Daffodil Replicator

Developer's Guide

August 2005

Table of Contents

Preface

Purpose

Daffodil Replicator Developer's Guide explains the various concepts and terminologies in Daffodil Replicator in detail. It explains how to work with Daffodil Replicator through JDBC API. The manual is illustrated with examples and cases wherever necessary. It describes the following:

1. The need for Replication and Synchronization.
2. Replicator Architecture – Publish and Subscribe model.
3. Various configuration and runtime details.
4. Replication technologies like Snapshot, Synchronization, Pull and Push replication.
5. General features of Daffodil Replicator.
6. Working with the Replicator JDBC API.
7. Possible error messages, explanations and work-arounds.
8. Various database pre-requisites and known limitations.

Replicator Developer's Guide focuses on working with Daffodil Replicator through JDBC API. For information on working with GUI or Console, see *Daffodil Replicator Console Guide*.

Audience

Daffodil Replicator Developer's Guide is intended to act as a ready reference tool for database administrators, developers, application programmers and any user who wish to replicate their databases through JDBC API.

It assumes that you have basic knowledge of the following:

- Server and Client concepts.
- Structured Query Language (SQL).
- Java programming language.
- Java Database Connectivity (JDBC) API
- RDBMS concepts.
- Understand terminology such as synchronization and classes.

Replicator Overview

Need for Data Replication

We are living in a world that is governed by Murphy's Laws, where anything can go wrong at any time. Accidents ranging from machine crashes, media failures, operator errors and random data corruption to such catastrophes as floods, earthquakes and terrorist attacks like that of 9/11 can result in loss of data, both temporarily and permanently. It is important for us to understand that data accidents like these cannot be eliminated fully, but can only be minimized. Data Replication between different sites, is a method to minimize such data losses.

In present-day organizations, data is distributed throughout the enterprise. The ability to manage and synchronize enterprise data between remote databases and across platforms has become an essential pre-requisite for any kind of business in the current scenario. For instance, the users connected to the enterprise may need to synchronize data between field sites and database systems at corporate headquarters occasionally. In Daffodil Replicator, you will find a scalable and lightweight tool, which allows you to replicate and synchronize data between databases.

Daffodil Replicator offers synchronization capability for data synchronization amongst all *enterprise databases,* which provide JDBC drivers.

Distributed Databases Vs Replication Technology

In recent years, DBAs have discovered that by moving data from mainframe and distributing it across multiple sites, one can minimize communication costs and increase the availability of data that resides locally. But a distributed system falls short when you run applications requiring frequent access to data which resides at multiple, remote sites.

Replication technology is a software solution that can resolve many data access problems associated with such applications. Also, carefully designed replication software can provide fault tolerance, high availability, and disaster recovery mechanisms!

Introduction to Daffodil Replicator

Daffodil Replicator is based on the simple idea of copying and maintaining data from one location to another. However, this simple idea has many practical applications like data protection and recovery, data synchronization, migrating data to several locations or centralizing data to one location.

Daffodil Replicator works within your current infrastructure, whether your clients use any compatible databases (Oracle, SQL Server, DB2, Sybase, Daffodil DB, PostgreSql, Derby and Firebird). It can also replicate data across a LAN or WAN for local or remote solutions. Because Replicator is capable of replicating data from both servers and workstations, everyone's data is protected and with Replicator's user-friendly, browser-based Graphical User Interface, it is simple enough for every employee in your company to use.

Daffodil Replicator enables data synchronization between data sources located anywhere. It supports a variety of synchronization modes and synchronizes data sources by using TCP/IP transfer protocols.

Replicator is designed for Java database applications. It resides as a layer between the JDBC API and client Java applications. Replicator can be integrated with all Daffodil database products as well as selected third-party databases that provide JDBC drivers.

Features of Daffodil Replicator

Significant features of Daffodil Replicator are listed below:

- **Supports multiple data sources** (**Heterogeneous database replication**) – Synchronization between *enterprise databases* running on J2EE server platform, *Workgroup databases* running on J2EE/J2SE platforms and *Java-enabled mobile devices* running on J2SE/J2ME platforms.

- **Application Independence** – Daffodil Replicator API allows the software to be fully embedded in any application and IT environment.

- **Conflict detector and resolution** – Daffodil Replicator offers a variety of automatic conflict detection and resolution functionalities that allow multiple users to send in changes from the workstations simultaneously, while ensuring the integrity of the data.

- **Platform independent synchronization** – 100% Java enabled client and server components allow Replicator users to work platform independently.

- Database-to-Database **Bi-directional Data Synchronization** with client-side initiating the synchronization process.

- ***Publish and Subscribe model*** with a Publisher (Server) and Subscriber (Client) architecture.

- **Comprehensive reloads and incremental updates** – Snapshot replication and Merge replication allows copying all the data (snapshot) or making delta changes (synchronization).

- **Push & Pull Replication –** This feature allows Subscribers to Push updated data to Publisher or Pull updated data from Publisher without merging the changes at both sides as in the case of Merge Replication.

- **Modify Publication and Subscription** – This feature allows Publisher to modify publication by adding more tables in existing publication or dropping tables from existing publication. Whenever publication is modified by Publisher, subscriber must get updated its Subscription.

- **Scheduling** – This feature allows subscribers to schedule (Real Time or Non Real Time) the replication process (Snapshot, Synchronize, Pull and Push).**Edit Schedule** facility is also supported to edit publisher server name or port no. (If publisher is running on different URL or Port from the URL or Port specified at time of add Schedule).If a user no longer need a schedule and wants to remove it, it can be done by **Remove Schedule** facility.

- **Debugging –** Replicator is providing the facility of ***debugging*** using **Log4j.jar**. In this file, user can set the logging option as per his requirement.

- **Special Characters Handling** – Special characters with ASCII value less than 32 are not parsable and create problem in replication operations. This can be fixed by using **EncodeConfig.ini** file. It is necessary to enter the column name (those having special characters) in **EncodeConfig.ini** file against the respective table name.

- **Transaction Log File –** Transaction log file maintains the record of the transactions performed by Replicator during replication process. There are two types of transaction detail ***full*** and ***partial.*** If the property '**TRANSACTIONDETAIL**' is set to **TRUE** then it will contain the '**Detailed**' (Full) information of the transactions that are performed during replication process, if **'TRANSACTIONDETAIL'** property is set to **FALSE** then it will give information about number of records only.

- **Change Replication Home –** Replication home is a path where transaction logs, error log and XML files are created. XML files come in use during replication process when snapshot, synchronization, push and pull operations are performed. User can set the replication home at specified path according to requirement. If it is set to default as given below then all above mentioned file will be created in user home else at the path specified by the user.

- **Partial Tables Replication –** A user can ignore the columns that he do not want to replicate. Columns that do not allow insert,update and delete operations, create problem in data replication.
  If any one working with API use the following statement to ignore the columns.
  void setIgnoredColumns("tableName", new String[]{"col1","col2",…………})
  throws RepException
  First parameter is name of table from a user want to ignore the columns. Second is array of columns.

- **Drop all system tables** if no more publications and subscriptions exist.

For more information about the various features of Daffodil Replicator, see http://www.daffodildb.com/replicator-key-feature.html

For a description of various advantages of using Daffodil Replicator, see http://www.daffodildb.com/replicator-advantage.html

Compatible Databases

Daffodil Replicator is compatible with the following databases:

1. **Daffodil DB**
2. **Oracle**
3. **SQL Server**
4. **PostgreSql**
5. **Derby (Cloudscape)**
6. **DB2**
7. **Sybase**
8. **Firebird**

Further, developers can make Daffodil Replicator compatible with any database (because Daffodil Replicator is an Open Source software and the code is freely available) that conform to certain pre-requisites.

The database should support the following features:

- **Auto Increment/Sequence**
- **Triggers/Procedure**
  - **(a)** Row Level Triggers
  - **(b)** Variable Declaration
  - **(c)** Alias
- **Connectivity with JDBC**
- **Primary Key** – All the tables that are to be included in replication should have a Primary key.

Daffodil Replicator Architecture

The architecture of Daffodil Replicator is described in terms of a **Publisher/Subscriber** metaphor. This section explains how Replicator conforms to the *Publication* and *Subscription* when synchronizing different databases. It defines the minute details of each involved components. It also illustrates the set-up and configuration process for Replicator and its synchronization operations performed during runtime.

Publish and Subscribe model

The ***Publication*** and ***Subscription*** allows a *Publisher* to publish data to one or more *Subscribers* where Replication server enacts the role of a listener on a pre-defined communication port and the Subscriber makes periodic requests for synchronization operations.

Publisher and Subscriber

Publishers are the basic building block of a *Replicator's synchronization network*. Each Publisher forms part of a client-server network and is assigned a unique name. The name as well as other details of each Publisher is stored in the Publisher database.

By definition, **a Publisher must have associated with it:**

- **A unique IP address and a port number**

Subscriber subscribes to Publisher's Publications. By definition, **a Subscriber must have associated with it:**

- **A Publisher IP address and a port number**
- **A Synchronized data source**

Daffodil Replicator Publisher and Subscriber

Daffodil Replicator Publisher and ***Subscriber*** are responsible for managing the synchronization of data across the network. Both the Publisher and Subscriber are objects instantiated by user applications and perform the core functions associated with the synchronization of data. The Replication Server API gives method calls to Publisher and Subscriber, which enable Subscriber to perform synchronization operations. A *Replicator Publisher* has the ability to handle multiple Subscriber connections simultaneously whereas a *Replicator Subscriber* maintains only a single connection. Multiple Subscribers may communicate to a single Publisher on a communication port, which is specified to that particular Publisher.

Data Source

A **data source** is a **JDBC-enabled (SQL) database** that acts as a repository of data to be synchronized between a Publisher and a Subscriber. Each Publisher and Subscriber defines one or more JDBC-enabled database as its data source. Users have the choice to specify all or select number of tables from the data source (of Publisher) for synchronization.

Daffodil Replicator Data Source

Daffodil Replicator architecture consists of ***Publishers*** and ***Subscribers*** wherein each maintains a data source, which holds the data to be synchronized as well as information related to Publication and/or Subscription.

There is one to one mapping between Publisher/Subscriber and data source i.e. only one data source for each Publisher/Subscriber can exist. A data source comprises of database URL, driver, user name and password.

Publication and Subscription

A **Publication** is a collection of one or more tables defined by the developers. It specifies what data contained in the data source will be published. Publishers publish only selected data that exists in their data source. Publishers typically publish data from a subset of data source tables.

A **Subscription** identifies the data received from a Publisher's Publication and is stored in the Subscriber's data source. Subscription is typically a subset of a particular Publication on a Publisher. Subscriber data sources must contain tables that are either identical to or compatible with Publisher data source tables.

Daffodil Replicator Publication and Subscription

Daffodil Replicator allows any number of Publications to be made based on different combination of tables in the Publisher database. Also a Publication can be subscribed by any number of Subscribers. Each Publication should have a unique name. Developers can also specify conditions on tables by setting *filters* by which only those records in the table which satisfy the conditions are available for Subscription by the Subscriber. While making a Publication, you can also specify the conditions for ***Conflict Resolution***.

Subscriber can make any number of Subscriptions but a Subscription can connect only to a single Publication. Daffodil Replicator allows Subscriptions to be created only if at least one Publication is already available.

Stating again, a Publication can connect to multiple Subscriptions but the reverse is not true.

In the Replicator *publish and subscribe model*, Subscribers pull complete copies (**snap shot**) or partial (delta changes) copies of data to each other (**merge replication**) to maintain data synchronization. To pull only the Publisher changes (subscriber changes will not get updated to Publisher), Subscriber performs **Pull replication** whereas to push only the Subscriber changes to Publisher (Publisher's changes will not get updated to Subscriber), Subscriber performs **Push replication**.

Publication Pre-requisites

- **Primary key requirement** – a table which is included in Publication should have the primary key.

Subscription Pre-requisites

- Publication server should be started before starting Subscription server.
- At least one Publication should exist in the Publisher's database.

Configuration and Runtime

Replicator set-up and operation can be divided into two parts:

- Configuration
- Runtime

Configuration

The configuration part of the Replicator API consists of classes and methods to create, set up, and manage Publisher, Subscriber, data sources, Publications and Subscriptions for synchronization operations.

Configuration involves creation of Replication Home, Publishers and Subscribers, selection of data sources, defining/creating of Publications and Subscriptions, setting logging properties, special character handling and Transaction Log File settings.

All the Replicator directory files that include error log file and transaction log file are created at Replication Home. Users can set Replication Home at desired / user specific directory. By default, Replication Home is set to user home.

While creating a Publication, one or more tables must be specified which are to be published. Each table to be specified in **Publication** must have a primary key. During configuration, you can specify ***Row Filtering*** and ***Conflict Resolution*** as well.

Each time the synchronization operation (Synchronization, Pull and Push) is executed a transaction log file will be created in Replication Home set both on Publisher and Subscriber data source. Transaction log file stores the statistical or detailed information regarding the data changes reported (i.e. Update, Insert or Delete) in Publisher and Subscriber data source.

Runtime

The runtime part of Replicator API consists of classes and methods to invoke synchronization operations. During runtime, following *methods* are called

Snapshot ()

Synchronize ()

Pull ()

Push ()

addTableToPublication (String [ ] newTableList, String [ ] filterClauses)

dropTableFromPublication (String [ ] dropTableList)

updateSubscription()

addSchedule ("schedule Name"," subscription Name"," schedule Type", "Publication Server Name", "publication Port No", "Schedule Type"," Replication Type, "start Date Time", "Schedule Counter")

editSchedule ("scheduleName","subscriptionName", "New Publication Server Name", "New publication Port No")

removeSchedule ("scheduleName","subscriptionName")

where ***snapshot ()*** invokes an operation to retrieve a complete copy of refreshed data and ***synchronize ()*** ensures the integrity and consistency of data propagated among the data sources by merging the updates to both the sides. ***Pull ()*** method pulls the Publisher changes to Subscriber's database whereas ***Push ()*** pushes the Subscriber changes to Publisher's database.

When Publisher want to modify his existing Publication, he can use ***addTableToPublication (String [ ] newTableList, String [ ] filterClauses)*** to add tables or ***dropTableFromPublication (String [ ] dropTableList)*** to drop tables. Now subscriber has to get modified its subscription using ***updateSubscription ().***

When subscriber want to add schedule for any replication process ,subscriber can use *addSchedule ("schedule Name"," subscription Name"," schedule Type", "Publication Server Name", "publication Port No", "Schedule Type"," Replication Type, "start Date Time", "Schedule Counter").*When user has to change publisher's Publication Server Name and publication Port No., *he can invoke editSchedule ("scheduleName","subscriptionName", "New Publication Server Name", "New publication Port No").*When subscriber has to drop schedule he can invoke *removeSchedule ("scheduleName","subscriptionName")*

Following are the runtime operations:

- Snapshot
- Synchronize
- Pull
- Push
- Modification of Publication or Subscription
- Scheduling (on subscriber side)
- Unsubscribe
- Unpublish

Partial Tables Replication

Replicator supports Partial Table Replication that allows user to ignore specific columns in a publisher database table that he / she do not want to replicate. While defining a publication, user has the option to include only those columns from the publisher database that needs to be replicated on the client / subscriber side. Partial table replication ignores the data residing in publisher database columns that is not required on the client side and hence makes replication operations more efficient. If the publisher database includes columns that do not allow insert, update and delete operations user can exclude such tables while creating Publication and ensure smooth data replication.

While working through API, use the following statement to ignore the multiple columns:

void setIgnoredColumns(“tableName”, new String[]{“col1”,”col2”,…………}) throws RepException

“tableName”:	Name of the table in publisher database that is to be included in Partial Replication operation.
{“col1”,”col2” …}:	Array of columns to be ignored.

Replication Methods supported by Daffodil Replicator

Daffodil Replicator supports four distinct types of Replication methods: Snapshot, Merge, Pull and Push replication, each of which has its own purpose.

Snapshot Replication

Snapshot Replication operation transfers a *complete* copy of the contents of a Publisher's Publication to a Subscriber's Subscription. Snapshot replication distributes data exactly as it appears at a specific moment in time and does not monitor for updates to the data. This type of replication is usually used when a Subscriber start to synchronize the data from the Publisher for the first time and an exact copy of tables from Publication is desired. Taking a snapshot of the tables in Publication will create an exact copy in the Subscription database.

After making the snapshot, Publisher and Subscribers can work independently while being disconnected from each other.

Although snapshot replication is the easiest type to set up and maintain, it requires copying all data each time a snapshot is taken.

Note

- Taking the snapshot of Publisher data (a one-time process) by the Subscriber is a pre-requisite for Merge, Pull and Push replication.
- Snapshot replication can **only** be initiated by the Subscriber.
- Row level filtering can be applied on tables which will limit the data published to the Subscriber side. In this case the Subscriber will not get the exact copy of database tables.

Merge Replication or Synchronization

Merge replication is the process of distributing data between Publisher and Subscribers by allowing the Publisher and Subscribers to make updates while connected or disconnected, and then merging the updates between sites when they are connected.

In this type of Synchronization operation, changes made to the data by both Publisher and the Subscriber are merged, but the synchronization is always initiated by the Subscriber. Before attempting Merge replication (for the first time), it is important to take the snapshot (a one-time process) of data from the Publisher. Once the snapshot is taken, you can perform merge replication any number of times. Merge Replication is also called as Change-based Replication or Synchronization. Unlike snapshot replication, synchronization is used to update minimal (delta) changes taking place at Subscribers as well as Publishers database.

Because updates are possible both at Publishers and Subscribers end, the same data may have been updated by the Publisher and the Subscriber. Therefore, conflicts can occur when updates are merged. In this scenario, Merge replication merges the data according to the **conflict detector mechanisms** specified during the creation of a Publication.

Note:

- The data is synchronized between a Subscriber and a Publisher only when it is initiated by the Subscriber.
- Daffodil Replicator does not support **simultaneous** synchronization between multiple Subscribers and a Publisher.
- Taking the snapshot of Publisher data (a one-time process) by the Subscriber is a pre-requisite for Merge replication.

Pull Replication

Pull replication is the process of distributing data between Publisher and Subscribers by allowing the Publisher and Subscribers to make updates while connected or disconnected, and then *pulling* the updates of Publisher (server) to Subscribers (client sites) when they are connected.

Pull replication compares Publishers and Subscribers data and the changes made to the data by the Publisher are updated in the Subscriber's database. But the changes made by the Subscriber are not updated to the Publisher's database. So in Pull replication, records (that are changed due to insert, update and delete queries at Publisher's database) are *pulled* from Publisher to Subscriber by the Subscriber. Before attempting Pull replication (for the first time), it is important to take the snapshot (a one-time process) of data from the Publisher. Once the snapshot is taken, you can perform Pull replication any number of times.

While Pull replication always keep client side (Subscriber) updated with changes happening at Publisher side, changes performed at Subscriber side are not reflected at Server side (Publisher). This replication is suited when the Server at Publisher side wants to propagate server-side updation to all its client sites while ensuring that data changes by individual clients will not get updated to the main server.

Because updates are possible both at Publishers and Subscribers end, the same data may have been updated by both the Publisher and the Subscriber. Therefore, conflicts can occur when updates are pulled. In this scenario, Pull replication updates the data according to the **conflict detector mechanisms** specified during the creation of a Publication.

Note

- The data is pulled from Publisher to a Subscriber only when it is initiated by the Subscriber.
- Daffodil Replicator does not allow Publisher to pull data from Subscriber. It uses Push technology to perform this task.
- Daffodil Replicator does not support **simultaneous** pulling of data from Publisher to multiple Subscribers.
- Taking the snapshot of Publisher data (a one-time process) by the Subscriber is a pre-requisite for Pull replication.

Push replication

Push replication is the process of distributing data between Publisher and Subscribers by allowing the Publisher and Subscribers to make updates while connected or disconnected, and then *pushing* the updates of Subscriber (client) to Publisher (server) when they are connected.

Push replication compares Publishers and Subscribers data and the changes made to the data by the Subscriber are updated in the Publisher's database. But the changes made by the Publisher are not updated to the Subscriber's database. So in Push replication (for the first time), records (that are changed due to insert, update and delete queries at Subscriber's database) are *pushed* to Publisher by the Subscriber. Before attempting Push replication (for the first time), it is important to take the snapshot (a one-time process) of data from the Publisher. Once the snapshot is taken, you can perform Push replication any number of times.

While Pull replication always keep server side (Publisher) updated with changes happening at Subscriber side, changes performed at Publisher side are not reflected at client side (Publisher). This replication is suited when all the clients at Subscriber side want to propagate client-side updation to the main server (Publisher) while ensuring that data changes by the server will not propagate to the clients."

Because updates are possible both at Publishers and Subscribers end, the same data may have been updated by both the Publisher and the Subscriber. Therefore, conflicts can occur when updates are pushed. In this scenario, Push replication updates the data according to the **conflict detector mechanisms** specified during the creation of a Publication.

Note

- The data is pushed to Publisher by the Subscriber **only** when it is initiated by the Subscriber.
- Daffodil Replicator does not allow Publisher to push data to Subscribers. Subscriber uses Pull technology to perform this task.
- Daffodil Replicator does not support **simultaneous** pushing of data to Publisher by multiple Subscribers.
- Taking the snapshot of Publisher data (a one-time process) by the Subscriber is a pre-requisite for Push replication.

Row Filtering

A Publication represents data from selected Publisher tables, which is filtered according to a specified criterion.

Filters specified by Publishers on the tables of a Publication determine what data will be made available to a Subscriber. Regardless of where the filter is specified, the filter applies only to data in Publication tables accessed by the Publisher.

Filters are always applied to Publication *only*. Users should always specify filters such that the filter refers to the names of tables and columns in the Publication. When Subscribers pull data from a Publication, the filters on the Publication are combined and applied to the Publication tables, Publishers do the actual filtering. Because row filtering can only be applied at the Publisher, Replicator supports row filtering for *get* operations. Row filtering is only applicable to getSnapshot () and synchronize () calls from Subscriber so that Subscriber gets filtered data.

Filter Formats

Row filtering – General syntax for a row filter that refers to a particular Publication is:

<Publication>pub1.setFilter (tableName, condition) ;

Where the first parameter tableName refers to the name of the table on which the specified condition (2nd parameter) is applied.

Example:

Publisher Table (Table1)

Regional Id	Product
1	P_1
2	P_2
3	P_1
4	P_1
5	P_1

Subscriber Table (Table1)

Regional Id	Product
4	P_1
5	P_1

Suppose a Publication "pub1" applies a filter to the data in Table 1 to create a Publication for Subscribers, based on records 4 and 5 as displayed above. Row 1 - 3 will not be included at Subscriber's end. If Daffodil DB were acting as the data source for the Publisher, the filter on Table1 of pub1 might look like this:

pub1.setFilter ("Table1", "Table1.RegionalId >= 4 ") //condition should be a Boolean Expression.

Conflict Detection and Resolution

Daffodil Replicator is capable of detecting and resolving conflicts that tend to occur while synchronizing data. Conflict Resolver comes into play only when operation has been performed on same records on both Server side as well as on Client side.

This Section takes into consideration a common conflict scenario, and then describes how to apply conflict detection and resolution in the Replicator. A typical conflict occurs in case the Publisher and the Subscriber update same row with different values and then try to synchronize. A simple scenario on how a conflict can occur is described as follows:

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Suppose pub1 changes the Roll no 30 to 40 and sub1 changes the Roll no 30 to 50, as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 50	Santa

Now, sub1 calls for synchronization. If conflict resolver is set to “subscriber_wins” then, at Publisher pub1, Roll no is changed to 50, otherwise, Roll no for Subscriber sub1 is changed to 40 because, by default, the conflict resolver is set to “publisher_wins”.

Synchronization Cases

A complete process for using the Replicator is illustrated in the form of an Example in Appendix D.

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

After taking the snapshot, data can be changed by the Publisher as well as Subscriber by a number of ways. All the possibilities are discussed below:

Update-Update

Case 1

Both the nodes have modified the same field i.e. pub1 has modified a field and the same field in the same row is modified by sub1. Data synchronization will be based on conflict resolver settings, whether it was set to "publisher_wins" or "subscriber_wins" during creation of Publication pub1. After individual updation, both the tables will look like the following.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ Santa A

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ Santa B

In this case, Conflict Resolver plays an important role as operation has been performed on both sides for the same field i.e. same primary key. If conflict resolver is set to "publisher_wins", then, after synchronization, data on both sides will be as follows:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa A

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa B~~ Santa A

If conflict resolver is set to "subscriber_wins", then, after synchronization, data on both sides will be as follows:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa A~~ Santa B

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa B

Case 2: (Update-Update)

Pub1 and Sub1 have updated different fields in the same row as follows:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ Santa B

Irrespective of what conflict resolver has been set to, after synchronize call from Sub1, the data will be synchronized at column level as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
40	~~Santa~~ Santa B

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa B

Case 3: (Update-Update)

Suppose we have the following records after a snapshot is taken at the Subscriber's end.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

After updating records on both sides, we have data as mentioned below:

Sequence of operation performed by Publisher on Roll no (Primary key) are:

1 ► 9 updated rollno 1 to 9
9 ► 1 updated rollno 9 to 1
8 ► 9 updated rollno 8 to 9
7 ► 8 updated rollno 7 to 8
6 ► 7 updated rollno 6 to 7
5 ► 6 updated rollno 5 to 6
4 ► 5 updated rollno 4 to 5
3 ► 4 updated rollno 3 to 4
2 ► 3 updated rollno 2 to 3
1 ► 2 updated rollno 1 to 2

Sequence of operations done at Subscriber's end is:
8 ► 9 updated rollno 8 to 9

Student Table – pub1

Roll no	Name
~~1~~ .. ~~9~~ .. ~~1~~ .. 2	Adam
~~2~~ .. 3	Eve
~~3~~ .. 4	Santa
~~4~~ .. 5	Michael
~~5~~ .. 6	Lina
~~6~~ .. 7	Sabby
~~7~~ .. 8	George
~~8~~ .. 9	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
~~8~~ .. 9	Jordan

After synchronization when Conflict Resolver has been set to "subscriber_wins" we have the following records in both the tables at Publishers and Subscriber's end.

Student Table – pub1

Roll no	Name
2	Adam
3	Eve
4	Santa
5	Michael
6	Lina
7	Sabby
8	George
9	Jordan

Student Table – sub1

Roll no	Name
~~1~~ .. 2	Adam
2	Eve
~~3~~ .. 4	Santa
~~4~~ .. 5	Michael
~~5~~ .. 6	Lina
~~6~~ .. 7	Sabby
~~7~~ .. 8	George
9	Jordan
3	Eve

Explanation:

Record with roll no 1 on Subscriber's end was updated to 2 because no operation was performed on this record by the Subscriber (In this scenario, conflict resolver does not play a role). So record with roll no 2 was deleted from sub1 as we can't have two records with the same primary key in a table. Before synchronization, record with roll no 2 on pub1 was updated to roll no 3. So Publisher also tries to update the record with roll no 2 on Subscribers end at the time of synchronization. As this record with roll no 2 was deleted from sub1, a new entry with roll no 3 will be inserted in sub1.

Case 4: (Update-Update)

Suppose we have following records after a snapshot is taken at the Subscriber's end.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

After updating records on both the sides we have data as mentioned below:

Sequence of operations performed by Publisher on Roll no (Primary key) are:
1 ► 9 updated rollno 1 to 9
2 ► 10 updated rollno 2 to 10
3 ► 11 updated rollno 3 to 11
4 ► 1 updated rollno 4 to 1
5 ► 2 updated rollno 5 to 2
6 ► 3 updated rollno 6 to 3
7 ► 4 updated rollno 7 to 4
8 ► 5 updated rollno 8 to 5
12 Inserted a new record

Sequence of operation performed at Subscribers end
6 ► 12 updated rollno 6 to12
9 ► Inserted a new record
10 ► Inserted a new record
11 ► Inserted a new record

Student Table – pub1

Roll no	Name
~~1~~ .. 9	Adam
~~2~~ .. 10	Eve
~~3~~ .. 11	Santa
~~4~~ .. 1	Michael
~~5~~ .. 2	Lina
~~6~~ .. 3	Sabby
~~7~~ .. 4	George
~~8~~ .. 5	Jordan
12	King

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
~~6~~ .. 12	Sabby
7	George
8	Jordan
9	Rome
10	Egypt
11	Spain

After synchronization when Conflict Resolver has been set to “subscriber_wins”, we have the following records in both the tables at Publishers and Subscriber’s end.

Student Table – pub1

Roll no	Name
9	Adam
10	Eve
11	Santa
1	Michael
2	Lina
~~3~~ .. 12	Sabby
4	George
5	Jordan

Student Table – sub1

Roll no	Name
~~1~~ .. 9	Adam
~~2~~ .. 10	Eve
~~3~~ .. 11	Santa
~~4~~ .. 1	Michael
~~5~~ .. 2	Lina
12	Sabby
~~7~~ .. 4	George
~~8~~ .. 5	Jordan

12	King

9	Rome
10	Egypt
11	Spain

Explanation:

New records inserted at Subscriber's end are deleted because records at the Publisher's end on which updation had been performed were not operated on the Subscriber's side and hence, conflict resolver does not play a role (Records with roll no 1-3 has been changed to 9, 10 and 11 due to the updation by the Subscriber). Similarly record no 12 inserted by the Publisher is also deleted.

Case 5: (Update- Update)

Suppose we have following records after a snapshot is taken at the Subscriber's end.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

After updating records on both sides we have data as mentioned below:

Sequence of operations done at the Publisher's side is
7 ► 9 changed rollno 7 to 9
7 ► Inserted a new Record
Whereas at the Subscriber's side operation was performed as follows:

8 ► 9 changed rollno 8 to 9

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
~~7~~ .. 9	George
8	Jordan
7	newRecord

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
~~8~~ .. 9	Jordan

After synchronization, when Conflict Resolver has been set to "subscriber_wins" we have the following records in both the tables at Publishers as well as Subscriber's end.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
9	George
8	Jordan
7	newRecord

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
~~7~~ .. 9	George
9	Jordan
7	newRecord

Explanation:

As record with roll no 7 on the Subscriber's end was not operated upon so we have to update 7 to 9 without considering any conflict resolver, which makes us delete record with roll no 9 on the Subscriber's side as well as record with roll no 8 on the Publisher's side. After this, the inserted record with roll no 7 by the Publisher will be added to the Subscriber as well.

Update- Delete

Case 1

Pub1 or Sub1 has updated a record while the other had deleted the same record.

Example:

Suppose that the Publisher had updated the record with roll no 30 to 40 whereas the Subscriber had deleted the same record.
The situation is illustrated below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~30~~	~~Santa~~

After synchronization, if conflict resolver was set to “subscriber_wins” then updated record at Pub1 will also be deleted. Otherwise, if Conflict resolver was set to “publisher_wins” then a new row with 40 – Santa will get inserted at Sub1 as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
40	Santa

Note: Suppose, only one node i.e. (Publisher or Subscriber) modifies (Insert/Update/Delete) the data, in this case, modification will be operated on the other node irrespective of the conflict resolver.

Case 2: (Update-Delete)

Suppose we have following records after a snapshot is taken at the Subscriber's end.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

After updating records on both the sides we have data as mentioned below:

Sequence of operations performed by Publisher on Roll no (Primary key) are:

1	►	10	updated rollno 1 to 10
10	►	1	updated rollno 10 to 1

In addition, all the records are deleted from Subscriber.

This is shown in the following table:

Student Table – pub1

Roll no	Name
~~1~~ .. ~~10~~ .. 1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

After synchronization, when Conflict Resolver has been set to "publisher_wins" we have following records in both the tables at the Publishers as well as the Subscriber's end.

Student Table – pub1

Roll no	Name

Student Table – sub1

Roll no	Name

Explanation:

As the record with roll no 1 on Publisher has been updated to 10 and then updated to 1 i.e. from finally updated from 1 to1, which we consider as NO_OPERATION means that again conflict resolver will have no role in this regard, so all the records at the Publisher's side are also deleted.

Update – Insert

Case 1
One node has modified the record while the other has inserted a new record with primary key with value same as that of the updated record on the first node.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Balance

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

If Conflict Resolver is set to publisher_wins then after Sub1 synchronizes the data, updation at Sub1 for its conflicting row will be rolled back and the new row which was inserted at pub1 will be created at sub1 also.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Balance

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~40~~ 30	Santa
40	Balance

However, if Conflict Resolver is set to subscriber_wins then after Sub1 synchronizes the data, the insertion done at Pub1 will be deleted and the row with toll no 30 will be updated at Pub1 based on the changes at Subscribers end.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa
~~40~~	~~Balance~~

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
40	Santa

Special Case

Following are the records in the table "Student" after a snapshot had been taken by the Subscriber.

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Clown

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Clown

After updating records on both the sides we have data as mentioned below:

Sequences of operations performed on the Publisher are:

1 ► 0 updated rollno 1 to 0
2 ► 1 updated rollno 2 to 1
3 ► 2 updated rollno 3 to 2
4 ► 3 updated rollno 4 to 3
4 ► record Inserted
0 ► record Deleted
2 ► 0 updated rollno 2 to 0
2 ► record Inserted

Sequences of operations performed on the Subscriber are:

1 ► 0 updated rollno 1 to 0
4 ► 1 updated rollno 4 to 1
1 ► 4 updated rollno 1 to 4
2 ► 1 updated rollno 2 to 1
3 ► 2 updated rollno 3 to 2

Student Table – pub1

Roll no	Name
1 .. 0 . Deleted	Adam
2 .. 1	Eve
3 .. 2 ..0	Santa
4 .. 3	Clown
4	newRecord_4
2	newRecord_2

Student Table – sub1

Roll no	Name
1 ..0	Adam
2 .. 1	Eve
3 ..2	Santa
4 ..1 .. 4	Clown

After synchronization, when Conflict Resolver had been set to “subscriber_wins” we have the following records in both the tables at the Publishers as well as the Subscriber’s end.

Student Table – pub1

Roll no	Name
0 (Inserted)	Adam
1 .. 1	Eve
0 .. 3 .. 2	Santa
3 .. 4	Clown
4	newRecord_4
2	newRecord_2

Student Table – sub1

Roll no	Name
0	Adam
1	Eve
2	Santa
~~4~~ .. 3	Clown
4	newRecord_4

Explanation:

Record with roll no 2 was not inserted as at that time a record was already present with roll no 2 at Subscriber’s side, which was set as conflict resolver. Record with roll no 4 was inserted as newRecord_4 at Subscriber’s side, because roll no 4 was updated to roll no 3 earlier.

At Publisher’s end, *“primary unique constraint violation”* occurs due to the Subscriber’s call for updating roll no 1 to roll no 0, which was deleted at Publisher’s end before synchronization. Due to this violation, record with roll no 0 was inserted at pub 1 calling for roll backing the operations done at the Publisher’s end. This will finally result in the deletion of the new record with roll no 4 from pub 1.

Record with roll no 3 at pub 1 will be changed to roll no 2 as the conflict resolver is set to subscriber_wins. This will delete the new record with roll no 2 from the Publisher's table.

Note: Changes are transferred in an XML file initially from Publisher to Subscriber and then changes at the Subscriber's end are transferred from Subscriber to the Publisher's end.

Pull Replication cases

A complete process for using the Replicator is illustrated in the form of an Example in Appendix D.

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

After taking the snapshot, data can be changed by the Publisher as well as Subscriber by a number of ways. Some of the possibilities are discussed below:

Insert

Case 1 - Suppose that a record with roll no 40 is inserted by the Publisher. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Now perform Pull replication. The new record which was inserted by the Publisher will be pulled to Subscriber's database also. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Case 2 - Insert case with conflict resolver

Tables after Snapshot

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Operations performed

Publisher's side - a record with Roll no 40 and Name *Sabby* is inserted.

Subscriber's side - a record with Roll no 40 and Name *George* is inserted.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Sabby

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	George

Now perform Pull replication. The result is dependent on the conflict resolver set by the Publisher.

Result for publisher_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Sabby

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	~~George~~ Sabby

Result for subscriber_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Sabby

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	George

Explanation - Although Publisher had inserted record with Roll no 40 and Name Sabby, it will not propagate to Subscriber's side even after pulling, as the conflict resolver is set to subscriber_wins. So Subscriber will have the record with Roll no 40 with the name George.

Delete

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Suppose Publisher has deleted the record with roll no 30. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Now perform Pull replication. The record which was deleted by the Publisher will also be deleted from Subscriber's database. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Update

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Case 1 - Publisher (pub1) have modified a field and the same field in the same row is modified by the Subscriber (sub1) i.e. pub1 have modified the record with roll no 30 to Mathew whereas sub1 have the modified the same record to John. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ Mathew

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ John

Now perform Pull replication. The result is dependent on the conflict resolver set by the Publisher.

Result for publisher_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Mathew

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~John~~ Mathew

Result for subscriber_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Mathew

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	John

Case 2 - pub1 and sub1 have updated different fields in the same row as follows:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ John

Irrespective of what conflict resolver has been set to, after Pull call from Sub1, the data will be updated at column level as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	John

Explanation - Here conflict resolver does not come into play because Publisher and Subscriber have updated different fields of the same record.

Case 3

Suppose we have following records after a snapshot is taken at the subscriber's end:

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Sequence of operation performed by Publisher on Roll no (Primary key) are:

1 ► 9 updated rollno 1 to 9
9 ► 1 updated rollno 9 to 1
8 ► 9 updated rollno 8 to 9
7 ► 8 updated rollno 7 to 8
6 ► 7 updated rollno 6 to 7
5 ► 6 updated rollno 5 to 6
4 ► 5 updated rollno 4 to 5
3 ► 4 updated rollno 3 to 4
2 ► 3 updated rollno 2 to 3
1 ► 2 updated rollno 1 to 2

Operation done at subscriber's end is:

8 ► 10 updated rollno 8 to 10

Tables after updation will be:

Student Table – pub1

Roll no	Name
~~1~~ .. ~~9~~ .. ~~1~~ .. 2	Adam
~~2~~ .. 3	Eve
~~3~~ .. 4	Santa
~~4~~ .. 5	Michael
~~5~~ .. 6	Lina
~~6~~ .. 7	Sabby
~~7~~ .. 8	George
~~8~~ .. 9	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
~~8~~ .. 10	Jordan

Now perform Pull replication. (Assume that conflict resolver has been set to publisher_wins). The result is shown below:

Student Table – pub1

Roll no	Name
2	Adam
3	Eve
4	Santa
5	Michael
6	Lina
7	Sabby
8	George
9	Jordan

Student Table – sub1

Roll no	Name
~~1~~ .. 2	Adam
2	Eve
~~3~~ .. 4	Santa
~~4~~ .. 5	Michael
~~5~~ .. 6	Lina
~~6~~ .. 7	Sabby
~~7~~ .. 8	George
9	Jordan
3	Eve

Explanation

Record with roll no 1 on Subscriber's end was updated to 2 irrespective of what conflict resolver has been set to, because no operation is performed on this record at Subscriber's end. Record with roll no 2 which was originally in Subscriber's side will be deleted as we can't have two records with same primary key in a table. Further record with roll no 2 on publishers was also updated to roll no 3. When Subscriber try to pull this change from Publisher to Subscriber, a new record with roll no 3 will be created at Subscriber's end as the record with roll no 2 was deleted by the Subscriber earlier.

Record with roll no 8 was updated by both Publisher and Subscriber. After Pull call, it will be changed to roll no 9, since conflict resolver is set to publisher_wins. All other Publisher changes will be pulled normally.

Push replication cases

A complete process for using the Replicator is illustrated in the form of an Example in Appendix D.

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

After taking the snapshot, data can be changed by the Publisher as well as Subscriber by a number of ways. Some of the possibilities are discussed below:

Insert

Case 1 - Suppose that a record with roll no 40 is inserted by the Subscriber. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Now perform Push replication. The new record which was inserted by the Subscriber will be pushed to Publisher's database also. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Tom

Case 2 - Insert case with conflict resolver

Tables after Snapshot

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Operations performed

Publisher's side - a record with Roll no 40 and Name *Sabby* is inserted.

Subscriber's side - a record with Roll no 40 and Name *George* is inserted.

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Sabby

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	George

Now perform Push replication. The result is dependent on the conflict resolver set by the Publisher.

Result for subscriber_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	~~Sabby~~ George

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	George

Result for publisher_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	Sabby

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa
40	George

Explanation - Although Subscriber had inserted record with Roll no 40 and Name George, it will not propagate to Publisher's side even after pushing, as the conflict resolver is set to publisher_wins. So Publisher will have the record with Roll no 40 with the name Sabby itself.

Delete

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Suppose Subscriber has deleted the record with roll no 30. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Now perform Push replication. The record which was deleted by the Subscriber will also be deleted from Publisher's database. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Update

Suppose that there is a Publisher (pub1) and a Subscriber (sub1). A snapshot is taken. Both the Publisher and Subscriber have the same data as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	Santa

Case 1 - Publisher (pub1) have modified a field and the same field in the same row is modified by the Subscriber (sub1) i.e. pub1 have modified the record with roll no 30 to Mathew whereas sub1 have the modified the same record to John. The situation is shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ Mathew

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ John

Now perform Push replication. The result is dependent on the conflict resolver set by the Publisher.

Result for subscriber_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	~~Mathew~~ John

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	John

Result for publisher_wins

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
30	Mathew

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	John

Case 2 - pub1 and sub1 have updated different fields in the same row as follows:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
~~30~~ 40	Santa

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	~~Santa~~ John

Irrespective of what conflict resolver has been set to, after Push call from Sub1, the data will be updated at column level as shown below:

Student Table – pub1

Roll no	Name
10	Adam
20	Eve
40	~~Santa~~ John

Student Table – sub1

Roll no	Name
10	Adam
20	Eve
30	John

Explanation - Here conflict resolver does not come into play because Publisher and Subscriber have updated different fields of the same record.

Case 3

Suppose we have following records after a snapshot is taken at the subscriber's end:

Student Table – pub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
8	Jordan

Sequence of operation performed by Publisher on Roll no (Primary key) are:

1 ► 9 updated rollno 1 to 9
9 ► 1 updated rollno 9 to 1
8 ► 9 updated rollno 8 to 9
7 ► 8 updated rollno 7 to 8
6 ► 7 updated rollno 6 to 7
5 ► 6 updated rollno 5 to 6
4 ► 5 updated rollno 4 to 5
3 ► 4 updated rollno 3 to 4
2 ► 3 updated rollno 2 to 3
1 ► 2 updated rollno 1 to 2

Operation done at subscriber's end is:
8 ► 10 updated rollno 8 to 10

Tables after updation will be:

Student Table – pub1

Roll no	Name
~~1~~ .. ~~9~~ .. ~~1~~ .. 2	Adam
~~2~~ .. 3	Eve
~~3~~ .. 4	Santa
~~4~~ .. 5	Michael
~~5~~ .. 6	Lina
~~6~~ .. 7	Sabby
~~7~~ .. 8	George
~~8~~ .. 9	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
~~8~~ .. 10	Jordan

Now perform Push replication. The result is dependent on the conflict resolver set by the Publisher.

Result for subscriber_wins

Student Table – pub1

Roll no	Name
2	Adam
3	Eve
4	Santa
5	Michael
6	Lina
7	Sabby
8	George
10	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
10	Jordan

Explanation

Record with roll no 8 was updated by both Publisher and Subscriber. Upon Push call, as the conflict resolver is set to subscriber_wins, the record will be updated to roll no 10 on both sides. All other changes are done at Publisher's end and will not be propagated to Subscriber.

Result for publisher_wins

Student Table – pub1

Roll no	Name
2	Adam
3	Eve
4	Santa
5	Michael
6	Lina
7	Sabby
8	George
9	Jordan

Student Table – sub1

Roll no	Name
1	Adam
2	Eve
3	Santa
4	Michael
5	Lina
6	Sabby
7	George
10	Jordan

Explanation

Record with roll no 8 was updated by both Publisher and Subscriber. Upon Push call, as the conflict resolver is set to publisher_wins, Subscriber change will not be pushed to Publisher. All the other changes are done at Publisher's end and will not be propagated to Subscriber.

Scheduling

Scheduling is the process of assigning tasks to a set of resources. It is an important concept in many areas such as computing, production processes or airlines.

It has been a key concept in multitasking and multiprocessing operating system design, and in real-time operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler.

In the replication process, scheduling can be very handy as it automates the process of replication. Scheduling reduces the overhead of requesting the replication process time and again. A user can add a schedule once and remain free from sending a replication request to publisher again. Replicator will automatically replicate data on that scheduled time.

A schedule for any replication operation is a specification of start time (the time when replication process is supposed to start) and repetition time (duration after which replication operations should be repeated).

We can schedule all the replication operations like

- **Synchronization**
- **Snapshot**
- **Push**
- **Pull**

In Daffodil Replicator, scheduling permits two modes of replication- **Real Time** and **Non Real Time** .Both these modes provide users a certain level of flexibility in performing replication operations.

REAL TIME:

In **Real Time** mode of replication, replication process specified in schedule will be performed continuously without any time delay. As soon as a user commits the records after performing any database related operation, it gets reflected on both the sides i.e. Publisher's as well as Subscriber's, irrespective of the location.(from which side ,the modifications have been done). All the specified operations will be performed on bi-directional basis.

Replicator will continue to replicate the data unless / until any of the servers (Publisher or Subscriber) shuts down or there is a connection –timeout.

Real time replication can be extremely beneficial in Time critical application where up-to-date data/information is required. Some examples of these types of applications are airline reservation system, missile management system and traffic control system.

<u>NON-REALTIME</u>:

In Non-Real Time mode of replication, whatever schedule is added by user, replication process specified in schedule will be performed at the scheduled time only. This kind of replication requires a user to specify a fixed time limit after which the replication process will take place. This will give a user the flexibility of performing replication operations on desired time.

In this kind of replication, there can be a time when any or both the sides might not have the up-to-date data.

In Non Real Time mode, following options are available for the user.

- **Yearly**

  This option allows the replication process to take place on yearly basis.
- **Monthly**

  This option allows the replication process to take place on monthly basis.
- **Daily**

  This option allows the replication process to take place day on daily basis.
- **Hourly**

  This option allows the replication process to take place on hourly basis.

*User has to specify **counter** for schedule type means after how much (counter value)*(Recurrence type):-time replication operation must be repeated.*

<u>Some points to note regarding Scheduling based Replication:</u>

1) Whenever sub server is started i.e. replication server instance is made, all scheduled operations itself get started.

2)If either publisher's server name or publisher port no. specified at the time of adding schedule is changed later on ,daffodil replicator provides facility for editing this information.

As soon as user edits this information, replication operation starts itself according to your schedule.

3) Only one schedule can be added for one subscriber.

Application Programming Interface (API)

Add Schedule:

Only after creating subscription or getting an existing subscription instance, client (subscriber) can add schedule using the following syntax.

sub.addSchedule (“schedule Name”,”subscription Name”,” schedule Type”, “Publication Server Name”, “publication Port No”, “Schedule Type”,“Replication Type, “start Date Time”, “Schedule Counter”)

For example:

sub.addSchedule("sch1", "Sub1","nonrealtime", “computer1”, "3001","hour","Synchronize",”2005-03-18 14:52:00.000”, 1);
Or

sub.addSchedule("sch1", "Sub1","realtime",“computer1” , "3001","","pull", null, 0);

Edit Schedule:

If Publisher is running on different port or system**,** schedule needs to be edited. User can edit using following Syntax:

Sub.editSchedule (“scheduleName”,”subscriptionName”, “New Publication Server Name”, “New publication Port No”)

For example:

Sub.editSchedule ("sch1", "Sub1", “computer2”, "3003”);

Drop Schedule:

If user wants to remove a schedule, he can remove using following syntax:

Sub.removeSchedule (“scheduleName”,”subscriptionName”)

For example:

sub.dropSchedule ("sch1", "Sub1");

Update Publisher And Subscriber

While developing any application, due importance should be given to end user requirements .But the requirements of the users may not be know to the developers always. So an application should have enough flexibility to get modified whenever necessary.

As we know, while working with database technologies, making changes in the structure of the database is no big deal. So, Daffodil Replicator is providing the option of updating the already created publisher and its corresponding subscribers. Now a user can add new tables in the existing Publication or drop tables from the Publisher as per his/her requirements. The Subscriber can also update his subscription corresponding to the publication. New added tables are created and added to subscription itself and dropped tables are also dropped from subscription (not from database).

Functionality supported:

- User can add tables in publication which are not already published (no need of publishing again)

  For example:

If a publisher has two tables, named 'table1' and 'table2' in his publication and he/she wants to add 'table3' in publication, he don't need to Unpublish and publishing again. He can add new tables in publications.

- User can drop tables from publication which are published (no need of publishing again)

For example:

If a publisher has 'table1' and 'table2' in his publication and he/she wants to drop 'table1' from publication later on, he/she don't need to Unpublish and publishing again. The user can drop tables from publication on desire.

- Subscriber can update corresponding subscription (no need of subscribing again).

For Example:

If corresponding Publication has been changed, Subscriber can simply get updated his subscription no need to unsubscribe it.

NOTE:

It is the responsibility of subscriber to get updated its subscription as soon as corresponding publication is updated.

API:

- **If a publisher "pub1 want to add tables, just write**

  pub1.addTableToPublication (String [ ] newTableList, String [ ] filterClauses);

  Where

  newTableList: it is the list of tables which user want to add in the publication

  FilterClauses: it is the list of filter clauses corresponding to newTableList in same sequence as of newTableList

  NOTE:-if user does not want to set any filter he can specify null for that table

  For ex:

  pub1.addTableToPublication (new String [ ] {"newTable1","newTable2"}, new String [ ] {"col1 is not null", null});

- **If a publisher "pub1" want to drop a table, he can use:**

  pub1.dropTableFromPublication (String [ ] dropTableList);

  Where

  dropTableList: it is the list of the tables to drop.

- **Now as publisher is updated, subscriber must get his subscription updated.**

  sub1.updateSubscription ();

Backward Integration

The provision for backward integration is provided to help users upgrade to the latest Replicator versions. The newer versions include new system tables and modifications or enhancements in existing system tables. Installing the latest version from scratch might be an arduous job for users as it will involve re-defining of Publisher and Subscriber data sources and take the snapshot again.

Replicator zip file contains 'UpdateVersion.bat' batch file. Running batch file creates the additional system tables and modifies the existing tables.

Summary

Daffodil Replicator is an API-framework, which is used in a network of distributed data sources where contents - defined by Publications and Subscriptions - must be kept. The Publisher and the Subscriber data source contain the Publication and Subscription meta-data that describes the information needed to be synchronized.

When the contents of one data source differ from the contents of another data source in the synchronization network, the Replicator API calls *pull ()* to pull changed data from Publisher to Subscriber, *push ()* to push changed data from Subscriber to Publisher, and *synchronize ()* to merge the Publisher as well as Subscriber changes to both the ends, which ensures that these data sources remain synchronized. The Replicator API also supports the creation of complete copies of Publisher's Publications to Subscriber's Subscriptions - called **snapshots** - that are propagated across the synchronization network.

Java applications interact directly with Daffodil Replicator and third party data sources through JDBC to manipulate database contents. Replicator ensures that selective changes made to data sources by Java applications in a distributed - but synchronized - network are propagated to the sites, which are configured to publish or subscribe the data source content.

In short, the Replicator API framework:

- Figures out as to what has changed amongst selected data sources.
- Provides a channel to facilitate transmission of those changes - or complete downloads of data - amongst designated Publisher and Subscriber sites.

In this way, Daffodil Replicator ensures data consistency and integrity across the synchronization network.

Appendix A: Data type Mappings

Standard JDBC	Java Native	Daffodil DB	SQL Server	Oracle	Postgre Sql	Cloudscape(Derby)	DB2
TINYINT	byte	tinyint	tinyint	number	int2	smallint	smallint
SMALLINT	short	smallint	smallint	number	int2	smallint	smallint
INTEGER	int	integer /int	int	number	numeric	integer	integer
BIGINT	long	bigint /long	bigint	number	int8	bigint	bigint
FLOAT	double	float	float	number	numeric	float	double
REAL	float	real	real	number	float4	real	real
DOUBLE	double	double precision	real	number	float8	double	double
NUMERIC	java.math.BigDecimal	numeric	numeric	float	numeric	numeric	decimal
DECIMAL	java.math.BigDecimal	decimal	decimal /dec	float	numeric	decimal	decimal
CHAR	java.lang.String	char	char /character	char	character	national char/char	clob/char
VARCHAR	java.lang.String	varchar/ char varying	varchar	varchar	text	national char varying/ blob /clob/ varchar	varchar
LONGVARCHAR	java.lang.String	long varchar	text	long	text	long varchar	clob/ long varchar
DATE	java.sql.Date	date	datetime	date	date	date	date
TIME	java.sql.Time	time	datetime	date	time	time	time
TIMESTAMP	java.sql.Timestamp	timestamp	datetime	date	timestamp	timestamp	timestamp

Appendix B: Error Messages

This section contains a list of possible error messages along with their codes that Replicator could return to an application. The table below displays error messages, error codes, a description of what can occur and a possible work around.

Errors related to Replication Server

REP001=Replication-Server not started : {0}.
Generally Replicator throws an exception if the port is already in use and Registry could not get the request from a specified port.

REP002=Data Source not found.
A data source requires database URL, driver, user and password. Replicator throws an exception if data source is not set.

REP003= Invalid Driver name or URL.
Replicator throws above exception if you have specified invalid URL, driver or unauthorized user name.

REP004= Invalid IP Address or System name.
Throws an exception when Replication Server is started with an invalid host name/IP address or when Subscription sets invalid remote server name/IP address. 'Local host' can not be specified the as the address of local or remote server.

REP005=Specified driver {0} not found.
Replicator throws above exception if the specified driver is not found.

REP006=General Error {0}

REP007=Invalid URL or Invalid user/password.
Throws an exception when Replication Server is started with an invalid URL, driver, unauthorized user name and password.

REP008=please edit the batch files, pass the arguments: pubserver and subserver for respective batch files.

Errors related to Creating Publication

REP011=Primary Key is not defined in table {0}.
The table to be published must have a primary key. Replicator throws an exception if the Publication includes a table which does not have a primary key.

REP012=No Tables in Publication {0}. At least one table should be specified.
Replicator throws an exception, if no table is specified in the Publication.

REP013=Can not specify the same Table Name {0} more than once in the one Publication {1}.
Replicator throws an exception, if the same table is included in a Publication more than once.

REP014=Publication with the name {0} already exists.
There exists a Publication with the same name in the data source.

REP015=Table ordering is incorrect due to FOREIGN KEY constraints. Table {0} must be specified before the table {1}.

REP016=Invalid Conflict Resolver. Conflict Resolver Can be only Publication-wins or Subscriber-wins.
Conflict Resolver can either be “publication_ wins” or “subscriber_wins” only, else the replicator will throw an exception.

REP017=Table {0} does not exist in the data source.
Replicator throws an exception, if the specified table does not exist in the data source.

REP018=Table exists in more than one catalog/schema. Ambiguous Table Name {0}.
When a Publication is created with two tables with the same name in a different schema, the table name must be specified with {<schemaName>.<tableName>} i.e. schema Name dot table Name. If such a table is specified with out a schema name, it causes ambiguity and throws an exception.

REP019=Could not set the filter clause : {0}
Replicator throws an exception with an appropriate message, if an error occurs while setting filter clause to a table.

REP020=Could not publish the Publication {0}: {1}
Replicator throws an exception with an appropriate message, if an error occurs while publishing the Publication.

REP0201=Could not publish because parent table {0} of table {1} is not included in publication.

REP0202=Table {0} do not publish due to {1}
Replicator throws an exception with an appropriate message, if an error occurs while publishing the Publication.

REP0203=Problem occur in publish process for publication {0} due to {1}
Replicator throws an exception with an appropriate message, if an error occurs while

publishing the Publication.

REP0204=Please enter the remove Cycle Table Names in proper format. Use "-" as separator between two table names.

REP0205=Tables :{0} are having cycle in relation. Please give a relation to be suppressed.

REP0206=Table {0} given in remove Cycle Table Names is not included in publication tables.

REP999=Unable to create table {0}: {1}.
Replicator throws the above exception when Table that you are creating is already exist in the database.

Errors related to Creating Subscription

REP021=Subscription {0} has not been subscribed.
Replicator throws an exception with an appropriate message, if an error occurs while subscribing the Subscription.

REP022=Subscription {0} is already subscribed to Publication {1}.
One Subscription can connect only to a single Publication; else the Replicator throws an exception.

REP023=Subscription with the name {0} already exists.
Two Subscriptions with the same name cannot be created. Replicator throws an exception, if there exists a Subscription with the same name in the data source.

REP024=Subscription {0} can not be subscribed, table {1} already exists on the client side with different structure.
Replicator throws an exception, if a Subscription is subscribed to a Publication with a table that already exists at the client side but with a different structure.

REP025=Error subscribing Subscription {0} : {1}.
Replicator throws an exception with an appropriate message, if an error occurs while subscribing with a Publication.

REP026=Subscription {0} can not be subscribed as Scale of a data Type on Publisher side is not between {0} and {1}.
Replicator throws this exception when the publisher data type scale is more than the supported data type scale length of subscriber database.

REP027=Subscription {0} can not be subscribed as table {1} has 'BIGINT' column which is not supported by Database SQL dialect 1.

Errors related to Snapshot and Synchronization

REP050=Unknown Internal Exception thrown, take snapshot again.
Some Internal error occurred due to wrong entry in shadow table, thereby preventing a proper synchronization. It is suggested to re-take a Snapshot and then try again.

REP051=Other record for same commonId {0} in shadow table {1} not found.
In case of update, there are two records with same id in shadow table. If one of them is not found, Daffodil Replicator will throw an exception.

REP052=Publication is locked by another user.
If more than one Subcription attempts to synchronize data with the Publication simultaneously, then Replicator throws an exception of this kind.

REP053=Problem in getting snapShot for Subscription {0} due to -- {1}.
This error message is due to SQL exception.

REP054=Problem in synchronizing data for Subscription {0} due to -- {1}.
This error message is due to SQL exception.

REP055=Problem in getting snapShot for Publication {0} due to -- {1}.
This error message is due to SQL exception.

REP056=Problem in synchronizing data for Publication {0} due to -- {1}.
This error message is due to SQL exception.

REP057=Problem in synchronizing data due to -- {0}.
This error message is due to SQL exception.

REP058=Snapshot can not be done as Subscription {0} does not exist in the database.
Replicator throws this exception if the specified Subscription was not created.

REP059=Subscription {0} can not be synchronized as it does not exist in the database.
Replicator throws this exception if the specified Subscription was not created.

Errors related to Socket, XML and Zip files

REP081=Can not parse the xml file at specified path.
An xml file is created while subscribing a Subscription, snapshot and synchronization. An exception is thrown, if there is any non-alphanumeric character in the table or column name.
Remark: Table and Column Name with non-alphanumeric characters are not supported in this version.

REP082=Can not find the xml file at specified path.
Replicator throws an exception, if problem occurs while reading temporary xml or zip files from a specified path.

REP083=Socket write Error.
Replicator throws this exception, if some problem occurs while writing the data from the file to a socket.

REP084=Socket read Error.
Replicator throws an exception, if some problem occurs while reading the data from the socket.

REP085=Can not create the xml file at specified path.
Replicator throws an exception, if some problem occurs while creating temporary xml or zip files from the specified path.

REP086=IO Exception: {0}
This error message is related to network.

REP087=Can not create the xml file due to -- {0}.
Replicator throws this exception with an appropriate message, if some problem occurs while creating the xml file .

Errors related to Push and Pull replication

REP0105=Problem in push replication for Subscription {0} due to -- {1}.

REP0106=Problem in push replication data due to -- {0}.

REP0107=Subscription {0} can not be pushed as it does not exist in the database. Replicator throws this exception if the specified Subscription was not created in the Subscriber's database.

REP0151=Problem in pull replication for Subscription {0} due to -- {1}.

REP0152=Problem in pull replication data due to -- {0}.

REP0153=Subscription {0} can not be pulled as it does not exist in the database. Replicator throws this exception if the specified Subscription was not created in the Subscriber's database.

Above mentioned Push and Pull replication exceptions occur in case of any internal exception like the exception that occurs while making an xml file or any SQLException while performing DML and DQL operations.

Error Related for writing into transaction log file

REP351=Error in writing data into log file due to {0}.
Replicator throws this exception with an appropriate message, if some problem occurs while writing data into transaction log file.

Errors related to Scheduling

REP201=More than one schedule can not be added for Subscription {0}.

Replicator throws this exception when you try to add more than one schedule for the same subscription.

REP202=Schedule time cannot be less than current time.

Replicator throws this exception when you try to add a schedule for a specified time and time is less than the current time of machine.

REP203=Can not drop the Schedule {0}, because it does not exist.

Replicator throws this exception when you try to drop a schedule which is not added to any subscriber.

REP204=Can not drop the Schedule {0} due to -- {1}.

Replicator throws this exception while drop a schedule, when Replicator gets a SQLException from a database.

REP205=Schedule {0} can not be added for Subscription {1} due to {2}.

REP206=Problem in Schedule {0} due to -- {1}.

REP207=Please select schedule to be started.

Replicator throws this exception when you do not select a schedule name while starting or editing it.

REP208=Please select schedule to be dropped.

REP209=Schedule could not be started as Schedule {0} does not exist for Subscription {1}.

REP210=No Schedule exists for subscription {0}.

Replicator throws this exception when you are going to drop a schedule, but the schedule does not exist for any subscription.

REP211=Please enter date in valid format.

Replicator throws this exception when you enter an invalid date format for any database.

REP212=Replication operation type is blank or not specified.

Replicator throws this exception when you do not specify any Replication operation type (snapshot, Synchronize, Push or Pull) while adding a schedule for Subscription.

REP213=Schedule Recurrence type is blank or not specified.

Replicator throws this exception when you do not specify any Recurrence type (year, month, Day or Hour) while adding a schedule for Subscription.

REP214=Schedule time is blank or not specified.

Replicator throws this exception when you do not specify any Schedule time while running or adding a schedule for Subscription.

REP215=Schedule name is blank or not specified.

Replicator throws this exception when you do not specify any Schedule Name while adding a schedule for Subscription.

REP216=Please specify the valid replication operation type.

Replicator throws this exception when you specify an invalid Replication operation type. The only valid Replication operation type is Snapshot, Synchronize, Pull, and Push.

REP217=Please specify the (positive) counter value after which schedule is to be started.

Replicator throws this exception when you specify any negative counter value for a schedule.

REP218=Please specify the valid schedule recurrence type.

Replicator throws this exception when you specify an invalid schedule recurrence type. The only valid schedule recurrence type is year, month, Day or Hour.

REP219=Problem in editing the schedule {0} for subscription {1}.

Replicator throws this exception when Replicator gets a SQLException while editing a schedule.

REP220=Please specify only (positive) Server Port No.

REP221=Can not edit the schedule {0}, because it does not exist.

Replicator throws this exception when you are going to edit a schedule, but the schedule does not exist for any subscription.

REP222=Schedule type is blank or not specified.

Replicator throws this exception when you do not specify any Schedule type while running or adding a schedule for Subscription.

REP223=Please specify the valid schedule type.

Replicator throws this exception when you specify an invalid schedule type. The only valid schedule type is Real-Time and Non Real-Time.

REP300=Invalid path {0}

Replicator throws this exception when Replicator gets an invalid replication home path or the replication home Directories does not exist on the specified path.

Errors related to updating of Publisher & Subscriber

REP310=Table {0} is already published.

Replicator throws this exception when you try to publish a table which is already published with same publication.

REP311=Problem in modifying Publication {0} due to -- {1}.

Replicator throws this exception when Replicator gets a SQLException while modifying the publication.

REP312=Problem in modifying Subscription {0} due to -- {1}.

REP313=Table {0} does not exists in publication {1}.

Replicator throws this exception when you try to drop a table from publication but the table does not exist in the publication.

REP314=Table {0} is the only published table, cannot drop table, unpublished the Publisher {1}.

Replicator throws this exception when you try to drop a table from publication but this is the only one table that is published to this publication, so you cannot drop the table from publication, you can only unpublished that publication.

REP315=Cannot drop all tables from publisher, for this unpublished the publisher {0}

Replicator throws this exception when you try to drop all tables from publisher. you cannot drop all tables from publisher, you can only unpublished that publication.

REP316=No tables are specified which can be added/dropped from publisher.

Replicator throws this exception when you try to drop or add tables in publisher but you do not specify any table for that.

REP317=Table name cannot be blank.

Replicator throws this exception when you do not specify any Table name in Publisher.

REP318=Error Updating subscription {0}: {1}.

Replicator throws this exception when you try to update subscription and Replicator gets any internally exception.

REP319=Please specify either null or any filter clause for all tables.

Replicator throws this exception when you do not specify any filter clause while publishing.

REP320=You cannot specify table Name to 'null' in the table Name array.

REP321=Table {1} must be dropped if you want to drop table {0}.

Replicator throws this exception when you try to drop a parent table which has a child table. So child table should be dropped first and then parent table.

Note: {0} - refers to the specified Subscription or Publication or Table or Schedule
{1} - refers to the reason for the exception

Miscellaneous Errors

REP031= Invalid data type {0}.

Replicator throws this exception when you specify any invalid data type while creating a table.

REP032=Unsupported data type {0}.

Replicator throws this exception when you specify any unsupported data type while creating a table.

REP033=Invalid table name {0}.

Replicator throws this exception when you specify any Invalid table name or schema name for the table.

REP034=Published Table {0} has no Primary Key.

Replicator throws this exception when you try to publish the tables which has no primary key column in the table.

REP035=Publication {0} is already published.

Replicator throws this exception when you try to publish the tables with the same publication name which already exists in the database.

REP036=Publication {0} does not exist.

Replicator throws this exception when you try to use a null publication or the publication does not exist in the data source.

REP037=Subscription {0} does not exist.

Replicator throws this exception when you try to use a null subscription or the subscription does not exist in the data source.

REP038=List passed doesn't contain super Table {0} for table {1}.

REP041=Can not create the publication {0}: {1}

REP042=Can not create the subscription {0}: {1}

This error message is due to any SQL Exception or if subscription is already exists in data source with the same name.

REP043=Can not drop the Publication {0}, because it does not exist.

REP044=Can not drop the Subscription {0}, because it does not exist.

REP045=Can not unpublished the Publication {0}: {1}

This error message is due to SQL Exception.

REP046=Can not unsubscribe the Subscription {0}: {1}

This error message is due to SQL Exception.

REP047=Can not unpublished the Publication, one or more Subscription(s) has subscribed to Publication {0}.

A Publication can be unpublished only if all the Subscriptions that are subscribed to it are unsubscribed.

REP048=Corresponding Publication {1} not found on server side.

REP091=SQL-Server does not support local variables for text, ntext and image data types.

Tables with these data types can not be published. This is an SQL specific exception.

REP092=Publication Name is blank or not specified.

REP093=Subscription Name is blank or not specified.

REP094=Port Name is blank or not specified.

REP095=Remote Replication Server Name can not be blank.

REP101=Error getting meta-data information {0}
Replicator throws an exception if some problem occurs while getting the meta information from the corresponding data source.

REP102={0} Database is not supported in this version.

Replicator throws an exception of this kind, if the specified database is not compatible with the current Replicator version

REP103=Problem in unsubscribe -- {0}
This error message is due to SQL Exception.

REP104=Problem in getting Subscription {0} due to -- {1}.
This error message is due to SQL Exception.

Appendix C: Daffodil Replicator Library

Daffodil Replicator Library – JDBC Driver Jars

DATABASE NAME	JDBC DRIVER JARS
Daffodil DB Network Edition	DaffodilDB_Client.jar
Daffodil DB Embedded Edition	DaffodilDB_Client.jar, DaffodilDB_Common.jar
Oracle	classes12.jar
SQL Server	JSQLConnect.jar
	msbase.jar, mssqlserver.jar, msutil.jar
Derby	db2jcc.jar
PostgreSQL	pg80b1.308.jdbc3.jar
DB2	db2jcc.jar
Sybase	Jconn2d.jar
Firebird	Firebirdsql-full.jar

Daffodil Replicator Library – Driver & URL

DATABASE NAME	DRIVER	URL
Daffodil DB Network Edition	in.co.daffodil.db.rmi.RmiDaffodilDBDriver	jdbc:daffodilDB://<DaffodilDB server name>:3456/<database name>;create=true
Daffodil DB Embedded Edition	in.co.daffodil.db.jdbc.DaffodilDBDriver	jdbc:daffodilDB_embedded:<database name>;create=true
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@<Oracle server name>:1521:<database name>
SQL Server	com.jnetdirect.jsql.JSQLDriver	jdbc:JSQLConnect://<SQL Server server name>:1433/database=<database name>/lastUpdateCount=true
	com.microsoft.jdbc.sqlserver.SQLServerDriver	jdbc:microsoft:sqlserver://<SQL Server server name>: 1433;DatabaseName=<database name>
Derby	com.ibm.db2.jcc.DB2Driver	jdbc:db2j:net:// <server name>:1527/<database name>;retrieveMessagesFromServerOnGetMessage=true;deferPrepares=true;create=true
PostgreSQL	org.postgresql.Driver	jdbc:postgresql://<PostgreSQL server name>:5432/<database name>
DB2	com.ibm.db2.jcc.DB2Driver	jdbc:db2://<DB2 server name>::50000/<database name>
Sybase	com.sybase.jdbc2.jdbc.sybDriver	Jdbc:sybase:Tds:host:port/database
Firebird	Org.firebirdsql.jdbc.FBDriver	Jdbc:firebirdsql:<system Name>/3050:<database path>

Appendix D: Working with Replicator API

The Replicator provides a *ReplicationServer* Class to create Publications and Subscriptions.

Common Setup

An instance of ReplicationServer Class is created with *getInstance () method* which requires the port number and the host name or IP address at which you want to start a replication server. The host name can not be 'local host'.

Example:
An instance rs of ReplicationServer Class is created which is started at port no 3001.

_ReplicationServer rs = ReplicationServer.getInstance (3001,"192.168.0.211")

An instance of ReplicationServer Class must be created for each database. Server side (Publication side) Replication Server can be used to create Publications and client side (Subscription side) Replication server can be used to create Subscriptions. Before creating Publication or Subscription you must set the data source for each replication server. There is one to one mapping between replication server and data source i.e. only one data source for each replication server can exist. A data source comprises of database URL, driver, user name and password and can be set with the replication server with *setDataSource () method* as shown below:

rs.setDataSource (driver, URL, user, password)

Example:
For setting Daffodil DB (Network Edition) as a data source, *setDataSource () method* can be called as follows:

rs.setDataSource ("in.co.daffodil.db.rmi.RmiDaffodilDBDriver","jdbc: daffodilDB: //localhost:3456/server;create=true","rep","rep")

Server Setup

Instances for Publication can be created with Server Side Replication Server.

Replication Server API provides *createPublication () method* to create Publication.

Example:

_Publication pub = rs. createPublication ("newPublication", new String [] {"Table01", Table02", Table03",Table04"});

This method requires the Publication name and the tables to be published as parameters and returns an instance of Publication class. Publication name must be unique in the data source. This method throws a *RepException*, if there are no tables specified or if second argument is passed as null. If two tables are specified with the primary and foreign key relationship, primary table needs to be specified before the foreign table, otherwise an exception is thrown.

Types of *RepExceptions* are described in detail in Appendix B.

While creating the Publication, you can also specify ***Row Filtering***.

Row Filtering can be set with the Publication through *setFilter () method*. This method requires Table Name and the filter condition. Table Name must be from the tables with which Publication is created and filter-condition must be a valid Boolean condition.

Example:

pub.setFilter ("Table01","Col01 > 100 ");

In the above example, a filter clause is set to the table “Table01” such that only those rows that have a value of col01 greater than 100 are allowed to be published.
(Note: - It is assumed that col01 is a column of Table01 of data type integer.)

The **Conflict Resolver** can be set with *setConflictResolver () method*. This method requires only one parameter, which must be either *publisher_wins* or *subscriber_wins*. By default, parameter for conflict resolver is *publisher_wins*.

Example:

The following code snippet set conflict-resolver to *subscriber_wins*.

pub.setConflictResolver (_Publication.subscriber_wins);

After making the Publication in the data source, it can be published using *publish () method*. While publishing the Publication, it creates several entries into replication system tables. This method throws *RepException* in case of error while publishing.

Example:

Publication name *newPublication* can be published by:

pub.publish ();

publication can be modified i.e **Update Publicaion** later on to add more tables o drop tables using:

addTableToPublication (String [ ] newTableList, String [ ] filterClauses)

dropTableFromPublication (String [ ] dropTableList)

A Publication can be unpublished using un*publish () method*. While using this method users shall keep in view that a Publication can be unpublished only if it does not have any corresponding Subscriptions. All corresponding Subscriptions must be unsubscribed before a particular Publication is unpublished.

Example:

pub.unpublish ();

Client Setup

Instances for Subscription can be created with Client Side Replication Server.

ReplicationServer API provides *createSubscription () method* to create Subscription.

Example:

_Subscription sub = rs. createSubscription ("newSubscription","newPublication");

This method requires the *Subscription* and *Publication* names as parameters and returns an instance of _Subscription interface. Subscription name must be unique in the data source. Each Subscription is subscribed to one Publication and this method requires the Publication name as the argument describing as to which Publication the Subscription is subscribing.

_Subscription provides the *setRemoteServerPortNo ()* and *setRemoteServerUrl ()* methods to set remote ReplicationServer URL and Port no on which the ReplicationServer with Publication named "*newPublication*" is started.

Example:

Subscription instance sub sets the remote server URL and address as follows:

sub.setRemoteServerPortNo (3001);
sub.setRemoteServerUrl ("192.168.0.211");

After making the Subscription in the data source, it can be subscribed with *subscribe () method*. While subscribing the Subscription, it creates several entries into replication system tables. This method throws *RepException* in case of error while subscribing.

Example:

Subscription name *newSubscription* can be subscribed to *newPublication* as

sub.subscribe ();

Once Publication and Subscription are created, user can use them repeatedly by getting their reference from _ReplicationServer reference as:

_Publication pub = rs.getPublication ("newPublication");
_Subscription sub = rs.getSubscription ("newSubscription");

In case if corresponding publication has been modified, users can update their subscriptions using:

updateSubscription()

After creating Subscription or getting an existing Subscription instance, client (Subscriber) can get the **snapshot** using the following syntax:

sub.getSnapShot();

If Publisher and/or Subscriber have made changes after taking the snapshot, the data can be updated by the following ways:

synchronize the data (updates will be merged on both the sides) using

sub.synchronize();

OR **Pull** only the Publisher changes to Subscriber's database using:

sub.pull();

OR **Push** only the Subscriber changes to Publisher's database using:

sub.push();

Users can **add schedule**(Non Realtime or Realtime) for Replication Processes using:

addSchedule ("schedule Name"," subscription Name"," schedule Type", "Publication Server Name", "publication Port No", "Schedule Type"," Replication Type, "start Date Time", "Schedule Counter")

Users can **edit schedule** to modify the remote server name and port using:

editSchedule ("scheduleName","subscriptionName", "New Publication Server Name", "New publication Port No")

Users can **remove schedule** using:

removeSchedule ("scheduleName","subscriptionName")

Users can unsubscribe a particular Subscription with the help of *unsubscribe ()* method. As a result of a call to this method, all tables corresponding to that Subscription will be dropped.

Example:

sub.unsubscribe();

Appendix E: Sample JDBC Programs

The following set of sample programs does the function of creating and connecting data sources, creating and manipulating Publications and Subscriptions through JDBC API. In all the programs, the data sources are assumed to be Daffodil DB.

Program to connect, create and populate the database

```
import java.sql.*;
import java.util.Arrays;

/**
 * Program to create and populate the database.
 */

public class Database {

  private String driver;
  private String url;
  private String user;
  private String password;
  private Connection con;

  public Database(String driver0, String url0, String user0, String password0) throws
      Exception {
    driver = driver0;
    url = url0;
    user = user0;
    password = password0;
    connect();
  }

  public void initialize() throws Exception {

    try {
      createTables();
      populateTables();
    }
    catch (Exception ex) {
    }
  }

  private void connect() throws Exception {
    // Load the database driver and get the Connection to the database
    Class.forName(driver);
    con = DriverManager.getConnection(url, user, password);
  }

  private void createTables() throws Exception {
    Statement stt = con.createStatement();
    stt.execute(
       " create table country ( cid int primary key , cname varchar(20)) ");
    stt.execute(" create table state ( sid int primary key , sname varchar(20) , cid int )");
    stt.close();
  }

  private void populateTables() throws Exception {
    Statement stt = con.createStatement();
    stt.execute(" insert into country values ( 1 , 'India')");
    stt.execute(" insert into country values ( 2 , 'USA')");
    stt.execute(" insert into country values ( 3 , 'Japan')");
    stt.execute(" insert into country values ( 4 , 'China')");
    stt.execute(" insert into state values   ( 1 , 'Haryana' , 1)");
    stt.execute(" insert into state values   ( 2 , 'Maxico'  , 2)");
    stt.execute(" insert into state values   ( 3 , 'Tokyo'   , 3)");
    stt.execute(" insert into state values   ( 4 , 'Paris'   , 4)");
    stt.close();
  }

  public void InsertRecordInCountry( int cid ) throws Exception {
    Statement stt = con.createStatement();
    stt.execute(" insert into country values ( "+cid+" , 'India')");
    stt.close();
  }

  public void InsertRecordsInState( int sid) throws Exception {
    Statement stt = con.createStatement();
    stt.execute(" insert into state values   ( "+sid+" , 'Haryana' , 1)");
    stt.close();
  }

  public void displayRows(String tableName) throws SQLException {
    ResultSet rs = con.createStatement().executeQuery("Select * from "+tableName);
    ResultSetMetaData metaData = rs.getMetaData();
    int columnCount = metaData.getColumnCount();
    Object[] displayColumn = new Object[columnCount];
    for (int i = 1; i <= columnCount; i++)
      displayColumn[i - 1] = metaData.getColumnName(i);
    System.out.println(Arrays.asList(displayColumn));
    while (rs.next()) {
      Object[] columnValues = new Object[columnCount];
      for (int i = 1; i <= columnCount; i++)
        columnValues[i - 1] = rs.getObject(i);
      System.out.println(Arrays.asList(columnValues));
    }
  }

}
```

Program to create and manipulate Publications

```
import com.daffodilwoods.replication.ReplicationServer;
import com.daffodilwoods.replication._ReplicationServer;
import com.daffodilwoods.replication._Publication;
import java.rmi.RemoteException;
import java.io.BufferedReader;
import java.io.*;

public class SampleServer {

  // The following constants specifies driver name , user-name and password
  // for Daffodil DB as Database Server on Publication(Server side).
  // manav = System Name
  private static final String DAFFODILDB_DRIVER =
"in.co.daffodil.db.rmi.RmiDaffodilDBDriver",
                              DAFFODILDB_URL =
"jdbc:daffodilDB://localhost:3456/serverDB;create=true",
                              DAFFODILDB_USER = "daffodil",
                              DAFFODILDB_PASSWORD = "daffodil";

  // Constant to define the Publication name
  private static final String PUB_NAME   = "Pub01";
  // Constant to define the port number of Replication Server (Server side)
  private static final int PORT_NUMBER   = 3001;
  // Constant to define the System name or IP address of the system
  // on which replication server is running
  private static final String SYS_NAME   = "vikas";

  private static void start() {
    try {

      // Creates country and state table and populates them
      Database database = new Database(DAFFODILDB_DRIVER, DAFFODILDB_URL,
                                  DAFFODILDB_USER, DAFFODILDB_PASSWORD);
      database.initialize();

      // Get an instance of the ReplicationServer by providing the Url, driver,
      // user name and password to connect to the Publisher database named
      // 'serverDB'.
      _ReplicationServer rsServer =
ReplicationServer.getInstance(PORT_NUMBER,SYS_NAME);
      rsServer.setDataSource(DAFFODILDB_DRIVER, DAFFODILDB_URL,
                                  DAFFODILDB_USER, DAFFODILDB_PASSWORD);

      // Gets an instance of _Publication if publication named PUB_NAME exists,
      // returns null otherwise.
      _Publication pub = rsServer.getPublication(PUB_NAME);
      if(pub == null) {
        // Creates the publication named PUB_NAME, and the tables to be published under
        // publication must be specified as an second argument.
        pub = rsServer.createPublication(PUB_NAME,new
String[]{"COUNTRY","STATE"});
        // set the conflict resolver as publisher_wins, for more information,
        // refer to replication pdf mannual.
        pub.setConflictResolver(pub.publisher_wins); // Optional-- by default its set to
publisher_wins
        // Sets filter clause on one or more tables to be published.
        pub.setFilter("COUNTRY"," CID > 0 "); // Optional
        // registers the newly created publication with the replication server.
        pub.publish();
      }

        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("\n Press [Enter] to display the data after synchronization ");
        input.readLine();
        // display data after synchronize been done
        System.out.println(" Data after synchronize at server... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");


        System.out.println("\n Press [Enter] to insert data for Pull on Server Side ");
        input.readLine();
        // inserting records in Country and state on Server side
        database.InsertRecordInCountry(8);
        database.InsertRecordInCountry(9);
        database.InsertRecordInCountry(10);
        database.InsertRecordsInState(8);
        database.InsertRecordsInState(9);
        database.InsertRecordsInState(10);


        // display data at client
        System.out.println(" Data after insertion at Serevr... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");


        System.out.println("\n Press [Enter] to display the data after Pull ");
        input.readLine();
        // display data after Pull been done
        System.out.println(" Data after Pull at server... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");
        System.out.println("\n Non RealTime Schedule has been added and started for
Push by Subscriber");
        System.out.println(" Please Wait for atleast 2 minutes............... ");

        System.out.println(
            "\n Press [Enter] to display the data after Push");
        input.readLine();
        // display data after push been done
        System.out.println(" Data after Push(done when schedule is running) at server... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");


        System.exit(00);
    }
    catch (Exception ex) {
      System.out.println("Caught exception: " + ex);
      ex.printStackTrace();
    }
  }

  public static void main(String[] args) {
    (new SampleServer()).start();
  }

}
```

Sample program to create and manipulate Subscriptions

```
import com.daffodilwoods.replication.ReplicationServer;
import com.daffodilwoods.replication._ReplicationServer;
import com.daffodilwoods.replication._Subscription;
import java.rmi.RemoteException;
import java.sql.Timestamp;
import java.io.BufferedReader;
import java.io.*;

public class SampleClient {

  // The following constants specifies driver name , user-name and password
  // for Daffodil DB as Database Server on Subscriber(Server side).
 // manav = System Name
  private static final String DAFFODILDB_DRIVER =
"in.co.daffodil.db.rmi.RmiDaffodilDBDriver",
                              DAFFODILDB_URL =
"jdbc:daffodilDB://vikas:3456/clientDB;create=true",
                              DAFFODILDB_USER = "daffodil",
                              DAFFODILDB_PASSWORD = "daffodil";

  // Constants to define the Subscription name
  private static final String SUB_NAME   = "Sub01";
  // Constant to define the port number of Replication Server (Client side)
  private static final int PORT_NUMBER   = 3002;
  // Constant to define the System name or IP address of the system
  // on which replication server is running
  private static final String SYS_NAME   = "vikas";

  // Constant to define the Publication name to which the subscription will be subscribed
  private static final String PUB_NAME   = "Pub01";
  // Constant to define the system name on which Publication resides
  private static final String REMOTER_SERVER_NAME   = "vikas";
  // Constant to define the port nuber on which publisher side replication server is running
  private static final int REMOTER_PORT_NO   = 3001;

  private static void start() {
    try {
      // Get an instance of the ReplicationServer by providing the Url, driver,
      // user name and password to connect to the Subscriber database 'clientDB'.
      _ReplicationServer rsClient =
ReplicationServer.getInstance(PORT_NUMBER,SYS_NAME);
      System.out.println("rsClient:"+rsClient);
      rsClient.setDataSource(DAFFODILDB_DRIVER, DAFFODILDB_URL,
                                 DAFFODILDB_USER, DAFFODILDB_PASSWORD);
      // Gets an instance of _Subscription if subscription named SUB_NAME exists,
      // returns null otherwise
      _Subscription sub = rsClient.getSubscription(SUB_NAME);
      if(sub == null) {
        // Creates the Subscription named SUB_NAME which requires the Publication
```

```
          // name as a second argument
          sub = rsClient.createSubscription(SUB_NAME,PUB_NAME);
          // sets the remote server name
          sub.setRemoteServerUrl(REMOTER_SERVER_NAME);
          // sets the remote replication server port number
          sub.setRemoteServerPortNo(REMOTER_PORT_NO);
          // registers the newly created subscription with the replication server
          sub.subscribe();
         }
         else {
            // sets the remote server name
          sub.setRemoteServerUrl(REMOTER_SERVER_NAME);
          // sets the remote replication server port number
          sub.setRemoteServerPortNo(REMOTER_PORT_NO);
          // registers the newly created subscription with the replication server
            }

      // For information related to snapshot process refer to user_guide manual sent along
with this file
      sub.getSnapShot();

      Database database = new Database(DAFFODILDB_DRIVER, DAFFODILDB_URL,
                                  DAFFODILDB_USER, DAFFODILDB_PASSWORD);

      // display data at client
      System.out.println(" Data after Snapshot operation at client... ");
      System.out.println("[ Country Table ]");
      database.displayRows("Country");
      System.out.println("[ State Table ]");
      database.displayRows("State");


      // inserting records in Country and state on Client side
      database.InsertRecordInCountry(5);
      database.InsertRecordInCountry(6);
      database.InsertRecordInCountry(7);
      database.InsertRecordsInState(5);
      database.InsertRecordsInState(6);
      database.InsertRecordsInState(7);

      // display data at client
      System.out.println(" Data after insertion at client... ");
      System.out.println("[ Country Table ]");
      database.displayRows("Country");
      System.out.println("[ State Table ]");
      database.displayRows("State");

      // For information related to synchronize process refer to user_guide manual sent
along with this file
      sub.synchronize();

      // display data at client
      System.out.println(" Data after synchronize at client... ");
```

```
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");


        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("\n Press [Enter] for Pull data ");
        input.readLine();

        sub.pull();

        // display data after Pull been done
        System.out.println(" Data after Pull at Client... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");



        // inserting records in Country and state on Client side
        database.InsertRecordInCountry(11);
        database.InsertRecordInCountry(12);
        database.InsertRecordInCountry(13);
        database.InsertRecordsInState(11);
        database.InsertRecordsInState(12);
        database.InsertRecordsInState(13);

        // display data at client
        System.out.println(" Data after insertion at client... ");
        System.out.println("[ Country Table ]");
        database.displayRows("Country");
        System.out.println("[ State Table ]");
        database.displayRows("State");


        //adding schedule which repeats itself after 1 hour,continuous schedule i.e realtime
schedule can also be added
        //sub.addSchedule("sch1", "sub1", "realtime", SYS_NAME, "3001", "", "push", null,0);
        Timestamp tm1= new Timestamp(System.currentTimeMillis());
        Timestamp tm = new
Timestamp(tm1.getYear(),tm1.getMonth(),tm1.getDate(),tm1.getHours(),tm1.getMinutes(
)+2,0,0);

        sub.addSchedule("sch1", SUB_NAME, "nonrealtime", SYS_NAME, "3001", "hour",
                  "Push", tm, 1);
        System.out.println("\n Non RealTime Schedule has been added and started for Push
");
        System.out.println(" Please Wait for atleast 2 minutes............... ");


        Thread.currentThread().sleep(218000);
```

```
      // display data at client
      System.out.println(" Data after Push at client...(No Change on Client Side) ");
      System.out.println("[ Country Table ]");
      database.displayRows("Country");
      System.out.println("[ State Table ]");
      database.displayRows("State");

      //edit the schedule in case publisher's url or port no is changed
//      sub.editSchedule("sch1", SUB_NAME, SYS_NAME, "3001");

      //To remove the schedule created
      sub.removeSchedule("sch1",SUB_NAME);

      System.exit(00);
    }
    catch (Exception ex) {
      System.out.println("Caught exception: " + ex);
      ex.printStackTrace();
    }
  }

  public static void main(String[] args) {
    (new SampleClient()).start();
  }

}
```

Appendix F: Database Pre-requisites

Daffodil Replicator handles the replication for Daffodil DB, Oracle and SQL Server, DB2, Sybase, PostgreSql, Derby (Cloudscape) and Firebird Databases. There are some pre-requisites for successful homogeneous/heterogeneous replication.

Homogeneous Database Replication

- **Daffodil DB to Daffodil DB**

  There is no restriction when Daffodil DB is used as both Publisher and Subscriber.

- **SQL Server to SQL Server**

  There is no restriction when SQL Server is used as both Publisher and Subscriber.

- **Oracle to Oracle**

  When Oracle is used as both Publisher and Subscriber, the same user must exist in both the Oracle databases.

- **PostgreSql to PostgreSql**

  There is no restriction when PostgreSQL is used as both Publisher and Subscriber.

- **Derby to Derby**

  There is no restriction when Derby is used as both Publisher and Subscriber.

- **DB2 to DB2**

  There is no restriction when DB2 is used as both Publisher and Subscriber.

- **Sybase to Sybase**

  There is no restriction when Sybase is used as both Publisher and Subscriber.

- **Firebird to Firebird**

  There is no restriction when Firebird is used as both Publisher and Subscriber.

  Note: Replicator does not support Firebird replication with heterogeneous databases.

Heterogeneous Databases Replication

- **Daffodil DB to SQL Server**

  When Daffodil DB database server is used as Publisher and SQL Server is used as Subscriber, then there should be a user name in SQL Server with a name same as that of the schema name in the Daffodil DB database to be replicated. The default schema for Daffodil DB database is ‘users’. There should be a SQL Server user ‘users’.

- **Oracle to SQL Server**

  When Oracle database is used as Publisher and SQL Server Database is used as Subscriber, there must be a user name in the SQL Server whose name is same as schema name in Oracle Publisher Database.

- **Derby/DB2 to Oracle/PostgreSql/SQL Server**

  When Derby / DB2 is used as a Publisher and Oracle / PostgreSql / SQL Server is used as Subscriber, there must be a schema in the Oracle / PostgreSql / SQL Server Databases whose name is same as that of the schema in Derby / DB2 database.

- **PostgreSql to Oracle/SQL Server**

  When PostgreSql is used as a Publisher and Oracle / SQL Server is used as Subscriber, there must be a user/schema in the Oracle / SQL Server Database whose name is same as that of the schema in PostgreSql database.

- **Daffodil DB to Oracle**

  When Daffodil DB is used as a Publisher and Oracle is used as Subscriber, there must be a user in the Oracle Database whose name is same as that of the schema in Daffodil DB database.

- **SQL Server to Oracle**

  When SQL Server is used as Publisher and Oracle database is used as Subscriber, then there must be a schema in the Oracle Database whose name is same as SQL Server schema name.

- **Oracle/Daffodil DB/SQL Server/Derby/DB2 to PostgreSql**

  When Oracle / Daffodil DB / SQL Server / Derby / DB2 is used as a Publisher and PostgreSql is used as a Subscriber, there must be a schema in the PostgreSql Database whose name is same as that of the schema in Oracle / Daffodil DB / SQL Server / Derby / DB2 databases.

- **Oracle/Daffodil DB/SQL Server/PostgreSql to Derby/DB2**

  There is no restriction when Oracle / Daffodil DB / SQL Server / PostgreSql database is replicated on Derby / DB2 database. Daffodil Replicator creates replicated tables in a schema that is same as user name in Oracle / Daffodil DB / SQL Server / PostgreSql Publisher Database.

- **SQL Server/ Oracle/ Derby/ PostgreSql/DB2 to Daffodil DB**

  There is no restriction when SQL Server / Oracle / Derby / PostgreSql / DB2 Database is replicated with Daffodil DB database Server. Daffodil Replicator creates replicated tables in a schema that is same as user/schema name of SQL Server / Oracle / Derby / PostgreSql / DB2 Publisher database.

- **Derby to DB2**

  When Derby is used as Publisher and DB2 is used as Subscriber, then there must be a schema in the Derby database whose name is same as DB2 schema name.

- **DB2 to Derby**

  There is no restriction when DB2 database is replicated on Derby database. Daffodil Replicator creates replicated tables in a schema that is same as user name in DB2 Publisher Database.

- **Sybase to Oracle**

  When Sybase is used as a Publisher and Oracle is used as Subscriber, there must be a user in the Oracle Database whose name is same as that of the schema in Sybase database.

- **SQL Server/ Oracle/ Derby/ PostgreSql/DB2 to Sybase**

  There is no restriction when SQL Server / Oracle / Derby / PostgreSql / DB2 Database are replicated with Sybase database Server. Daffodil Replicator creates replicated tables in a schema that is same as user/schema name of SQL Server / Oracle / Derby / PostgreSql / DB2 Publisher database.

Appendix G: Known Issues and Limitations

Daffodil Replicator has the following issues and limitations:

1. **Issue with PostgreSql**
   By editing the *config* file of PostgreSql database, you need to specify the IP address of the system where Replication Server is running.

2. **Unsupported Microsoft SQL Server data types**
   uniqueidentifier and *timestamp* data types in SQL Server are not supported, as these data types need external functions for usage. Daffodil Replicator will not be able to perform operations on tables having such columns.

3. **Precision in heterogeneous databases**
   While dealing with heterogeneous databases, precision of data types shall be considered. If the precision of publishing RDBMS is more than that of the subscribing RDBMS, then the following problems may occur:
   (a) The users will not be able to subscribe such Publications.
   (b) Subscription will result in inconsistent data.

4. **Primary key constraint**
   While dealing with heterogeneous databases, if users are trying to subscribe a table having binary / varbinary data type column as the primary key from an external RDBMS (SQL Server) to Daffodil DB, this operation is not permitted in Daffodil Replicator as Daffodil DB does not allow creation of tables having binary / varbinary data type column as the primary key.

5. **Limitations regarding identity Columns in SQL Server**
   While using Daffodil Replicator with an SQL Server database as Publisher and if a table has an identity column as its primary key, in such cases users of Daffodil Replicator will not be able to synchronize such tables but snapshot operations can be performed.

6. **System Tables**
   Manual updation in system tables affect the functioning of the replicator. It may prevent synchronization of data, or show an error message or may give an undesired result.

9. **Auto Increment Column**
   As JDBC does not provide the required meta data information to append the auto increment columns in table structure, the auto increment columns when subscribed from the publisher does not support this property.

10. **Backward Compatibility**
    For users working with older versions of Daffodil Replicator (upto 1.6) need to use the ReplicatorUpto1_6.jar found in the Daffodil Replicator binary for backward compatibility with features of newer versions.

End Note

Although this manual reflects the most current information possible, you should read the ***Daffodil Replicator Release Notes*** from time to time for latest information and updates on Daffodil Replicator.

Release Notes are available at: http://www.daffodildb.com/daffodil-release-notes.html

For more detailed information about Replicator Console, read *Daffodil Replicator Console Guide*.

Sign Up for Support

If you have successfully installed and started working with Daffodil Replicator, please remember to sign up for the benefits you are entitled to as a Daffodil Replicator customer.

For free support, be a part of our online developer community at Daffodil Developer Forum

For buying support packages, please visit: http://www.daffodildb.com/support-overview.html

For more information regarding support, write to us at: support@daffodildb.com

We Need Feedback!

If you spot a typographical error in the ***Daffodil Replicator Developer's Guide***, or if you have thought of a way to make this manual better, we would love to hear from you!

Please submit a report in Bugzilla http://www.daffodildb.com/bugzilla/index.cgi OR write to us at: feedback@daffodildb.com

License Notice