation level. You can override these defaults for individual SELECT statements within a work unit. See the Cloudscape Developer’s Guide for more information about setting isolation levels in a SELECT statement.

LOCK TABLE is not automatically propagated in either direction. You can, however, specify target-to-source table locking by putting LOCK TABLE into a work unit.

Error Handling in Work Units

If an exception in a work unit is raised and not caught, the CALL or VALUES statement and all its transactional work are rolled back. If the exception is caught, the work unit determines what action to take at the source.

Cloudsync provides two exceptions in COM.cloudscape.synchronization specifically for work units. These exceptions let you decide how you want error situations to affect your transaction. The exceptions are:

- **SkipTransactionSQLException**
  For consistency errors. Causes the provisional transaction to fail and allows the refresh operation to continue.

- **StopRefreshSQLException**
  For environmental errors. Causes the transaction to remain provisional and aborts the refresh.

Consistency errors and environmental errors are described in “Applying Provisional Transactions at the Source” on page 4-3. Work units let you determine when you want these errors raised. See “Work Unit Examples” on page 6-7 for examples of these exceptions.

**NOTE:** If the target-side method fails with an error, the source-side method does not run.
It is important not to catch deadlock and other transaction-severity errors in a work unit. It is safer to let the entire transaction roll back when a severe error is encountered. To find out whether an SQLException is severe enough to terminate the current transaction, use the COM.cloudscape.database.checkJBMSException method. The following code fragment shows how to use this method:

```java
public static void workUnit1( ... ){
    try {
        PreparedStatement ps = ...;
        ps.executeQuery();
    } catch ( SQLException se ) {
        checkJBMSException( se );
        // if we get here, the exception is mild
        // enough to continue
        ...
    } // end tryCatch
    ...
}
```

### Altering Work Unit Methods

You can invoke CALL or VALUES even if the source-side method has been altered. This gives you the flexibility to alter update procedures without changing target-side code.

The source reloads the class containing the source work unit method only when the server is restarted or the class is garbage collected (which is a characteristic of Java, not only of Cloudscape.) Therefore, if you change a source-side work unit method stored in the file system, the changes may not take effect immediately. A way of circumventing this problem is to put classes into the database using jar files, and replace the jar files as needed. See “Distributing Application Code Within the Database” on page 7-2 for details.

Any changes you make to a work unit method, even changes to its signature, are distributed to targets, as long as you use the following guidelines:

- Make sure that the new code behaves correctly when called by the older version of the application.
- Do not remove the older version of the method until the guaranteed refresh interval (plus a short lag time, generally two hours) has passed. (See
“cloudscape.refresh.guaranteedRefreshInterval” on page 12-15 for details.)
This chapter shows one approach to creating a distributed application using Cloudsync. The application in this chapter is much simpler than a real-world application would be, but it does illustrate the basic steps for creating a deployable Cloudscape synchronization application. It also discusses some issues of publication design.

- “Application Design Overview” on page 6-1
- “Design Guidelines” on page 6-2
- “Steps for Designing a Distributed System” on page 6-2
- “Work Unit Examples” on page 6-7
- “Publication Design Issues” on page 6-15
- “Creating Universal Keys” on page 6-18

Application Design Overview

Cloudscape synchronization is designed to make it easy to create distributed applications using intermittently connected databases. Designing a synchronized system using Cloudsync is not very different from designing a multi-user single-database system. Much of the logic in Cloudsync’s work units is the same as in a multi-user client/server database application—for example, validating a customer or checking whether enough items are available to fill an order.

There is one important difference between single-database and distributed applications. In a typical single-database client/server framework, the user is continually connected to the database and can provide immediate direction and
input. For example, if a user tries to order a toy that is not available, the application can immediately send the user a message asking whether to choose another toy or cancel the order. In an intermittently connected distributed system, on the other hand, the changes a user makes at a target may apply at the source some time later, making immediate user feedback impractical.

**Design Guidelines**

The user in a distributed application does not resolve conflicts in real time. Because of this, the following guidelines may be useful when designing an application:

- Include source work unit logic to automatically handle cases that, in an interactive environment, would prompt the user.
- Store information in the database on the status of each update so that someone can later decide how to handle the consequences of the update.
- Make sure that any order status information (e.g., filled, back-ordered, or rejected) is in the publication and is therefore copied to the target. After the target refreshes, the target application can see this status information by querying the appropriate table.

In short, a typical Cloudscape synchronization application should perform the following steps at a target:

1. Apply changes at the target by issuing work units.
2. Issue a refresh to synchronize target and source.
3. Look at the synchronized target for the status of the changes applied in step 1.

**Steps for Designing a Distributed System**

Before creating the databases or writing code:

1. State the purpose of the application.
2. List the logical operations that the application will perform.
   
   *Represents the application’s work units*
   
3. For each operation, list the data that the application needs.
   
   *Determines the source database design and the tables in each publication*
For each operation, list the data that an individual target needs.  
*Determines the parameters passed to the publication*

List the inputs to each operation.  
*Represents the parameters passed to the work units*

List the possible outcomes of each operation.  
*Determines the status information stored in the database*

Describe how to represent each possible outcome in the database. 

**NOTE:** If you can design your application so that no conflicts are possible, you may not need to perform steps 5, 6, and 7. If an update cannot cause a conflict, and if the only action you want to perform is updating the same data at target and source, you do not need to write a work unit for it.

- “Step 1: Defining the Application” on page 6-3
- “Step 2: Listing the Operations” on page 6-4
- “Step 3: Describing the System-Wide Data” on page 6-4
- “Step 4: Describing the Target Data” on page 6-5
- “Step 5: Listing the Inputs” on page 6-6
- “Step 6: Listing the Outcomes” on page 6-6
- “Step 7: Representing the Outcomes in the Database” on page 6-7

### Step 1: Defining the Application

This chapter uses a very simple toy sales order scenario. There are two sales representatives, each at her own target, and two types of toys available. Each salesperson does a refresh to see how many of each toy are in stock, and to get the latest price information. The salesperson places an order for a single item, entering the name of the customer, the item she wants, and the quantity. Each salesperson has access to her own orders only. The salesperson can also send a lead to another salesperson. The central office keeps track of valid salespeople, and updates from unapproved salespeople are rejected.

Please note that, for simplicity’s sake, this application puts the source and target directories under the same directory on the same machine. A real deployed application would probably put the source database into a server framework.
Step 2: Listing the Operations

The two operations needed for this simple scenario are:

- place an order
- send a lead (the name of a potential customer)

You will write a work unit for each operation. The work unit code is shown in “Work Unit Examples” on page 6-7.

Step 3: Describing the System-Wide Data

This application requires the following data about each toy:

- name of toy
- unit price
- quantity available

The application also needs the following data about each order:

- customer name
- name of toy
- sales representative ID
- status

Since salespeople (and people at the central office) can send each other leads, the application uses the following data about each lead:

- the salesperson who receives the lead
- the salesperson who sends the lead
- the name of the customer

The application needs access to a list of valid salespeople to determine whether a lead is deliverable and whether the salesperson sending an update is valid. For security purposes, it would be useful to refuse update permission to a salesperson who has been let go. Therefore, the application maintains the following information about each salesperson:

- the sales representative’s ID
- last name
- first name

The TOYS, ORDERS, LEADS, and SALESREPS tables will contain this data. They are created as follows:
CREATE TABLE toys (  
toy VARCHAR(30) CONSTRAINT toys_pk PRIMARY KEY,  
price DECIMAL(5,2),  
quantity INT)

CREATE TABLE orders (  
  orderId BIT VARYING(128) CONSTRAINT orders_pk PRIMARY KEY,  
salesRepID INT,  
customer VARCHAR(50),  
toy VARCHAR(50),  
quantity INT,  
status VARCHAR(15))

CREATE TABLE leads (  
  leadID BIT VARYING(128) CONSTRAINT leads_pk PRIMARY KEY,  
sentToRepID INT,  
sentFromRepID INT,  
customer VARCHAR(50))

CREATE TABLE salesReps (  
salesRepID INT CONSTRAINT reps_pk PRIMARY KEY,  
lastName VARCHAR(20),  
firstName VARCHAR(20))

NOTE: See “Creating Universal Keys” on page 6-18 for an explanation of the BIT VARYING column used as a primary key in some of these tables.

Step 4: Describing the Target Data

Each target is restricted to one salesperson’s orders and leads. Therefore, the publication uses parameters to restrict each target accordingly.

The publication to create a target is:

CREATE PUBLICATION toySales (salesRepID INT)  
ADD TABLE toys  
ADD TABLE orders  
  WHERE salesRepID = ?salesRepID  
ADD TABLE leads  
  WHERE sentToRepID = ?salesRepID  
ADD METHOD ALIAS getKey  
ADD WORK ALIAS placeOrder  
ADD WORK ALIAS sendLead
NOTE: The getKey method alias is shorthand for the universal key generator described in “Creating Universal Keys” on page 6-18. The getKey method alias and the two work alias definitions are shown in the section “Work Unit Examples” on page 6-7.

Step 5: Listing the Inputs

The parameters in the placeOrder work unit are:

1. a universal key for the orderID (see “Creating Universal Keys” on page 6-18)
2. the sales representative’s ID
3. the customer name
4. the toy name
5. the number of toys ordered

The parameters to the sendLead work unit are:

1. a universal key for the lead
2. the ID of the salesperson to whom the lead is being sent
3. the ID of the salesperson sending the lead
4. the customer name

Step 6: Listing the Outcomes

This application represents provisional (pre-successful-refresh) outcomes differently from durable (post-successful-refresh) ones. This gives the user a better sense of the actual status of her request. The possible outcomes of placing an order are:

- **Provisional**
  - There appear to be items available, so we expect to order one.
  - There appear to be no items available, so we expect to back-order one.
  - We want to order a toy that is not carried.
- **Durable**
  - The item is available and has been ordered.
  - The item is not available and has been back-ordered.
  - The ordered item is not carried.
For simplicity, the application does not track the status of leads.

**Step 7: Representing the Outcomes in the Database**

The STATUS column of the ORDERS table represents the possible outcomes described in step 5. Since there are six possible statuses of an order, this column can have one of six values:

- WILL ORDER
- WILL BACKORDER
- ORDERED
- BACKORDERED
- TOY UNKNOWN
- TOY UNAVAILABLE

Although in this example the values in the STATUS column represent the outcomes of only one work unit, status values typically represent the outcomes of more than one work unit. For example, if there were also a cancelOrder work unit, STATUS might also permit the values WILL CANCEL and CANCELED.

Note also that there are many ways of representing status in a database. You do not necessarily need a STATUS column. You might instead have an EXPECTED_DELIVERY_DATE column that is assigned a later date if the product is backordered. Doing this would not overtly show the user whether an order is provisional or durable, but this may not be necessary in some applications.

**Work Unit Examples**

- “The placeOrder Work Unit” on page 6-8
- “The sendLead Work Unit” on page 6-9
- “Work Unit Example Code” on page 6-10

Since multiple salespeople can alter the same source data when placing orders or sending leads, you should put the logic that places orders and leads into work units. For ease of deployment, source methods have been put into a sourceWork class and target methods into a separate targetWork class.

The work aliases for this example are created at the source as follows:
CREATE WORK ALIAS placeOrder
FOR sourceWork.makeSaleAtSource,
targetWork.makeSaleAtTarget

CREATE WORK ALIAS sendLead
FOR sourceWork.sendLeadAtSource,
targetWork.sendLeadAtTarget

This example also contains a method alias for the getUniversalKeyValue method (described in “Creating Universal Keys” on page 6-18.) This method alias is created at the source as follows:

CREATE METHOD ALIAS getKey
FOR COM.cloudscape.util.KeyGen.getUniversalKeyValue

The placeOrder Work Unit

The placeOrder work unit places an order for a given quantity of a toy.

As mentioned in “Step 5: Listing the Inputs” on page 6-6, the parameters of the placeOrder work unit are: the order ID (a generated universal key), the sales representative ID, the customer name, the toy name, and the quantity ordered.

As with all work units, the target-side method applies when the work unit is invoked, and the source-side method applies when the target refreshes.

The Target-side Method

The target-side method first checks that the ordered toy is valid. If it is not, the status is set to TOY UNKNOWN. The transaction is not aborted because the item may have been added at the source since this target’s last refresh, so the order could still be filled. The method then checks the number of units available as of the last refresh, and if the order is likely to succeed, sets the status to WILL ORDER. Otherwise, it sets the status to WILL BACKORDER.

The Source-side Method

The source-side method does more than the target-side method. This is typical because the source usually contains more information than the target.

First, the source-side method checks that the sender is a valid salesperson. If the salesperson is invalid, the refresh is aborted by raising a StopRefreshSQLException. This makes it impossible for the target to refresh again until the source SALESREPS
table (which is not published to the target) adds the salesperson’s ID. (Production code would of course enforce security in a more sophisticated way; this example illustrates that raising a StopRefreshSQLException is a good way to disable updates from a target while that target’s security is questionable.)

The source-side method then checks that the toy exists on the list of items, and sets status to TOY UNAVAILABLE if it does not. The method then checks the number of items in stock, which may differ from the results of the provisional target transaction due to conflicting updates from other targets. The method then assigns a durable value (ORDERED or BACKORDERED) to the STATUS column of the ORDERS table, and this value is copied to the target at the end of the refresh.

**Using the placeOrder Work Unit**

As an example of using this work unit, the following SQL-J statement places an order from sales representative number 1 for three Beanie Baby Elephants for customer “Tucker”:

```sql
CALL placeOrder(getKey(),1,'Tucker','beanie baby elephant',3)
```

**The sendLead Work Unit**

The sendLead work unit sends a customer name to a specified salesperson. The parameters of the sendLead work unit are: a universal key (the lead ID), the destination salesperson ID, the sending salesperson ID, and the lead.

**The Target-side Method**

sendLead is an example of a work unit that acts very differently at the target and at the source. Only leads assigned to a given salesperson are published to that salesperson, and it is assumed that the salesperson sending the lead has no interest in tracking it once she has sent it. Therefore, the target-side method does almost nothing except confirm the transaction. No status on the lead is maintained at the target. If there are problems with the lead, it is assumed the system administrator will handle them.

**The Source-side Method**

As in the source-side method of placeOrder, the method checks to see that the sender is valid. If she is not, the method aborts the refresh by raising a
StopRefreshSQLException, which disables the target from further refreshes until the database administrator updates the SALESREPS table. The method then checks that the destination salesperson ID is valid, and aborts and logs the transaction by raising a skipTransactionSQLException if it is not. It is assumed that the system administrator will check the log for leads that have been sent to invalid salespeople, and will take appropriate action. Finally, it adds the lead to the LEADS table. Only the leads intended for the target issuing the refresh are sent to that target at the end of the refresh.

Using the sendLead Work Unit

As an example of using this work unit, the following SQL-J statement sends the lead “Tucker” from salesperson number 2 to salesperson number 1:

\[
\text{CALL sendLead(getKey(),1,2,’Tucker’)}
\]

If the system administrator, rather than a sales person, wants to send a lead, he can run sendLead directly at the source. (The source might be salesperson number 0.) If a work unit is invoked at the source, the target-side work unit code never runs; the source database changes are simply sent to the target upon the next refresh.

Work Unit Example Code

This section shows target and source code for the placeOrder and sendLead work units. Note that this code is not optimized for performance; replacing the queries and prepared statements with stored prepared statements would improve performance.

The following code is for the target-side work unit methods:

```java
import java.sql.*;
import COM.cloudscape.synchronization.*;

public class targetWork {
    public static void makeSaleAtTarget(
        byte[] orderID, int salesRepID, String customer,
        String toy, int quantityOrdered) throws SQLException {
            Connection conn = DriverManager.getConnection(
                "jdbc:cloudscape::current=true");
            Statement s = conn.createStatement();
            String status = "";
```

Cloudscape Synchronization Guide
int quantityAvailable;

// find out the number of toys available as of the last refresh
ResultSet rs = s.executeQuery("SELECT quantity FROM toys WHERE toy = '"+toy+"'");
if (!rs.next()) {
    quantityAvailable = 0;
    status = "TOY UNKNOWN";
} else {
    quantityAvailable = rs.getInt(1);
}
rs.close();

// set status provisionally
// to WILL ORDER or WILL BACKORDER
if (quantityAvailable >= quantityOrdered) {
    s.executeUpdate("UPDATE toys SET quantity=quantity-"+quantityOrdered+
    " WHERE TOY='"+toy+"'");
    status = "WILL ORDER";
} else if (status != "TOY UNKNOWN") {
    status = "WILL BACKORDER";
}

PreparedStatement ps = conn.prepareStatement("INSERT INTO orders VALUES(?,?,?,?,?)");
ps.setBytes(1, orderID);
ps.setInt(2, salesRepID);
ps.setString(3, customer);
ps.setString(4, toy);
ps.setInt(5, quantityOrdered);
ps.setString(6, status);
ps.executeUpdate();
s.close();
conn.close();
}

public static void sendLeadAtTarget
    (byte[] leadID, int sentToRepID,
The following is the code for the source-side work unit methods:

```java
import java.sql.*;
import COM.cloudscape.synchronization.*;

public class sourceWork {

  public static void makeSaleAtSource(
      byte[] orderID, int salesRepID, String customer,
      String toy, int quantityOrdered) throws SQLException {
    Connection conn = DriverManager.getConnection(
        "jdbc:cloudscape:;current=true");
    Statement s = conn.createStatement();
    String status = "";
    int quantityAvailable;

    // Abort refresh if salesRepID is invalid
    ResultSet repsRS = s.executeQuery(
        "SELECT salesRepID FROM salesReps WHERE salesRepID = "+salesRepID+""");
    if (!repsRS.next())
      throw StopRefreshSQLException.stopRefreshSQLException(
        "Your sales rep ID "+salesRepID+
        " is invalid. Please contact your system administrator.");
    repsRS.close();

    // Find out how many toys are available
    ResultSet rs = s.executeQuery(
        "SELECT quantity FROM toys WHERE toy = "+toy+"";
    if (!rs.next())
      throw StopRefreshSQLException.stopRefreshSQLException(
        "No toy available for "+toy+". Please contact your system administrator.");
    status = "Status: "+rs.getString("quantity");
    quantityAvailable = rs.getInt("quantity");
    rs.close();
    System.out.println(status);
  }
}
```
"SELECT quantity FROM toys WHERE toy = '"
+toy+'''');
if (!rs.next()) {
    quantityAvailable = 0;
    status = "TOY UNAVAILABLE";
} else {
    quantityAvailable = rs.getInt(1);
}
rs.close();

// set status durably
// to ORDERED or BACKORDERED
if (quantityAvailable >= quantityOrdered) {   
    s.executeUpdate("UPDATE toys SET quantity=quantity-" +
    quantityOrdered +
    " WHERE TOY=’" + toy + '''");
    status = "ORDERED";
} else if (status != "TOY UNAVAILABLE") {
    status = "BACKORDERED";
}
PreparedStatement ps = conn.prepareStatement
    ("INSERT INTO orders VALUES(?,?,?,?,?,?)");
ps.setBytes(1, orderID);
ps.setInt(2, salesRepID);
ps.setString(3, customer);
ps.setString(4, toy);
ps.setInt(5, quantityOrdered);
ps.setString(6, status);
ps.executeUpdate();
s.close();
conn.close();
}

public static void sendLeadAtSource
    (byte[] leadID, int sentToRepID,
      int sentFromRepID, String customer)
    throws SQLException {
    Connection conn = DriverManager.getConnection(  
        "jdbc:cloudscape::current=true");
Designing Synchronized Applications

Statement s = conn.createStatement();

// Abort refresh if lead sent from a non-qualifying sales rep
ResultSet repsRS = s.executeQuery(
    "SELECT salesRepID FROM salesReps WHERE salesRepID = '"
    +sentFromRepID+"'");
if (!repsRS.next()) throw
    StopRefreshSQLException.stopRefreshSQLException(
        "Invalid sales rep ID " + sentFromRepID + 
        " sent a lead to sales rep " + sentToRepID + ".");
repsRS.close();

// Abort transaction if lead sent to a non-existent sales rep
ResultSet assignedRepsRS = s.executeQuery(
    "SELECT salesRepID FROM salesReps WHERE salesRepID = '"
    +sentToRepID+"'");
if (!assignedRepsRS.next()) throw
    SkipTransactionSQLException.skipTransactionSQLException(
        "Lead assigned to invalid sales rep " + sentToRepID);
assignedRepsRS.close();

// Add lead to leads table
PreparedStatement ps = conn.prepareStatement(
    "INSERT INTO leads VALUES(?,?,?,?)");
ps.setBytes(1, leadID);
ps.setInt(2, sentToRepID);
ps.setInt(3, sentFromRepID);
ps.setString(4, customer);
ps.executeUpdate();

s.close();
conn.close();
}
Publication Design Issues

- “Making Targets Small and Efficient” on page 6-15
- “Target-Side Validation” on page 6-16
- “Handling Published Foreign Keys” on page 6-17

Making Targets Small and Efficient

By making each target as small as possible, you may reduce the time required to:

- send the target or copy file through the network or Internet.
- perform local queries and updates.

Targets without extraneous data are also more secure.

To make targets small, whenever possible:

- Have the table’s WHERE clauses restrict on primary keys or on indexed (and published) fields
- Design your tables so that you can make each WHERE clause as restrictive as possible.

Another technique to make targets small is to put methods in the publication’s WHERE clause. This is especially helpful when you don’t know beforehand how many parameter values will be passed to the publication. Java collections like ArrayList and Vector provide contains, an especially useful method for this purpose.

For example, suppose you want to create a publication like the one defined in “Step 4: Describing the Target Data” on page 6-5, except that it contains data for multiple salespeople. If you know the number of salespeople whose data you want in the target, you can simply pass it more parameters. For example, if you know that there are three salespeople’s data in the target, you can define the publication as follows:

```
CREATE PUBLICATION toySalesForThreeReps
    (salesRep1ID INT, salesRep2ID INT, salesRep3ID INT)
ADD TABLE toys
ADD TABLE orders
    WHERE (salesRepID IN (?salesRep1ID, ?salesRep2ID, ?salesRep3ID))
ADD TABLE leads
    WHERE (sentToRepID IN (?salesRep1ID, ?salesRep2ID, ?salesRep3ID))
ADD METHOD ALIAS getKey
ADD WORK ALIAS placeOrder
ADD WORK ALIAS sendLead
```
However, if you don’t know beforehand how many salespeople will be included in the publication, you cannot know how many parameter to pass to the publication.

Using methods in the publication’s WHERE clause can solve this problem. The following publication allows you to pass a vector containing any number of salesperson IDs as a parameter to the publication, and the publication will be optimally small:

```
CREATE PUBLICATION toySalesForMultipleReps
    (salesReps SERIALIZE(java.util.Vector))
ADD TABLE toys
ADD TABLE orders
    WHERE ?salesReps.contains(salesRepID)
ADD TABLE leads
    WHERE ?salesReps.contains(sentToRepID)
ADD METHOD ALIAS getKey
ADD WORK ALIAS placeOrder
ADD WORK ALIAS sendLead
```

**NOTE:** When using methods in publications, make sure that the method always returns the same results for the same data. A non-deterministic publication can cause incorrect data at a target. A method like `contains` is safe, but a method that, for instance, calls a random number generator is not. For the same reason, it is advisable not to include SQL-J statements in the method, since a query may return different results at different times.

**NOTE:** Using a method in the WHERE clause to reduce target size may reduce copy publication performance, since it may force a table scan on an indexed column.

### Target-Side Validation

The target should “screen” the source from unnecessary transactions. A transaction expected to fail at the source should fail first at the target. Otherwise, a transaction that succeeds at the target may unexpectedly disappear from the target after the refresh completes.

For the most part, this screening happens automatically. A WHERE clause of a CREATE PUBLICATION statement becomes a check constraint on a target created from the publication. Therefore, any target transaction that violates the publication’s WHERE clause fails at the target and never reaches the source.
However, there are cases in which a change at a target may unexpectedly fail at the source and may therefore disappear from the target’s user tables after a refresh. Two situations that can cause this are:

- **A target change violates an unpublished unique BTREE index.**
  Solution: Use unique constraints rather than unique BTREE indexes on published tables whenever possible, since unique constraints on published columns are automatically published. If you do use unique BTREE indexes, make sure to explicitly publish them.

- **A target change violates a constraint.**
  For example, in the following scenario, an update to the `maxBonus` field of the `compensation` table at a target would be rolled back after a refresh:

```sql
// Compensation table.
// Everyone either is on commission or gets a year-end bonus

CREATE TABLE compensation
(
    employeeID INT,
    departmentID INT,
    baseSalary FLOAT,
    commissionPercent FLOAT,
    maxBonus FLOAT
    CHECK ( commissionPercent NOT NULL OR maxBonus NOT NULL )
)
...

CREATE PUBLICATION salaried_employees( departmentID INT )
    ADD TABLE employees WHERE departmentID = ?departmentID
    ADD TABLE compensation
    ( employeeID, departmentID, baseSalary, maxBonus )
    WHERE departmentID = ?departmentID  ...
```

Solution: If you publish any column mentioned in a constraint, publish all other columns mentioned in the constraint as well. At the very least, if you do leave any columns that are mentioned in a constraint unpublished, make sure they evaluate to TRUE or NULL when default values are entered at the source.

### Handling Published Foreign Keys

A foreign key involves two tables. A foreign key constraint may be satisfied on a database but may be violated if parts of the primary or foreign key tables are filtered
out by a publication. Since a target database may be a subset of a source database, a foreign key constraint may succeed on the source but fail at the target.

If a publication contains a foreign key constraint and incompatible subsets of the two tables’ data, it may become impossible to refresh the target without violating the target’s foreign key constraint. If this happens, all future refreshes from this target will fail. Once the target is in this state, you must recreate the target with the proper subset of columns and data, or without the foreign key.

You can avoid the possibility of targets violating foreign key constraints by following these guidelines when creating publications:

- place no WHERE restrictions on the primary key table
  or
- have identical WHERE restrictions on the primary and foreign key tables.

If you create publications that violate these guidelines, be aware that target foreign key constraint violation may occur, and that you may lose the ability to successfully refresh the target.

Creating Universal Keys

One difficulty with occasionally connected distributed applications is that a key value that is unique at a single database may not be unique in all databases in the system. You may want to create a key that is unique to the whole synchronized system, even if key values get created at multiple targets. To do this, Cloudscape provides the `COM.cloudscape.util.KeyGen.getUniversalKeyValue` method. This method produces a random value of Java type `byte[]`, whose corresponding SQL type is BIT VARYING(128). This method is virtually certain to produce a value unique to the whole synchronized system.

For example, the following SQL-J code produces a table containing one universal key column and inserts a unique key value into that column:

```sql
CREATE TABLE mytable(mycolumn BIT VARYING(128) PRIMARY KEY)
INSERT INTO mytable VALUES
KeyGen.getUniversalKeyValue()
```

You can also create and publish a method alias for `getUniversalKeyValue` for work units to use. For example:
CREATE METHOD ALIAS getKey
FOR KeyGen.getUniversalKeyValue

This way, to invoke a work unit whose parameters are a universal key value and an INT, for example, the user at the target need only enter something like:

CALL myWorkUnit(KeyGen.getKey(), 5)

**NOTE:** Some tools provided by vendors other than Cloudscape may not adequately display a column of type BIT VARYING. If this is a problem in your application, you can generate a universal key of type String by using the `COM.cloudscape.util.KeyGen.getUniversalKeyStringValue` method instead. The string generated by this method is also virtually certain to be unique to the whole synchronized system. However, it is twice as big as the BIT VARYING value produced by `getUniversalKeyValue`. 
Designing Synchronized Applications
Typically, the source database runs in a server framework on a large machine and the targets run embedded in applications on small local machines.

Cloudsync lets you store application code in jar files within the synchronized databases. Target application code stored in the source database is automatically deployed when a target is created. If you modify the application at the source, the modifications are automatically distributed to the targets on refresh.

- “Configuring Source and Target Machines” on page 7-1
- “Distributing Application Code Within the Database” on page 7-2
- “Locally Configuring Target Database Properties” on page 7-4

### Configuring Source and Target Machines

This section tells how to configure source and target machines for Cloudscape synchronization.

### What the Server Needs

When running a source database on a server, you need access to the following directories and files:

- the Cloudscape synchronization library (*cloudsync.jar*)
- the source database
Deploying Synchronized Systems

- server framework directories and files (see the Cloudscape Server and Administration Guide for information)
- any unpublished .zip, .class, or .jar files the target application needs

**NOTE:** Some or all of this information may be stored in the database instead of external files. See “Distributing Application Code Within the Database” on page 7-2 for more information.

Note that target-side work unit methods and application code are not required on the server.

What the Target Machine Needs

The target database is typically embedded in an application running on a different machine from the source. The target machine must contain:

- the synchronization version of the standard Cloudscape library (cloudsync.jar)
- files for connecting to the server framework (see the Cloudscape Server and Administration Guide for information)
- Any unpublished .zip, .class, or .jar files required by your target application
- client.jar (only necessary if you are using one of the provided server frameworks: Cloudconnector or RmiJdbc)

Note that server-side work unit methods and application code are not required on the target machine.

The target machine also initially needs access to the copy file that will create the target database. The copy file may be downloaded to the target machine, but this is not required as long as you can specify the location of the copy file with an OS pathname from the target machine. Once the target database has been created from the copy file, the copy file itself is no longer needed.

Distributing Application Code Within the Database

You can use Cloudsync to distribute application code and code changes to targets. To do this:
1. Put the target application code into jar files in the source. (See the *Cloudscape Developer’s Guide* for instructions.)

2. Add the jar files to the publication that the target will subscribe to.

3. Add the `cloudscape.database.classpath` property to the publication and set it to include all published jar files.

The target you create with this publication will function just as if the application class files were deployed at the target. Additionally, changes to the published jar files at the source are automatically reproduced at the target when it refreshes, eliminating the need for local administration.

### Example of Jar Files in a Database

To illustrate including source and target application code in the database, consider the application described in Chapter 6, “Designing Synchronized Applications”. The following example shows how to put that application’s work units directly into source and target databases rather than external files. (Note that you could do exactly the same thing for application code other than work units.) This example assumes you have already created the system described in Chapter 6, “Designing Synchronized Applications”. (For more information on putting code into the database, see the *Cloudscape Developer’s Guide*.)

1. Before creating the databases, put the work unit classes into jar files, as follows:

   ```
   jar cf targetWork.jar TargetWork.class
   jar cf sourceWork.jar SourceWork.class
   ```

2. After creating the source database, but before creating the publication and targets, run the following SQL-J commands to put the jar files and properties into the source database:

   ```
   CALL (CLASS COM.cloudscape.tools.dbclasses).addJar(
      ‘APP’, ‘SourceWork’, ‘sourceWork.jar’)
   CALL (CLASS COM.cloudscape.tools.dbclasses).addJar(
      ‘APP’, ‘TargetWork’, ‘targetWork.jar’)
   CALL (CLASS COM.cloudscape.database.PropertyInfo).setDatabaseProperty
      (‘cloudscape.database.classpath’, ‘APP.SourceWork’)
   ```

**NOTE:** You could also use Cloudview to perform this and the remaining steps. For more information see the Cloudview online help.
Deploying Synchronized Systems

3 Create the publication described in “Step 4: Describing the Target Data” on page 6-5.

4 Include the target-side work unit code in the publication, as follows:

   ALTER PUBLICATION toysales
   ADD JAR FILE APP.TargetWork
   ADD TARGET DATABASE PROPERTY
       cloudscape.database.classpath='APP.TargetWork'

5 Create copy files:

   COPY PUBLICATION toysales (salesRepID = 1)
       TO FILE 'toysalespub1.cpy'

   COPY PUBLICATION toysales (salesRepID = 2)
       TO FILE 'toysalespub2.cpy'

The targets you create with these copy files will be able to call work units and perform refreshes exactly like the application described in Chapter 6, “Designing Synchronized Applications”, but without any accompanying code outside the database. If target work unit code stored in the source database is later changed, the changes will automatically be distributed to the targets when they refresh.

See the Cloudscape Tools and Utilities Guide for information on using DBClassLoad to store an application’s entry point in the database.

Locally Configuring Target Database Properties

You can directly set database-wide properties for target databases. Previously, database properties could only be set from the source database by publishing them. Locally setting target database properties enables users to configure individual targets differently. For example, an application could configure each target to support a different set of users. Setting properties from the target database is optional; you can continue to publish properties as well.

NEW: The ability to directly configure target database properties is new in version 3.0.

- “Setting Properties” on page 7-5
- “Checking Property Values” on page 7-6

For a full discussion of properties, see the Tuning Cloudscape Guide.
Setting Properties

Full-access users can set target database properties locally. Setting database-wide properties from the target is handled in the same way as doing so from the source:

- connect to the database
- create a SQL-J statement
- from inside the statement, call the `setDatabaseProperty` method, passing in the name and value of the property.

**NOTE:** The `setDatabaseProperty` method is in the `COM.cloudscape.database.PropertyInfo` class, for which Cloudscape has provided the alias `PropertyInfo`.

The following example demonstrates using ij to:

- create a copy file
- create and connect to a target database
- set a target database property (`p1`) to the value ‘p1val’
- verify its value

```sql
---create the copy file
ij> connect 'sourceDB';
ij> create publication p1 add table mytab;
ij> copy publication p1 to file 'p1.cpy';

---create and connect to the target and set the property
ij> connect 'targDB';createTargetFrom=p1.cpy';
ij> call PropertyInfo.setDatabaseProperty('p1','p1val');
ij> values PROPERTYINFO.getDatabaseProperty('p1')

SQLCol1
------------------------
p1val
```

The next example demonstrates a call which grants full access user authorization to ‘Joe’ on a target database:

```
call
PropertyInfo.setDatabaseProperty('cloudscape.database.fullAccess
sUsers', 'Joe')
```
Checking Property Values

To check the current value of a property at a target database, use the 
getDatabaseProperty method, just as you would at a source or standard database. Connect to the database, then create a SQL-J statement containing the 
getDatabaseProperty method, passing in the name of the property whose value you want to check. To check the current values of all database-wide properties, use the 
getProperties() method.

For example, the following statement retrieves a list of users with full database access:

```java
    call
    PropertyInfo.getDatabaseProperty('cloudscape.database.fullAccessUsers')
```

SQLCol1
----------------------------------
Joe

For more information about PropertyInfo, see the Tuning Cloudscape Guide and the API for COM.cloudscape.database.PropertyInfo.
When a transaction that succeeds at the target fails at the source, information on the error is written to the SYSERRORS system table. However, the SYSERRORS table contains layers of complex objects and is somewhat difficult to scan visually. For this reason, Cloudsync provides a way to analyze failed transaction information from SYSERRORS.

- “The printFailedTransaction Method” on page 8-1
- “Creating a Method Alias” on page 8-2
- “Using printFailedTransaction” on page 8-2
- “Note to Cloudconnector Users” on page 8-2
- “Example Queries” on page 8-3
- “Understanding SYSERRORS Query Results” on page 8-4
- “printFailedTransaction Output” on page 8-4

**NEW:** The ability to analyze failed transaction information from SYSERRORS is new in version 3.0.

### The printFailedTransaction Method

Cloudsync provides the

```java
COM.cloudscape.synchronization.TransactionListVTI.printFailedTransaction
```

method to extract information on failed transactions from SYSERRORS and present it in a more useful form.
The `printFailedTransaction` method takes the `errorID` column of `SYSERRORS` as a parameter, and returns a `ColumnResultSet` wrapping the transaction contents of a `SYSERRORS` row.

The simplest way to use this method is to register it as a method alias and then use it in a `SELECT` statement against `SYSERRORS`.

### Creating a Method Alias

To register the `printFailedTransaction` method as an alias, issue the following SQL-J statement:

```
CREATE METHOD ALIAS printFailedTransaction FOR
TransactionListVTI.printFailedTransaction
```

### Using `printFailedTransaction`

To list all failed transactions in `SYSERRORS`, issue the following statement:

```
SELECT printFailedTransaction(errorID) FROM SYS.SYSERRORS
```

If failed transactions exist, a series of nested result sets are returned. If there are no failed transactions, an empty result set is returned:

```
SQLCol1
--------------
0 rows selected
```

You can optionally specify a second argument to modify the width of the `SUMMARY` columns. If no second argument is specified, `SUMMARY` columns are 25 characters long. To modify the column width, specify a number as the second argument. For example, if you have long object names, you might issue the following query to increase the column width to 40 characters:

```
SELECT printFailedTransaction(errorID, 40) FROM SYS.SYSERRORS
```

### Note to Cloudconnector Users

Because the `printFailedTransaction` method uses nested `ResultSets`, you should append the following attribute to the Cloudconnector database connection URL:

```
weblogic.t3.cacheRows=0
```
This prevents Cloudconnector from pre-fetching the outer ResultSet first. A complete database connection URL might look like the following:

'jdbc:cloudscape:weblogic:sourceDB=weblogic.t3.cacheRows=0'

Since Cloudconnector processes outer nested ResultSets first, failure to specify this corrective attribute causes `printFailedTransaction` to generate the following error:

```java
ERROR 38000: The exception 'java.lang.reflect.InvocationTargetException' was thrown while evaluating an expression.
ERROR XJ001: Java exception: ': java.lang.reflect.InvocationTargetException'.
ERROR (no SQLState): TransactionListVTI not correctly positioned.
```

**Example Queries**

To list in order all failed transactions from publication MY_PUB, issue the following query:

```sql
SELECT e.errorTime, printFailedTransaction(e.errorID)
FROM SYS.SYSERRORS e, SYS.SYSPUBS p
WHERE p.publicationName='MY_PUB'
  AND p.publicationID=e.publicationID
ORDER BY e.errorTime
```

To restrict the previous query to a particular one-hour window, issue a query like the following:

```sql
SELECT e.errorTime, printFailedTransaction(e.errorID)
FROM SYS.SYSERRORS e, SYS.SYSPUBS p
WHERE p.publicationName='MY_PUB'
  AND p.publicationID=e.publicationID
  AND e.errorTime BETWEEN TIMESTAMP'1999-05-01 10:00:00'
  AND TIMESTAMP'1999-05-01 11:00:00'
ORDER BY e.errorTime
```

The following query generates output that includes the time of the transaction failure, the associated error message, and the `TransactionListVTI` output from SYSERRORS. The output shown here is a fragment.

```sql
SELECT e.errorTime, e.errorInfo.getErrorMessage(),
  printFailedTransaction(e.errorID) FROM SYS.SYSERRORS e;
```

<table>
<thead>
<tr>
<th>ERRORTIME</th>
<th>SQLCol1</th>
<th>SQLCol2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-09-29 11:28:48.158</td>
<td>The statement was aborted because it would have caused a duplicate key value in a unique primary key constraint</td>
<td>TransactionListVTI</td>
</tr>
</tbody>
</table>
Debugging Synchronized Systems

Understanding SYSERRORS Query Results

To understand the results returned by queries like those shown above, note the following:

- A refresh message consists of one or more transactions.
- A transaction consists of one or more statement(s).
- Each statement is a sequence of rows.
- A row may contain nested statements.

(Nested statements include those scenarios in which a statement fires a trigger or calls a function which issues other SQL statements.)

The TransactionListVTI class, which contains the printFailedTransaction method, is a Virtual Table Interface that presents a series of rows for each transaction in the refresh message. (See the Cloudscape Developer’s Guide for information on VTIs.)

The statement that failed in each transaction is flagged with the following box:

```
********************************
*       FAILED STATEMENT       *
********************************
```

printFailedTransaction Output

The column list displays at the top of the output stream.

```
SQLCol1
--------
TransactionListVTI
```

Then, a set of rows is displayed for each failed transaction.

For each failed transaction:

The first row shows the contents of TransactionListVTI.

```
Contents of SQLCol1:
TRAN_NUMBER | ORIGINATING_DBID | ORIGINATING_DB | SUMMARY | STATEMENT_LIST
-----------------------------------------------
1           | a816c00e-00da-ce22-a326-00da | (2, 42744) | TRANSACTION | StatementListVTI
```

This row displays information about the transaction including the transaction number, the ID of the originating database, and a StatementListVTI, which is a VTI containing all statements in the transaction.
The next row shows the values for the StatementListVTI.

Contents of STATEMENT_LIST:

<table>
<thead>
<tr>
<th>STATEMENT_NUMB</th>
<th>TYPE</th>
<th>SUMMARY</th>
<th>SUC</th>
<th>CHANGED_COLUMNS</th>
<th>ROW_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>FAILED STATEMENT</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This row displays once for each statement in the transaction, and includes information about the statement, such as the statement number, statement type, and RowListVTI, which is another VTI containing information about each statement in the transaction.

The next row shows the values for RowListVTI.

Contents of ROW_LIST:

<table>
<thead>
<tr>
<th>RLV_ROW_NUMBER</th>
<th>RLV_TYPE</th>
<th>RLV_STMT_LIST</th>
<th>ARG_1</th>
<th>ARG_2</th>
<th>ARG_3</th>
<th>ARG_4</th>
<th>ARG_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ARGUMENTS</td>
<td>NULL</td>
<td>018f00</td>
<td>1</td>
<td>ann</td>
<td>barbie</td>
<td>11</td>
</tr>
</tbody>
</table>

This row displays once for each row in the StatementListVTI.

The contents of RowListVTI depend on the statement type. In the output above, the statement type is a work unit, and RowListVTI contains information about the work unit arguments.

RowListVTI has a column called RLV_STMT_LIST that holds a nested StatementListVTI for any nested statement. If no nested statements exist, RLV_STMT_LIST is null. If a nested statement exists, its StatementListVTI and RowListVTI are also displayed.

More detail on TransactionListVTI, StatementListVTI, and RowListVTI are available in the javadoc for these classes.

The next example uses this query:

```sql
SELECT e.errorinfo.getErrorMessage(),
       printFailedTransaction(e.errorID)
FROM SYS.SYSERRORS e
```

to show sample output for a refresh containing three failed transactions:

- a failed insert containing a nested statement
- a failed simple update
- a multi-statement transaction consisting of an insert containing a nested statement and a failed work unit call.
The statement was aborted because it would have caused a duplicate key value in a |TransactionList

Contents of SQLCol2:

<table>
<thead>
<tr>
<th>TRAN_NUMBER</th>
<th>ORIGINATING_DBID</th>
<th>ORIGINATING_CODEC</th>
<th>SUMMARY</th>
<th>STATEMENT_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>71ce41e2-00d8-4af7-831a-04e1</td>
<td>(1,47271)</td>
<td>TRANSACTION</td>
<td>StatementListV4</td>
</tr>
</tbody>
</table>

Contents of STATEMENT_LIST:

<table>
<thead>
<tr>
<th>STATEMENT_NUMB</th>
<th>TYPE</th>
<th>SUMMARY</th>
<th>SUC</th>
<th>CHANGED_COLUMNS</th>
<th>ROW_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE</td>
<td>UPDATE APP.SELLERS</td>
<td>S</td>
<td>(3)</td>
<td>RowListVTI</td>
</tr>
</tbody>
</table>

Contents of ROW_LIST:

<table>
<thead>
<tr>
<th>RLV_ROW_NUMBER</th>
<th>RLV_TYPE</th>
<th>RLV_STMT_LIST</th>
<th>&quot;RLV_OLD_&quot;</th>
<th>&quot;SELLERID&quot;</th>
<th>&quot;MAXORDER&quot;</th>
<th>&quot;MAXBUYER&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
<td>Spuds</td>
<td>101</td>
<td>Acme</td>
<td>Paradise</td>
<td></td>
</tr>
</tbody>
</table>

1 row selected

2 rows selected

1 row selected

The statement was aborted because it would have caused a duplicate key value in a |TransactionList

Contents of SQLCol2:

<table>
<thead>
<tr>
<th>TRAN_NUMBER</th>
<th>ORIGINATING_DBID</th>
<th>ORIGINATING_CODEC</th>
<th>SUMMARY</th>
<th>STATEMENT_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>71ce41e2-00d8-4af7-831a-04e1</td>
<td>(1,47271)</td>
<td>TRANSACTION</td>
<td>StatementListV4</td>
</tr>
</tbody>
</table>

Contents of STATEMENT_LIST:

<table>
<thead>
<tr>
<th>STATEMENT_NUMB</th>
<th>TYPE</th>
<th>SUMMARY</th>
<th>SUC</th>
<th>CHANGED_COLUMNS</th>
<th>ROW_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cloudscape Synchronization Guide
### Understanding SYSERORS Query Results

<table>
<thead>
<tr>
<th>Row</th>
<th>STATUS</th>
<th>TABLE</th>
<th>SCHEMA</th>
<th>FIELD.DOMAIN</th>
<th>VALUE</th>
<th>ROW_LIST</th>
<th>CONTENTS OF ROW_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE</td>
<td>UPDATE APP.BUYERS</td>
<td>F</td>
<td>{3, 4}</td>
<td>RowListVTI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Contents of ROW_LIST:**

```
RLV_ROW_N|RLV_TYP|RLV_STMT_LIST|"SELLER"|"BUYER"|NAME|ADDRESS
1      |UPDATE ROW|NULL|Spuds|Angstwear|NULL|NULL|Acme|Paradise
```

1 row selected

**The exception 'COM.cloudscape.synchronization.SkipTransactionSQLException: TransactionList exception.'**

**Contents of SQLCol2:**

```
TRAN_NUMBER | ORIGINATING_DBID | ORIGINATING_CO | SUMMARY | STATEMENT_LIST
1            | 71ce41d2-00d8-daf7-831e-00|0|TRANSACTION|StatementListV4
```

**Contents of STATEMENT_LIST:**

```
STATEMENT_NUMBER|TYPE |SUMMARY |SUC|CHANGED_COLUMNS|ROW_LIST
1             |INSERT |INSERT APP.BUYERS |S|NULL|RowListVTI
```

**Contents of ROW_LIST:**

```
RLV_ROW_N|RLV_TYP|RLV_STMT_LIST|"SELLER"|"BUYER"|NAME|ADDRESS
1      |STATEMENT LIST|StatementListV4|NULL|NULL|NULL|NULL
```

**Contents of RLV_STMT_LIST:**

```
STATEMENT_NUMBER|TYPE |SUMMARY |SUC|CHANGED_COLUMNS|ROW_LIST
1              |UPDATE |UPDATE APP.SELLERS |S |{3} |RowListVTI
```

**Contents of ROW_LIST:**

```
RLV_ROW_N|RLV_TYP|RLV_STMT_LIST|"BUYER"|"NAME"|"ADDRESS"
1      |STATEMENT LIST|StatementListV4|NULL|NULL|NULL|NULL
```

**Contents of RLV_STMT_LIST:**

```
STATEMENT_NUMBER|TYPE |SUMMARY |SUC|CHANGED_COLUMNS|ROW_LIST
1              |UPDATE |UPDATE APP.BUYERS |S |{3} |RowListVTI
```

**Contents of ROW_LIST:**

```
RLV_ROW_N|RLV_TYP|RLV_STMT_LIST|"SELLER"|"BUYER"|NAME|ADDRESS
1      |STATEMENT LIST|StatementListV4|NULL|NULL|NULL|NULL
```

**Contents of STATEMENT_LIST:**

```
STATEMENT_NUMBER|TYPE |SUMMARY |SUC|CHANGED_COLUMNS|ROW_LIST
1              |UPDATE |UPDATE APP.SELLERS |S |{3} |RowListVTI
```

2 rows selected

********************************
*                         *
*       FAILED STATEMENT    *
*                         *
******************************

**Contents of ROW_LIST:**

```
WORK UNIT | CALL NEWORDER | F |NULL |RowListVTI
```

---

*Cloudscape Synchronization Version 3.0*
### Debugging Synchronized Systems

<table>
<thead>
<tr>
<th>RLV_ROW_NUMBER</th>
<th>RLV_TYPE</th>
<th>RLV_STMT_LIST</th>
<th>ARG_1</th>
<th>ARG_2</th>
<th>ARG_3</th>
<th>ARG_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ARGUMENTS</td>
<td>NULL</td>
<td>Spuds</td>
<td>201</td>
<td>100</td>
<td>Tractor</td>
</tr>
</tbody>
</table>

1 row selected

2 rows selected

1 row selected

3 rows selected
PART TWO

Cloudsync
Reference
This chapter describes the SQL-J commands used only with Cloudsync. For information about other SQL-J commands, and about many of the elements within these commands, see the Cloudscape Reference Manual.

The Cloudscape synchronization commands are:

- “ALTER PUBLICATION” on page 9-4
- “COPY PUBLICATION” on page 9-6
- “CREATE PUBLICATION” on page 9-8
- “CREATE WORK ALIAS” on page 9-13
- “DROP PUBLICATION” on page 9-14
- “DROP WORK ALIAS” on page 9-15
- “REFRESH” on page 9-16
ALTER PUBLICATION

Changes an existing publication. Run at the source database. This statement can only be executed on a source database. The effects of ALTER PUBLICATION are replayed at a subscribing target at its next refresh.

Syntax

```
ALTER PUBLICATION PublicationName
  [ ADD PARAMETER ParameterSpec ]*
  [ ADD TABLE TableName
    [ { SimpleColumnName [, SimpleColumnName ]* } ]
    [ WHERE restriction ] ]*
  [ ADD INDEX IndexName ]*
  [ ADD VIEW ViewName ]*
  [ ADD STATEMENT StatementName ]*
  [ ADD METHOD ALIAS MethodAliasName ]*
  [ ADD WORK ALIAS WorkAliasName ]*
  [ ADD CLASS ALIAS ClassAliasName ]*
  [ ADD FOREIGN KEY ForeignKeyName ]*
  [ ADD AGGREGATE AggregateName ]*
  [ ADD TRIGGER TriggerName ]*
  [ ADD TARGET DATABASE PROPERTY PropertyName='propertyValue' ]*
  [ ADD JAR FILE JarFileName ]*
  [ DROP TABLE TableName ]*
  [ DROP INDEX IndexName ]*
  [ DROP VIEW ViewName ]*
  [ DROP STATEMENT StatementName ]*
  [ DROP METHOD ALIAS MethodAliasName ]*
  [ DROP WORK ALIAS WorkAliasName ]*
  [ DROP CLASS ALIAS ClassAliasName ]*
  [ DROP FOREIGN KEY ForeignKeyName ]*
  [ DROP AGGREGATE AggregateName ]*
  [ DROP TRIGGER TriggerName ]*
  [ DROP TARGET DATABASE PROPERTY PropertyName ]*
  [ DROP JAR FILE JarFileName ]*
```

ParameterSpec

```
ParameterName DataType [ DEFAULT NULL ]
```

NEW: Triggers, Class Aliases and User Aggregates are new in version 3.0.

The ADD and DROP statements can appear in any order. DROPs apply before ADDs, so you can DROP and ADD the same dictionary object in either order.

The limitations on the WHERE restriction are the same as for the CREATE PUBLICATION statement. See “CREATE PUBLICATION” on page 9-8 for more information.
During the time a transaction containing ALTER PUBLICATION is uncommitted, no other transaction may:

- alter, copy, or drop the publication
- alter a table in the publication

ALTER PUBLICATION cannot make a publication inconsistent or incomplete. To ensure this:

- If ALTER PUBLICATION adds a view, index, or stored prepared statement, then the same ALTER PUBLICATION statement must also add all tables that the view, index, or stored prepared statement mentions, unless they are already in the publication.
- If ALTER PUBLICATION drops a table, it must also drop all views, indexes, and stored prepared statements that mention the table.
- You cannot add an object that is already in the publication, though you can drop an object and then add it again in the same ALTER PUBLICATION statement.

Like with any published object, before you can drop a published jar file, you must first drop all publications in which it appears. You cannot drop the file using `COM.cloudscape.tools.dbclasses.removeJar`. Instead, use `COM.cloudscape.tools.dbclasses.replaceJar` to replace the source jar file, and subsequent refreshes will update the target jar file.

A refresh that changes a target’s `cloudscape.database.classpath` produces a JDBC warning.

See “CREATE PUBLICATION” on page 9-8 for information on target properties you can include in a publication.

**Example**

```sql
-- issued at source
ALTER PUBLICATION toySales
DROP TABLE salesReps
ADD TABLE salesReps
   WHERE salesRepID=?salesRepID
ADD JAR FILE APP.TargetWORK
ADD TARGET DATABASE PROPERTY
   cloudscape.database.classpath='APP.TargetWork'
```

When you alter a publication to add a table or index, its properties are automatically set upon creation.
**COPY PUBLICATION**

Creates a file from which to create a target database. This statement can only be executed on a source database.

**Syntax**

```sql
COPY PUBLICATION PublicationName
[ ( ParameterName = value [, ParameterName = value ]* ) ]
TO FILE 'FileName'
```

*PublicationName* and *ParameterName* are SQL-J-style identifiers. (See the Cloudscape Reference Manual for more information on SQL-J identifiers.) *PublicationName* is schema-qualifiable and *ParameterName* is not schema-qualifiable.

You must have operating system write permission on the destination file. *FileName* is a string delimited by single quotes ('). *FileName* can use either a relative path (to the database’s JVM’s current working directory) or an absolute path, and uses operating system, not URL, syntax. If the specified file already exists, it is overwritten.

When copying a publication to a file, you can specify values for the parameters. Parameters not stated in the COPY PUBLICATION statement assume their default values. Parameter values are plugged into the WHERE clauses (defined at CREATE PUBLICATION time) to pick out the desired rows.

When a CREATE PUBLICATION statement is prepared, COPY PUBLICATION’s *ParameterName* values and *FileName* can be “?” parameters that will be specified at run-time.

COPY PUBLICATION always occurs at isolation level 3 (SERIALIZABLE), even if the current transaction is at another isolation level.

To see the parameter values in an existing target:

1. Get the publication ID by calling `COM.cloudscape.database.PropertyInfo.getDatabaseProperty`.
2. Look at the row in SYSCOLUMNS whose key is that publication ID.

   For example, if you have created and published a method alias called `getDatabaseProperty` that refers to `COM.cloudscape.database.PropertyInfo.getDatabaseProperty`, the following query at the target returns the target’s parameter information:

   ```sql
   SELECT * FROM SYS.SYSCOLUMNS
   WHERE referenceid=getDatabaseProperty('sysinfo.publicationID')
   ```

*Cloudscape Synchronization Guide*
See “CREATE PUBLICATION” on page 9-8 for details about publication contents.

**Example**

-- issued at source
COPY PUBLICATION toysales (salesRepID = 2)
  TO FILE 'toysalespub2.cpy'

-- in a prepared statement
COPY PUBLICATION toysales( salesRepID = ?)
  TO FILE ?
CREATE PUBLICATION

Creates a publication based on a source database. This statement can only be executed at the source database. A publication defines which objects in the source database will be synchronized with a certain set of targets.

NEW: The ability to add, class aliases, aggregates, and triggers is new in version 3.0. The functionality of the SET STATEMENT and TARGET DATABASE PROPERTY clauses has also changed.

Syntax

```
CREATE PUBLICATION PublicationName
  [ ( ParameterSpec [ , ParameterSpec ]* ) ]
  [ ADD TABLE TableName
    [ ( SimpleColumnName [ , SimpleColumnName ]* ) ]
    [ WHERE restriction ] ]
  [ ADD INDEX IndexName ]*
  [ ADD VIEW ViewName ]*
  [ ADD STATEMENT StatementName ]*
  [ ADD METHOD ALIAS MethodAliasName ]*
  [ ADD CLASS ALIAS ClassAliasName ]*
  [ ADD FOREIGN KEY ForeignKeyName ]*
  [ ADD AGGREGATE AggregateName ]*
  [ ADD TRIGGER TriggerName ]*
  [ ADD JAR FILE JarFileName ]*
  [ ADD TARGET DATABASE PROPERTY PropertyName='propertyValue' ]*
```

ParameterSpec

```
ParameterName DataType [ DEFAULT NULL ]
```

NOTE: The ADD clauses can be in any order.

*SimpleColumn* is the column’s name, unqualified by a table or correlation name, as within a CREATE TABLE statement. *SimpleColumn* must identify a column in the table. *ParameterName* is an SQL-92-style identifier. (See the Cloudscape Reference Manual for more information on SQL-J identifiers.)

Object names (*PublicationName, TableName, IndexName, ViewName, StatementName, ForeignKeyName, JarFileName, and so on) are SQL-J identifiers that may be qualified with a schema name. While objects in the SYS schema cannot appear in a publication, views can reference SYS schema objects. If the schema is not specified, the schema that holds the publication is assumed.

Syntax of *DataType, TableName, SimpleColumn*, IndexName, ViewName, StatementName, ForeignKeyName, TriggerName, AggregateName,
ClassAliasName, and JarFileName are described in the Cloudscape Reference Manual.

NULL is the only currently supported DEFAULT value for parameters.

If no column list is provided for a published table, all of the table’s columns are published, including those added in later ALTER TABLE ADD COLUMN commands.

The WHERE Clause

Each ADD TABLE declaration can include an optional WHERE clause. The WHERE clause may refer to the publication parameters and published columns of that table only. Parameter references are prefixed with a ‘?’.

You must explicitly publish all work units and method aliases (as well as other objects) used by targets made from the publication. You must also make available to the target any Java classes used in the WHERE clause.

The WHERE clause of a CREATE PUBLICATION statement becomes a check constraint on a target created from the publication. Therefore, a target transaction that would violate the publication’s WHERE clause will fail at the target.

A publication’s WHERE clause should evaluate to the same boolean value each time it is run on an unchanged row. To ensure this, a WHERE clause cannot contain:

- subqueries
- aggregates
- any of the following operators:
  - CURRENT_DATE
  - CURRENT_TIME
  - CURRENT_TIMESTAMP
  - RUNTIMESTATISTICS
  - USER
  - CURRENT_USER
  - SESSION_USER

A WHERE clause can contain any other SQL-J operators, including:

- EXTRACT
- CHAR_LENGTH
- CHAR_LENGTH (or CHARACTER_LENGTH)
- OCTET_LENGTH
Be careful when including object methods or fields in a CREATE PUBLICATION statement’s WHERE clause. Make sure that the publication will always create the same target file if the published data is unchanged. To ensure this, be careful when using SQL-J statements in the method, since a query may return different results at different times.

**Notes**

A published table must have a primary key. The primary key must be contained in the table’s published columns. This makes it possible to keep source and target data synchronized, since row identity comes from the primary key.

Tables must be explicitly published using the ADD TABLE clause. Once a table (or any other dictionary item) is explicitly published, you cannot drop it from the source until you drop all publications in which it occurs.

When you publish a table, you also automatically publish:

- the table’s primary key
- the table’s unique and check constraints, except those that refer to unpublished columns
- the indexes that support the table’s primary key or unique constraints
- the column defaults
- the table’s conglomerate-level properties

Published statements keep their NO COMPILE status.

Indexes other than those supporting the table’s primary key or unique constraints must be published explicitly. To do this, use the ADD_INDEX clause. The table and all columns in the index must also be published.
Triggers must be published explicitly. Even if a trigger is published, changes made by the target trigger are not sent to the source at refresh time, unless these changes were made in a different connection (which is strongly discouraged). Cloudsync relies on the source trigger to ensure that the correct data is copied to the target at the end of the refresh operation. For more information about trigger behavior during refresh, see the section “Triggers and the Refresh Operation” on page 4-7.

Foreign keys must also be explicitly published.

To publish a view, you must also publish all tables, columns, method aliases, and other views that it depends on. There is no restriction on the operators used in a view. Views can reference built-in class aliases, since all databases, including targets, contain them.

When you publish a method alias, work unit, user-aggregate, or class alias, only the alias names are copied to the target; the classes themselves are not copied. Also, when you publish a table or view that uses a Java class, the class itself is not automatically copied. To synchronize such classes, put them into the database in jar files, and publish the jar files. See “Distributing Application Code Within the Database” on page 7-2 for instructions.

Because DDL is not allowed on a target database, a publication which contains a stored prepared DDL statement will fail at run-time. All dictionary objects (e.g., tables, views) referenced by stored prepared DML statements in the publication must be published.

Changes to published static properties do not take effect until the target reboots.

JarFileName is a qualified name, that is, an optional schema name, followed by a dot, followed by an object name (a SQL-92 identifier). PropertyName is a dot-separated name (a Java identifier). PropertyValue is a string.

Please refer to Tuning Cloudscape for more information about publishable properties.

If ADD TARGET DATABASE PROPERTY publishes a database class path, CREATE PUBLICATION fails unless the class path has two-part names and mentions only published jar files.

Example

CREATE PUBLICATION toySales (salesRepID INT)
ADD TABLE toys
ADD TABLE orders
    WHERE salesRepID = ?salesRepID
ADD TABLE leads
WHERE sentToRepID = ?salesRepID
ADD METHOD ALIAS getKey
ADD WORK ALIAS placeOrder
ADD WORK ALIAS sendLead
ADD JAR FILE APP.targetWORK
ADD TARGET DATABASE PROPERTY
  cloudscape.database.classpath='APP.TargetWork'
CREATE WORK ALIAS

Declares two static Java methods as a work unit. This statement can only be executed on a source database.

Syntax

```
CREATE WORK ALIAS WorkAliasName
FOR SourceMethodName [, TargetMethodName ]
```

Work units are modeled on method aliases. Like method aliases, they are never qualified with a schema name.

If the target method is not found, an error is raised when the work unit is invoked, and if the source method is not found, an error is raised at refresh time, and the refresh halts.

If the source method and target method do not have the same number and type of parameters, CREATE WORK ALIAS fails at run time.

If you don’t specify the `TargetMethodName`, it is assumed to be same as the `SourceMethodName`.

The name of a work unit cannot duplicate the name of a method alias or user-defined aggregate.

See the Cloudscape Reference Manual for information on the related CREATE METHOD ALIAS command.

Example

```
CREATE WORK ALIAS placeOrder
  FOR SourceWork.makeSaleAtSource,
       TargetWork.makeSaleAtTarget
```
DROP PUBLICATION

Removes a publication from a source database. This statement can only be executed on a source database.

**Syntax**

```
DROP PUBLICATION PublicationName
```

Once DROP PUBLICATION is run, the publication is removed, and you cannot make copy files from it, nor can you refresh targets made from it. You can, however, still create target databases from any copy file created from the publication even after the publication is dropped. You can change data in such databases (if they are configured to allow data changes), but any refreshes they issue will fail.

The items specified in the publication are not affected by the DROP PUBLICATION statement, except that they may now be droppable themselves.

**Example**

```
-- publication is no longer needed, so drop it
DROP PUBLICATION toySales
```
DROP WORK ALIAS

Drops a work unit from the database. This statement can only be executed at a source database.

Syntax

```
DROP WORK ALIAS WorkAliasName
```

You cannot issue DROP WORK ALIAS on a published alias. You must first drop or alter the publication.

Example

```
DROP WORK ALIAS sendLead
```
REFRESH

Performs a refresh operation. This statement can only be executed on a target database.

Syntax

```
REFRESH
```

The refresh operation:

1. sends provisional target transactions to the source
2. copies source changes (since this target’s last refresh) back to the target

The connection waits until the refresh has completed.

You cannot run REFRESH if there is already uncommitted work in the transaction (this is not an issue if autocommit is on). REFRESH implicitly commits when it completes successfully. You cannot roll back a successful refresh operation.

REFRESH fails with an error in any of the following cases:

- It is issued at a database that is not a target.
- The source database has been dropped.
- The source database cannot be found at its designated address and port.
- A network communication error occurs.
- The publication has been dropped.
- It contains a work unit that has been dropped from the publication.
- The last refresh was more than one week before the current one (this interval is configurable with the `cloudscape.refresh.guaranteedRefreshInterval` property).
- It appears in a transaction that has modified the database. (To be safe, either commit immediately before issuing REFRESH or use auto-commit.)
- It occurs within Java code called by an SQL statement.

If the target has been cloned, or the source has been copied or restored from a backup, the following cases also cause REFRESH to fail with an error:

- The source did not create the target’s copy file.
- The source did not perform this target’s previous refresh.
- The source has performed a refresh on a clone of this target.

NEW: Detecting obsolete work units and divergent target clones on REFRESH is new in version 3.0.
The Cloudscape Reference Manual provides information about the system tables that are common to all Cloudscape databases. Some of these tables contain information relevant to Cloudscape synchronization.

This chapter describes the additional system tables used only by Cloudscape synchronization. These tables appear only in databases created as source or target databases. Like all system tables, they are in the SYS schema.

You can query system tables, but you cannot alter them directly. Use the JDBC DatabaseMetaData feature to get more information about these tables.

The system tables for Cloudsync are:

- “SYSPUBITEMS” on page 10-2
- “SYSPUBS” on page 10-3
- “SYSSYNCINSTANTS” on page 10-4
**SYSPUBITEMS**

Lists published dictionary objects. It contains one row per object per publication.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLICATIONID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Unique identifier for the publication (joins with SYSPUBS).</td>
</tr>
<tr>
<td>ITEMID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Object ID of an item included explicitly or implicitly in the publication.</td>
</tr>
<tr>
<td>ITEMTYPE</td>
<td>INT</td>
<td></td>
<td>false</td>
<td>Indicates whether this row corresponds to a table, view, trigger, etc. Determines which system table itemID joins to. The possible values of ITEMTYPE are states in the COM.cloudscape.database.Database class.</td>
</tr>
<tr>
<td>EXPLICIT</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>Indicates whether the item was included explicitly or implicitly.</td>
</tr>
<tr>
<td>DETAILS</td>
<td>java.io.Externalizable</td>
<td></td>
<td>true</td>
<td>Contains information on the item.</td>
</tr>
</tbody>
</table>

**Indexes**

Non-unique index SYSPUBITEMS_INDEX1 on (PUBLICATIONID, ITEMTYPE)

Unique index SYSPUBITEMS_INDEX2 on (PUBLICATIONID, ITEMID)

Unique index SYSPUBITEMS_INDEX3 on (ITEMID, PUBLICATIONID)
SYSPUBS

Lists the publications in the database. It contains one row per publication. It appears only at the source database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLICATIONID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Unique identifier for the publication</td>
</tr>
<tr>
<td>PUBLICATIONNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>Name of the publication</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Unique identifier for the schema containing the publication (joins with SYSSCHEMAS)</td>
</tr>
</tbody>
</table>

**Indexes**

Unique index SYSPUBS_INDEX1 on (PUBLICATIONID)

Unique index SYSPUBS_INDEX2 on (PUBLICATIONNAME, SCHEMAID)
**SYSSYNCINSTANTS**

Saves the instant of the last interesting event processed from another database. In the source, this table contains one row for each target that has performed a refresh. In a target, this table contains exactly one row, which represents the source.

Cloudscape synchronization uses this table to ensure that events (such as transactions) apply exactly once, even if a refresh fails to complete successfully.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>ID of the database being synchronized with</td>
</tr>
<tr>
<td>INSTANT</td>
<td>COM.cloudscape.types.DatabaseInstant</td>
<td>false</td>
<td>An “instant” in a database</td>
<td></td>
</tr>
<tr>
<td>TRUNCGUARD</td>
<td>COM.cloudscape.types.DatabaseInstant</td>
<td>false</td>
<td>A past instant used for deleting unneeded SysSyncInstant rows</td>
<td></td>
</tr>
<tr>
<td>SYNCID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>ID of the event to which the instant refers</td>
</tr>
</tbody>
</table>

**NEW:** These SYSSYNCINSTANTS columns are new in version 3.0.

**Indexes**

Unique index `SYSSYNCINSTANTS_INDEX1` on `(DBID)`
To create or modify a database, use a database connection URL containing database creation attributes. See the *Cloudscape Developer’s Guide* for information about using database connection URLs.

There are three Cloudsync-specific database connection URL attributes:

- `convertToSource=true`
- `createSource=true`
- `createTargetFrom=copyFileName`

Attribute names are case-sensitive.
**convertToSource** = true

**Function**

The *convertToSource* attribute converts a non-synchronization-enabled database to a source database. The *convertToSource* attribute has two possible values: *true* and *false*. The default is *false*.

**NOTE:** You cannot convert a target database to a source database.

Set the *convertToSource* attribute only once, at the time you convert the non-synchronization-enabled database to a source database.

Using *convertToSource* in a database connection URL raises an SQL exception if any of the following is true:

- The database was created using a previous version of Cloudscape.
- The database connection URL also upgrades the database to the current version of Cloudscape.
- The database is a target.
- The database is already booted.
- Any of the following attributes are also specified on the database connection URL:
  - *create*
  - *createSource*
  - *createTargetFrom*

**Example**

The following database connection URL converts a non-synchronization-enabled database to a source database on a local Cloudconnector server:

```
jdbc:cloudscape:weblogic:MyNonSyncDB;convertToSource=true
```

**NEW:** The *convertToSource* attribute is new in Version 3.0.
**createSource=true**

**Function**

The `createSource` attribute specifies to create a synchronization-enabled source database, and is used in place of `create=true`. It has two possible values: `true` and `false`. The default is `false`.

Set the `createSource` attribute only once, at the time you create the source database. If you specify `createSource` for a database that already exists, it has no effect, and no warning is issued.

**Example**

The following database connection URL creates and connects to a source database on a Cloudconnector server:

```
jdbc:cloudscape:weblogic:MySourceDB;createSource=true
```

**NOTE:** To convert an existing non-synchronization-enabled database to a source database, use the `convertToSource=true` attribute in the database connection URL.
**URL Attributes for Synchronization**

**createTargetFrom=copyFileName**

**Function**

Use the `createTargetFrom` attribute when creating a target database. `copyFileName` specifies the name of the copy file from which to create the target database. Set the value of the `createTargetFrom` attribute to the path or URL of the copy file (created using the `COPY PUBLICATION` command). This attribute is required when you create a target database.

You should set this attribute only once, at the time you create the target database. If you specify `createTargetFrom` for a database that already exists, it has no effect, and no warning is issued.

If you want, you can move the copy file onto the machine where the target database will reside before creating the target database.

Note that no quotes are used around the name of the copy file.

**Example**

The following database connection URL creates a target database named `MyTargetDB` from the `copyPub1.out` file:

```
jdbc:cloudscape:MyTargetDB;createTargetFrom=copyPub1.out
```

You can use an URL as well as a path name to specify the copy file. For example:

```
jdbc:cloudscape:myTargetDB;createTargetFrom=http://adomain.com/createcopyfiles?var1=35
```

Since Cloudscape uses semicolons to delimit attributes, if the URL contains a semicolon(;) do one of the following:

- Put the `createTargetFrom` attribute in the `Properties` set passed to the `DriverManager.getConnection` call.
- Use the standard URL encoding (%3b) for semicolons.
12 Cloudsync Properties

For information about using properties in non-synchronized Cloudscape systems, see the *Tuning Cloudscape Guide.*

This chapter discusses both system-wide and database-wide properties. System-wide properties apply to an entire system, including databases and conglomerates. Database-wide properties apply only to the database associated with the property. For more information, see *Tuning Cloudscape* and the *Cloudscape Reference Manual.*

You can set most database-level properties at the target by publishing them or by setting them directly at the target database using the `PROPERTYINFO.setDatabaseProperty` method. Only a full-access user can set a property at a target database. Setting a property at a target overrides the property’s published value, if any. Deleting a property value at a target (that is, setting it to null) resets the property to its published value, if any.

NEW: The ability to directly set target database properties is new in version 3.0.

This chapter describes properties used only with Cloudscape synchronization. They are:

- “cloudscape.database.readAccessTarget” on page 12-3
- “cloudscape.database.refresh” on page 12-5
- “cloudscape.database.sourceUser” on page 12-6
- “cloudscape.database.synchronizationCleanup” on page 12-8
- “cloudscape.DataComm.listenType” on page 12-9
- “cloudscape.DataComm.port” on page 12-10
- “cloudscape.DataComm.url” on page 12-11
- “cloudscape.listener.synchronization.address” on page 12-14
- “cloudscape.refresh.guaranteedRefreshInterval” on page 12-15
Cloudsync Properties

- “cloudscape.synchronization.workUnitOnly” on page 12-16
- “sysinfo.publicationID” on page 12-18
- “sysinfo.publicationName” on page 12-19
- “sysinfo.sourceDatabaseID” on page 12-20
- “sysinfo.sourceDatabaseName” on page 12-21
- “sysinfo.sourceServerAddress” on page 12-22
cloudscape.database.readAccessTarget

Function

The `cloudscape.database.readAccessTarget` property specifies whether target writes (that is, INSERTs, UPDATEs, and DELETEs) are prevented. It has two possible values: `true` and `false`. The default is `false`.

If `cloudscape.database.readAccessTarget` is set to `true`, the target database can still be refreshed from the source, but it cannot be altered by target-side statements.

The only way to set the `cloudscape.database.readAccessTarget` property is to publish it using the ADD TARGET DATABASE PROPERTY of CREATE PUBLICATION. Once published, this property cannot be altered.

`cloudscape.database.readAccessTarget` can only be set for a target database.

A read-access target is useful when the target displays data but is not allowed to update anything.

**NOTE:** The `cloudscape.database.readAccessTarget` property optimizes the refresh operation. You could also disable target updates by configuring the target as a read-only database, but that would not optimize the refresh operation.

If `cloudscape.database.readAccessTarget` is set to `true`, `Database.isReadOnly` returns `true`, even though the database can be updated by a refresh. This is so applications that use `Database.isReadOnly` know they cannot issue INSERTs, UPDATEs, and DELETEs.

**Syntax**

```
CREATE PUBLICATION pubname
...
ADD TARGET DATABASE PROPERTY
  cloudscape.database.readAccessTarget=value
```

**Default**

`false`

**Example**

```
CREATE PUBLICATION mypub
ADD TABLE testtable
ADD TARGET DATABASE PROPERTY
  cloudscape.database.readAccessTarget=true
```
Cloudsync Properties

Scope
Database-wide
cloudscape.database.refresh

**Function**

The `cloudscape.database.refresh` property determines whether a source database accepts refresh requests. This property is meaningful only at a source database. Possible values of this property are ON and OFF. These strings are case-insensitive.

**Syntax**

```
CALL PropertyInfo.setDatabaseProperty('cloudscape.database.refresh', value)
```

**Default**

ON

**Example**

```
CALL PropertyInfo.setDatabaseProperty('cloudscape.database.refresh', 'OFF')
```

**Scope**

Database-wide

*Cloudscape Synchronization Version 3.0*
Cloudsync Properties

cloudscape.database.sourceUser

Function

The `cloudscape.database.sourceUser` property specifies the principal name the target will use to identify itself to the source during a refresh. The name can be either delimited or non-delimited. If this property is not supplied, the target uses the current user name.

This property is meaningful only at a target database. It is ignored if set for other databases.

When a refresh applies target transactions to the source, the SQL-J CURRENT_USER and SESSION_USER expressions evaluate to the connected source user.

Syntax

```
CREATE PUBLICATION pubname
...
ADD TARGET DATABASE PROPERTY
cloudscape.database.sourceUser=value
```

or:

```
CALL PROPERTYINFO.setDatabaseProperty(
    propertyName, value)
```

Default

The current user name.

Examples

At the source:

```
CREATE PUBLICATION myPub
...
ADD TARGET DATABASE PROPERTY
cloudscape.database.sourceUser='George'
```

By a full-access user at the target:

```
CALL PROPERTYINFO.setDatabaseProperty(
    'cloudscape.database.sourceUser','George')
```
Scope
Database-wide
**cloudscape.database.synchronizationCleanup**

**Function**

The `cloudscape.database.synchronizationCleanup` property determines whether synchronization-related “housecleaning” occurs automatically at a source database. It has meaning only at a source database. It should be set to `false` only to enhance performance during peak load times.

**Syntax**

```
cloudscape.database.synchronizationCleanup=value
```

**Default**

`true`

**Example**

```
cloudscape.database.synchronizationCleanup=false
```

**Scope**

Database-wide
cloudscape.DataComm.listenType

Function
The `cloudscape.DataComm.listenType` property specifies whether you will be using the servlet API or built-in HTTP support to manage refresh messages. The two possible values are `servlet` and `builtinHttp`. If you set it to `servlet`, you must register the servlet at the source server and you also need to set the `cloudscape.listener.synchronization.address` property to point to it. Of course, the source server must have servlet capabilities.

This property is only meaningful on a source database. It is ignored at other databases. You must reboot for this property to take effect.

Syntax
```
cloudscape.DataComm.listenType=refreshService
```

Default
`builtinHttp`

Example
```
cloudscape.DataComm.listenType=servlet
```

Scope
System-wide
**cloudscape.DataComm.port**

**Function**

The `cloudscape.DataComm.port` property specifies the port number on which the source databases of the given Cloudscape system expect to receive refresh messages. All the databases in the system will use this port. Set this property in the source database system before running COPY PUBLICATION. The host name of the source’s system is automatically set in the target databases when they are created. This property is only meaningful on a system for which `cloudscape.DataComm.listenType` is set to `builtinHttp`.

Be sure to specify a new port number if there is any chance that another source database will run concurrently in another Cloudscape system on the same host, since by default it will try to use the same port.

**Syntax**

```
cloudscape.DataComm.port=portNumber
```

**Default**

The default port number is 2001.

**Example**

```
cloudscape.DataComm.port=2002
```

**Scope**

System-wide
**Function**

Enables the standard JDK URL support for POST messages and thus makes SSL use possible. Cloudscape targets use HTTP POST messages to communicate with the Cloudsync source database for the refresh message and reply.

Version 3.0 provides a choice of implementations for the processing of HTTP POST requests for the refresh command.

- The default implementation \( \text{cloudscape.DataComm.url=false} \) is the one that was part of all previous releases.
  The implementation uses Java sockets rather than the standard URL support provided by the JDK. No SSL support is not possible.
- The optional implementation \( \text{cloudscape.DataComm.url=true} \) is one that uses the standard JDK URL support for post messages. SSL support is possible.
  The optional configuration uses the standard URL support provided by the JDK.

**NOTE:** The optional configuration is the default at a target if the listening address of the Cloudsync server \( \text{cloudscape.listener.synchronization.address} \) does not start with http: (typically, it would start with https: instead).

The optional configuration works only if the source database is configured to use the Servlet API \( \text{cloudscape.DataComm.listenType=servlet} \).

This optional configuration allows:

- Use of SSL (Secure Sockets Layer) for refresh messages.
  This requires that the server framework in which Cloudsync is embedded supports SSL (The Cloudconnector server framework supports SSL.). This also requires that the JVM the target is running in supports SSL or that the target’s application includes a package that provides the SSL support. (Netscape Navigator, Microsoft Internet Explorer, JavaSoft Java plugin 1.2.2).
  (In this case the target will automatically use the optional configuration as the Cloudsync listening address will start with 'https:')
- Use of Cloudscape synchronization with server frameworks that previously could not with the Cloudscape implementation of HTTP POST due to inconsistencies in the HTTP 1.0 specification for POST messages.
Cloudsync Properties

NOTE: When using the optional configuration (standard JDK URL support) targets may still run into JavaSoft bug 4212479, i.e. not having enough memory to send a large refresh message, unless the JVM uses HTTP 1.1 (e.g. Netscape Navigator, Microsoft Internet Explorer). In HTTP 1.1 the complete message is not required to be in memory before sending it.

The server framework may provide an HTTP tracing mechanism to determine if HTTP clients (e.g. Cloudsync targets) are talking HTTP 1.0 or HTTP 1.1. Cloudconnector provides such a facility.

NOTE: This property is valid only for target databases.

NEW: The cloudscape.DataComm.url property is new in Version 3.0. In releases prior to Version 3.0 the implementation of this HTTP client for refresh used java sockets, rather than the standard URL support provided by the JDK. This was to work around Javasoft bug 4212479 (See the Java Developer Connection at http://www.javasoft.com). The problem is that the JDK implementation of the POST message builds the message in memory until it was complete, this meant that it was impossible to send large messages in a memory restricted environment.

Default

false, unless cloudscape.listener.synchronization.address does not start with http:, in which case the default is true.

Example

# in cloudscape.properties file at target
cloudscape.DataComm.url=true

PropertyInfo.setDatabaseProperty(
   'cloudscape.DataComm.url', 'true')

CREATE PUBLICATION pubname
...
ADD TARGET DATABASE PROPERTY
   cloudscape.DataComm.url='true'

Scope

Database (publishable)
System
Dynamic or Static

Static. You must restart for the changes to take effect.
cloudscape.listener.synchronization.address

**Function**

The `cloudscape.listener.synchronization.address` property sets the URL to which targets send refresh messages. COPY PUBLICATION uses this property value for in the targets the copy file creates; if this property value changes, previously created targets do not automatically send refresh messages to this new address. This property is meaningful only at a source database.

The `cloudscape.listener.synchronization.address` property resembles a published property. You set it at the source, it is included in the copy file that creates the target, and it is used only at the target.

**Syntax**

```
cloudscape.listener.synchronization.address=address
```

**Default**

With the servlet API, there is no default value.

With built-in HTTP support, the default is:

```
http://hostname:port/application
```

where `hostname` is the result of the `java.net.InetAddress.getLocalHost().getHostName()` method, and `port` is 2001, unless it has been set to another value with the `cloudscape.DataComm.port` property. The word `application` is required.

**Example**

```
cloudscape.listener.synchronization.address=http://myhost:7001/application
```

**Scope**

System-wide
The `cloudscape.refresh.guaranteedRefreshInterval` property sets the maximum amount of time that a target database can wait between refreshes. If this interval is exceeded, the target can no longer be refreshed. If this happens, create a new target.

The `cloudscape.refresh.guaranteedRefreshInterval` property is set at the source and affects all targets connecting to that source. This property is only meaningful at a source database.

The `cloudscape.refresh.guaranteedRefreshInterval` property limits how much storage space is set aside to store historical information supporting target refreshes. Decreasing the interval frees up storage space. Increasing the interval does not allow targets that have already passed the previously set interval to refresh.

Make sure not to drop classes that are used to store data in published tables until the refresh interval has expired.

**Syntax**

```
cloudscape.refresh.guaranteedRefreshInterval=<integer> { h / d }
```

where `h` is hours and `d` is days.

**Default**

One week (7d).

**Example**

To set the refresh interval to five hours:

```
cloudscape.refresh.guaranteedRefreshInterval=5h
```

**Scope**

System-wide
**Cloudsync Properties**

**cloudscape.synchronization.workUnitOnly**

**NEW:** This property is new in Version 3.0.

**Function**

The `cloudscape.synchronization.workUnitOnly` property determines whether targets subscribing to a given publication are forced to use work units to update data. If `cloudscape.synchronization.workUnitOnly` is set to `true`, the targets accept only data changes (INSERTs, UPDATEs, and DELETEs) made within a work unit, and the refresh operation rejects any data changes not made using a work unit. Normally this property does not cause a refresh operation to fail, because the target rejects any non-work-unit (changed value) updates before they can participate in the refresh cycle.

You cannot set the `cloudscape.synchronization.workUnitOnly` property at the source or target; to set it, you must include it in a CREATE PUBLICATION or ALTER PUBLICATION statement.

Be careful when adding the `cloudscape.synchronization.workUnitOnly` property to an ALTER PUBLICATION command. If the target accepts non-work-unit updates, and it sends such an update to the source in the same refresh operation that sets the property to `true`, the refresh fails and the target can no longer be refreshed. This may be acceptable, since it is easy to create a new target. But to make sure that targets are not disabled by the `cloudscape.synchronization.workUnitOnly` property in an ALTER PUBLICATION statement, update the targets’ applications to allow only work unit updates before running the refresh operation that sets the `cloudscape.synchronization.workUnitOnly` property to `true` at the target.

**Syntax**

```
CREATE PUBLICATION pubname
  ...
ADD TARGET DATABASE PROPERTY
  cloudscape.synchronization.workUnitOnly=Boolean_value
```

**Default**

`false`

**Example**

```
CREATE PUBLICATION myPub
  ...
```

*Cloudscape Synchronization Guide*
ADD TARGET DATABASE PROPERTY
cloudscape.synchronization.workUnitOnly='true'

Scope
Database-wide

NEW: The `cloudscape.synchronization.workUnitOnly` property is new in Version 3.0.
sysinfo.publicationID

Function

The `sysinfo.publicationID` property gets the UUID of the publication that defines a target database. This property is used only at target databases. Source DBAs may ask target users for this information. The user at the target should not change this property.

Syntax

```
VALUES PropertyInfo.getDatabaseProperty('sysinfo.publicationID')
```

Default

N/A.

Example

See Syntax.

Scope

Database-wide
**sysinfo.publicationName**

**Function**
The `sysinfo.publicationName` property gets the name of the publication that defines a target database. This property is used only at target databases. Source DBAs may ask target users for this information. The user at the target should not change this property.

**Syntax**
VALUES PropertyInfo.getDatabaseProperty('sysinfo.publicationName')

**Default**
N/A.

**Example**
See Syntax.

**Scope**
Database-wide
**Cloudsync Properties**

*sysinfo.sourceDatabaseID*

**Function**

The `sysinfo.sourceDatabaseID` property gets the UUID of the source database that created the current target. This property is used only at target databases. Source DBAs may ask target users for this information. The user at the target should not change this property.

**Syntax**

```java
VALUES PropertyInfo.getDatabaseProperty('sysinfo.sourceDatabaseID')
```

**Default**

N/A.

**Example**

See Syntax.

**Scope**

Database-wide
**Function**

The `sysinfo.sourceDatabaseName` property tells the name of the source database that created the current target. This property is used only at target databases. Source DBAs may ask target users for this information. The user at the target should not change this property.

**Syntax**

```
VALUES PropertyInfo.getDatabaseProperty(
    'sysinfo.sourceDatabaseName')
```

**Default**

N/A.

**Example**

See Syntax.

**Scope**

Database-wide
**sysinfo.sourceServerAddress**

**Function**

The `sysinfo.sourceServerAddress` property indicates the address of the source database. This property is meaningful only at a target. The value of `sysinfo.sourceServerAddress` is the address specified with `cloudscape.listener.synchronization.address` at the source that created this target. Target users should not change `sysinfo.sourceServerAddress`.

**Syntax**

```java
VALUES PropertyInfo.getDatabaseProperty(‘sysinfo.sourceServerAddress’) 
```

**Default**

N/A

**Example**

```text
sysinfo.sourceServerAddress=http://myhost:7001/application 
```

**Scope**

Database-wide
PART THREE

Appendixes
Appendix A

Illustrations of the Refresh Operation

This appendix provides the following illustrations of the refresh operation:

- “Possible Outcomes of a Provisional Transaction” on page A-1
- “Timelines” on page A-3

**NOTE:** You do not need to read this appendix to develop synchronized applications. This appendix serves only to show graphically how the refresh operation works.

**Possible Outcomes of a Provisional Transaction**

Figure 1-1 illustrates what can happen to a provisional transaction before, during, and after a refresh. This illustration shows that there are only three possible outcomes when a provisional transaction is applied at the source (assuming that communication between source and target does not fail during the refresh):

- It applies durably at the source, is written back to the target, and is propagated to all other subscribing targets as they refresh.
- It fails at the source, is written into the error log, and disappears from both source and target user tables.
- It aborts the refresh, remains provisional, and is applied at a later refresh.

If the transaction applies successfully at the source, but then the refresh does not complete because of a communication failure, the target is correctly updated at the
next successful refresh. Cloudsync assures that provisional transactions apply once and only once at the source.

![Illustrations of the Refresh Operation](image)

**Figure 1-1** Possible outcomes of a provisional transaction

**NOTE:** Source changes can fail to apply at a target because they violate a target foreign key constraint. You can avoid this situation by designing your publications correctly. See “Handling Published Foreign Keys” on page 6-17 for details.
Timelines

The following two illustrations (Figure 1-2 and Figure 1-3) show the state of a target and the source at different times: after an update at the target, during a refresh, and after a refresh.

The queries applied at the target are shown on the target’s timeline. The circles on the timelines represent events, such as when a transaction is committed or when a refresh is issued. The boxes represent the content of tables at the given time. An asterisk (*) after an entry in a table means this value is from a provisional transaction. Values without asterisks are durable.

A Simple Scenario

Figure 1-2 shows a very simple scenario:

1. Two simple inserts to a single table commit at the target.
2. The target issues a refresh, which is successful.
3. The transactions are applied at the source and become durable at the target.
Illustrations of the Refresh Operation

Figure 1-2  A simple synchronization scenario
A Slightly More Complicated Synchronization Scenario

Figure 1-3 shows a slightly more complicated synchronization scenario than the one in Figure 1-2:

1. Two simple inserts commit at the target.
2. The target issues a refresh. One of the transactions fails at the source because of a check constraint.
3. Before the reply comes back from the first refresh operation, a different connection to the target completes a transaction.
4. After the refresh returns, the provisional transaction that succeeded at the source becomes durable at the target, and the provisional transaction that failed at the source is removed from the user tables and written to the error log.
5. A subsequent refresh sends the “stranded” transaction to the source. This transaction succeeds at the source and then becomes durable at the target.
Illustrations of the Refresh Operation

Figure 1-3  A slightly more complicated synchronization scenario
Index

A
Aliases
class and method in publications 2-2
ALTER PUBLICATION statement 2-3, 9-4
ALTER TABLE statement 2-3
Altering publications 2-3
Application code
distributing and updating 7-3
Authentication
of users 4-9
Authentication provider
specifying at a target 4-10
Authorization
of users 4-9

B
Backups
implications for synchronization 4-8
Built-in HTTP support 3-7

C
CALL statement
and work units 5-6, 5-7
checkJBMSException method 5-10
Class alias
in a publication 2-2
cloudscape.authentication.provider 4-10
cloudscape.database.defaultConnectionMode 4-9
cloudscape.database.fullAccessUsers 4-9
cloudscape.database.propertiesOnly 4-10
cloudscape.database.readAccessTarget 12-3
cloudscape.database.refresh 12-5
cloudscape.database.sourceUser 4-10, 12-6
cloudscape.database.synchronizationCleanup 12-8
cloudscape.DataComm.listenType 3-6, 12-9
cloudscape.DataComm.port 3-7, 12-10
cloudscape.listener.synchronization.address 3-7, 12-14
cloudscape.refresh гарантированная refreshInterval 12-15
cloudscape.synchronization.workUnitOnly 12-16
COM.cloudscape.database.checkJBMSException on method 5-10
COM.cloudscape.database.PropertyInfo.getDatabaseProperty method 3-3
COM.cloudscape.util.KeyGen.getUniversalKeyValue StringValue method 6-19
COM.cloudscape.util.KeyGen.getUniversalKeyValue Value method 6-18
Conflicts
definition 5-2
framework for handling 1-5
Consistency errors 4-3
Converting non-synchronization-enabled
databases to source databases 11-2
convertToSource 11-2
Copy files
creating target databases with 2-1
COPY PUBLICATION statement 9-6
Copying databases
implications for synchronization 4-8
CREATE PUBLICATION statement 9-8
CREATE WORK ALIAS statement 5-6, 5-6, 9-13
createSource 11-3
createTargetFrom 11-4
Creating a Cloudscape synchronization system 3-1
Creating a copy file
how to 9-6
overview 1-1
Creating a publication
how to 9-8
overview 1-1
Creating a target
overview 1-1
CURRENT_USER 12-6

D
Database-wide properties
verifying value of 7-6
DDL statements
not permitted at a target 2-1
Deployment
source database 7-1
target database 7-2
Design issues
and publications 6-15
DROP PUBLICATION statement 9-14
DROP WORK ALIAS statement 5-6, 9-15

E
Environmental errors 4-3
Errors
and work units 5-9
consistency 4-3
during the refresh operation 4-3
environmental 4-3
Exception handling
in work units 5-9

F
Failed transactions
viewing 8-1
Foreign keys
designing publications to correctly handle 6-17
publishing 2-4

G
getUniversalKeyStringValue method 6-19
getUniversalKeyValue method 6-18

H
Host name
specifying 3-5
HTTP
built-in support for 3-7
use in synchronization 3-5
Hub-and-spoke topology 1-3

I
Isolation levels
and work units 5-9
for COPY PUBLICATION 9-6

J
Jar files
altering published 9-5
changes in published 4-7
dropping from a publication 9-5
publishing 7-3
removing published 9-5

K
Keys
unique to the whole synchronized system 6-18

L
LOCK TABLE
default behavior in synchronized application 5-9
in work units 5-9

M
Method alias
in a publication 2-2

N
Network addresses
configuring 3-5

P
Parameters
finding values of at a target 9-6
in publications 2-2
Port numbers
specifying for source databases 3-5, 12-10
printFailedTransaction method 8-1
printFailedTransaction method
using with Cloudconnector 8-2
using with WebLogic 8-2
Properties
setting at the target 12-1
verifying 7-6
Provisional transactions 1-6, 4-2
errors and 4-3
Publications
altering 2-3
and ALTER TABLE statement 2-3
creating 2-1
definition 1-5, 2-1
design issues 6-15
overview 1-1

Cloudscape Synchronization Guide
parameters in 2-2
what can be contained in 2-2
what is implicitly included in 2-3
Published tables
altering 2-3
Publishing
triggers 9-11

R
Refresh interval
setting 12-15
Refresh operation 1-5, 4-2
concurrency during 4-6
errors during 4-3
required frequency of 4-6
steps of 4-2
REFRESH statement 4-2, 9-16
Refreshing copied databases 4-8

S
Security
using Cloudscape Synchronization 4-8
Servlet API 3-6
SESSION_USER 12-6
setDatabaseProperty 12-1
SkipTransactionSQLException 5-9
Source databases
creating from non-synchronization-enabled
databases 11-2
definition of 1-3
StopRefreshSQLException 5-9
Synchronizing system
creating 3-1
Synchronizing databases 1-5, 4-2
SYSERRORS system table 4-4, 8-1
SYSERRORS system table
extracting information from 8-1
removing records from 4-4
sysinfo.publicationID 12-18
sysinfo.publicationName 12-19
sysinfo.sourceDatabaseID 12-20
sysinfo.sourceDatabaseName 12-21
sysinfo.sourceServerAddress 12-22
SYSPUBITEMS system table 10-2
SYSPUBS system table 10-3
SYSSYNCINSTANTS system table 10-4

T
Tables
publishing altered 2-3
Target databases 1-3
creating 1-1
Targets
finding parameter values of 9-6
Topology
hub-and-spoke 1-3
Transactions
and work units 5-9
provisional 1-6, 4-2
seeing information on failed 8-1
Triggers
and the refresh operation 4-7, 9-11
in publications 2-4
publishing 9-11
target data changes not sent to the source 4-7, 9-11

U
Universal unique keys 6-18
Updates
changed-value 5-3
default 5-3
work unit 5-3, 5-4
User authentication 4-9
User authentication provider
specifying at a target 4-10
User authorization 4-9, 4-9
User-defined aggregates
in a publication 2-2

V
VALUES clause
and work units 5-6, 5-7

W
Work units
advantages of 5-5
altering one of the methods in 5-10
compared with method aliases 5-5
creating 5-6
definition 5-3
dropping 5-6
examples of 6-7
exception handling in 5-9
introduction to 1-6
invoking 5-6, 5-7
registering 5-6
sharing name space with method aliases 5-7
specifying a connection in 5-8
transactions inside 5-9
values returned from 5-6