SymmetricDS 2 User Guide

v2.5

Table of Contents

Preface

SymmetricDS is an open-source, web-enabled, database independent, data synchronization software application. It uses web and database technologies to replicate tables between relational databases in near real time. The software was designed to scale for a large number of databases, work across low-bandwidth connections, and withstand periods of network outages.

This User Guide introduces SymmetricDS and its uses for data synchronization. It is intended for users who want to be quickly familiarized with the software, configure it, and use its many features. This version of the guide was generated on 2012-05-30 at 21:23:10.

Chapter 1. Introduction

This User Guide will introduce both basic and advanced concepts in the configuration of SymmetricDS. By the end of this chapter, you will have a better understanding of SymmetricDS' capabilities, and many of its basic concepts.

1.1. What is SymmetricDS?

SymmetricDS is an asynchronous data replication software package that supports multiple subscribers and bi-directional synchronization. It uses web and database technologies to replicate tables between relational databases, in near real time if desired. The software was designed to scale for a large number of databases, work across low-bandwidth connections, and withstand periods of network outage. The software can be installed as a standalone process, as a web application in a Java application server, or it can be embedded into another Java application.

A single installation of SymmetricDS attached to a target database is called a *node*. A node is initialized by a properties file and is configured by inserting configuration data into a series of database tables. It then creates database triggers on the application tables to be synchronized so that database events are captured for delivery to other SymmetricDS nodes.

In most databases, the transaction id is also captured by the database triggers so that the insert, update, and delete events can be replicated transactionally via the transport layer to other nodes. The transport layer is typically a CSV protocol over HTTP or HTTPS.

SymmetricDS supports synchronization across different database platforms through the concept of *Database Dialects*. A Database Dialect is an abstraction layer that SymmetricDS uses to insulate the main synchronization logic from database-specific implementation details.

In addition to synchronization, SymmetricDS is also capable of performing fairly complex *transformations* of data as the synchronization data is loaded into a target database. The transformations can be used to merge source data, make multiple copies of source data across multiple target tables, set defaults in the target tables, etc. The types of transformation can also be extended to create even more custom transformations.

SymmetricDS is extendable through extension points. Extension points are custom, reusable Java code that are configured via XML. Extension points hook into key points in the life-cycle of a synchronization to allow custom behavior to be injected. Extension points allow custom behavior such as: publishing data to other sources, transforming data, and taking different actions based on the content or status of a synchronization.

1.2. Background

The idea of SymmetricDS was born from a real-world need. Several of the original developers were, several years ago, implementing a commercial Point of Sale (POS) system for a large retailer. The development team came to the conclusion that the software available for trickling back transactions to

corporate headquarters (frequently known as the 'central office' or 'general office') did not meet the project needs. The list of project requirements made finding the ideal solution difficult:

- Sending and receiving data with up to 2000 stores during peak holiday loads.
- Supporting one database platform at the store and a different one at the central office.
- Synchronizing some data in one direction, and other data in both directions.
- Filtering out sensitive data and re-routing it to a protected database.
- Preparing the store database with an initial load of data from the central office.

The team ultimately created a custom solution that met the requirements and led to a successful project. From this work came the knowledge and experience that SymmetricDS benefits from today.

1.3. SymmetricDS Features

At a high level, SymmetricDS comes with a number of features that you are likely to need or want when doing data synchronization. A majority of these features were created as a direct result of real-world use of SymmetricDS in production settings.

1.3.1. Notification Schemes

After a change to the database is recorded, the SymmetricDS nodes interested in the change are notified. Change notification is configured to perform either a *push* (trickle-back) or a *pull* (trickle-poll) of data. When several nodes target their changes to a central node, it is efficient to push the changes instead of waiting for the central node to pull from each source node. If the network configuration protects a node with a firewall, a pull configuration could allow the node to receive data changes that might otherwise be blocked using push. The frequency of the change notification is configurable and defaults to once per minute.

1.3.2. Two-Way Table Synchronization

In practice, much of the data in a typical synchronization requires synchronization in just one direction. For example, a retail store sends its sales transactions to a central office, and the central office sends its stock items and pricing to the store. Other data may synchronize in both directions. For example, the retail store sends the central office an inventory document, and the central office updates the document status, which is then sent back to the store. SymmetricDS supports bi-directional or two-way table synchronization and avoids getting into update loops by only recording data changes outside of synchronization.

1.3.3. Data Channels

SymmetricDS supports the concept of *channels* of data. Data synchronization is defined at the table (or

table subset) level, and each managed table can be assigned to a *channel* that helps control the flow of data. A channel is a category of data that can be enabled, prioritized and synchronized independently of other channels. For example, in a retail environment, users may be waiting for inventory documents to update while a promotional sale event updates a large number of items. If processed in order, the item updates would delay the inventory updates even though the data is unrelated. By assigning changes to the item tables to an *item* channel and inventory tables' changes to an *inventory* channel, the changes are processed independently so inventory can get through despite the large amount of item data. Channels are discussed in more detail in Section 3.5, Choosing Data Channels (p. 19).

1.3.4. Transaction Awareness

Many databases provide a unique transaction identifier associated with the rows that are committed together as a transaction. SymmetricDS stores the transaction identifier, along with the data that changed, so it can play back the transaction exactly as it occurred originally. This means the target database maintains the same transactional integrity as its source. Support for transaction identification for supported databases is documented in the appendix of this guide.

1.3.5. Data Filtering and Rerouting

Using SymmetricDS, data can be filtered as it is recorded, extracted, and loaded.

- Data routing is accomplished by assigning a router type to a ROUTER configuration. Routers are responsible for identifying what target nodes captured changes should be delivered to. Custom routers are possible by providing a class implementing `IDataRouter`.

- As data changes are loaded in the target database, a class implementing IDataLoaderFilter can change the data in a column or route it somewhere else. One possible use might be to route credit card data to a secure database and blank it out as it loads into a centralized sales database. The filter can also prevent data from reaching the database altogether, effectively replacing the default data loading process.

- Columns can be excluded from synchronization so they are never recorded when the table is changed. As data changes are loaded into the target database, a class implementing `IColumnFilter` can remove a column altogether from the synchronization. For example, an employee table may be synchronized to a retail store database, but the employee's password is only synchronized on the initial insert.

- As data changes are extracted from the source database, a class implementing the `IExtractorListener` interface is called to filter data or route it somewhere else. By default, SymmetricDS provides a handler that transforms and streams data as CSV. Optionally, an alternate implementation may be provided to take some other action on the extracted data.

1.3.6. HTTP(S) Transport

By default, SymmetricDS uses web-based HTTP or HTTPS in a style called Representation State Transfer (REST). It is lightweight and easy to manage. A series of filters are also provided to enforce authentication and to restrict the number of simultaneous synchronization streams. The `ITransportManager`

interface allows other transports to be implemented.

1.3.7. Remote Management

Administration functions are exposed through Java Management Extensions (JMX) and can be accessed from the Java JConsole or through an application server. Functions include opening registration, reloading data, purging old data, and viewing batches. A number of configuration and runtime properties are available to be viewed as well.

SymmetricDS also provides functionality to send SQL events through the same synchronization mechanism that is used to send data. The data payload can be any SQL statement. The event is processed and acknowledged just like any other event type.

1.4. System Requirements

SymmetricDS is written in Java 5 and requires a Java SE Runtime Environment (JRE) or Java SE Development Kit (JDK) version 5.0 or above.

Any database with trigger technology and a JDBC driver has the potential to run SymmetricDS. The database is abstracted through a *Database Dialect* in order to support specific features of each database. The following Database Dialects have been included with this release:

- MySQL version 5.0.2 and above
- Oracle version 8.1.7 and above
- PostgreSQL version 8.2.5 and above
- Sql Server 2005
- HSQLDB 1.8
- H2 1.x
- Apache Derby 10.3.2.1 and above
- IBM DB2 9.5
- Firebird 2.0 and above

See Appendix C, *Database Notes* (p. 102)for compatibility notes and other details for your specific database.

1.5. What's new in SymmetricDS 2

SymmetricDS 2 builds upon the existing SymmetricDS 1.x software base and incorporates a number of architectural changes and performance improvements. If you are brand new to SymmetricDS, you can

safely skip this section. If you have used SymmetricDS 1.x in the past, this section summarizes the key differences you will encounter when moving to SymmetricDS 2.

The first significant architectural change involves SymmetricDS's use of triggers. In 1.x, triggers capture and record data changes as well as the nodes to which the changes must be applied as row inserts into the DATA_EVENT table. Thus, the number of row-inserts grows linearly with the number of client nodes. This can lead to an obvious performance issue as the number of nodes increases. In addition, the problem is made worse at times due to synchronizing nodes updating the same DATA_EVENT table as part of the batching process while the row-inserts are being created.

In SymmetricDS 2, triggers capture *only* data changes, not the node-specific details. The node-specific row-inserts are replaced with a new routing mechanism that does both the routing and the batching of data on one thread. Thus, the real-time inserts into DATA_EVENT by applications using synchronized tables have been eliminated, and database performance is therefore improved. The database contention on DATA_EVENT has also been eliminated, since the router job is the only thread inserting data into that table. The only other access to the DATA_EVENT table is from selects by synchronizing nodes.

As a result of these changes, we gain the following benefits:

- Synchronizing client nodes will spend less time connected to a server node,
- Applications updating database tables that are being synchronized to a large number of nodes will not degrade in performance as more nodes are added, and
- There should be almost no database contention on the `data_event` table, unlike the possible contention in 1.X.

Because routing no longer takes place in the SymmetricDS database triggers, a new mechanism for routing was needed. In SymmetricDS 1.x, the `node_select` expression was used for specifying the desired data routing. It was a SQL expression that qualified the insert into DATA_EVENT from the SymmetricDS triggers. In SymmetricDS 2 there is a new extension point called the data router. Data routers are configured in the router table with a `router_type` and a `router_expression`. Several different routers have been provided to serve the majority of users' routing needs, but the framework is in place for a SymmetricDS programmer to develop domain- or application-specific routers. See Section 4.6.2, Router (p. 29) for a complete list of provided routers.

Since the routing and capturing of data are now performed with two separate mechanisms, the two concepts have been separated into separate configuration tables in the database, with a join table (TRIGGER_ROUTER) specifying the relationships between routing (ROUTER) and capturing of data (TRIGGER). This solves a long standing issue with some databases which only allow one trigger per table. On those database platforms, we can now route data in multiple directions since we only require one SymmetricDS trigger to capture data. This also helps performance in those scenarios, since we only capture the data once instead of once per routing instance.

As part of the new routing job, we have introduced another new extension point to allow more flexibility in the way data events get batched. A batch is the unit by with captured data is sent and committed on target nodes. In SymmetricDS 2, batching is now configured on the channel configuration table. This provides additional flexibility for batching:

- Batching can have the traditional SymmetricDS 1.x behavior of batching up to a max batch size,

but never breaking on a database transaction boundary.

- Batching can be completely tied to a database transaction. One batch per database transaction.
- Batching can ignore database transactions altogether and always batch based on a max batch size.

Another significant change to note in SymmetricDS 2 is the removal of the incoming and outgoing batch history tables. This change was made because it was found that over 95% of the time the statistics the end user truly wanted to see were those for the most recent synchronization attempt, not to mention that the outgoing batch history table was difficult to query. The most valuable information in the batch history tables, the batch statistics, have been moved over to the batch tables themselves. The statistics in the batch tables now always represent the latest synchronization attempt.

Chapter 2. Hands-on Tutorial

Now that several of the features of SymmetricDS have been discussed, a quick working example of SymmetricDS is in order. This section contains a hands-on tutorial that demonstrates how to synchronize a sample database between two running instances of SymmetricDS. This example models a retail business that has a central office database (called "root") and multiple retail store databases (called "client"). For the tutorial, we will have only one "client", as shown in Figure 2.1.

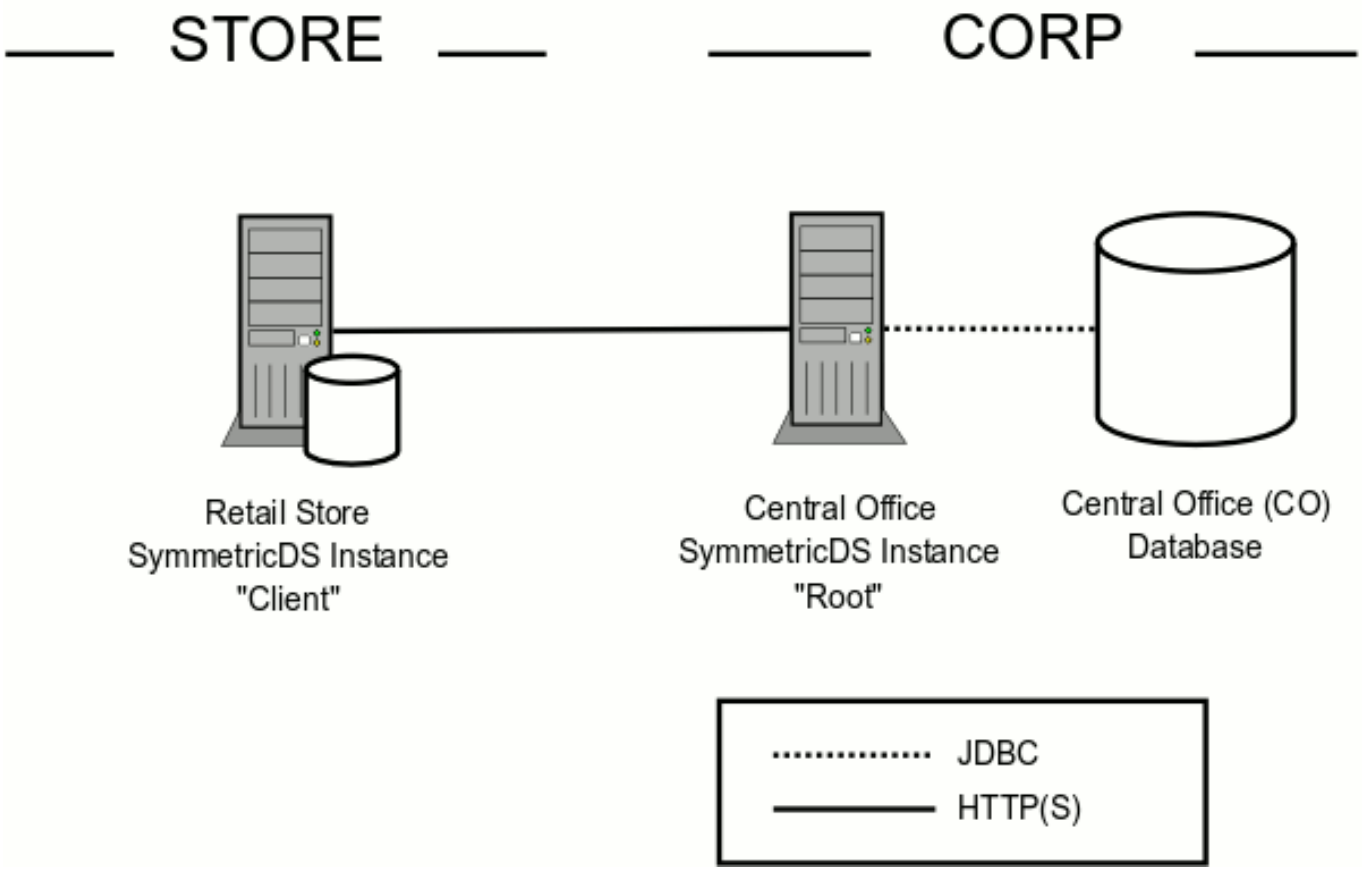


Figure 2.1. Simplified Two Tiered Retail Store Tutorial Example

The root SymmetricDS instance sends changes to the client for item data, such as item number, description, and price. The client SymmetricDS sends changes to the root for sale transaction data, such as time of sale and items sold. The sample configuration specifies synchronization with a pull method for the client to receive data from root, and a push method for the root to receive data from client.

This tutorial will walk you through:

1. Installing instances of SymmetricDS for the tutorial,
2. Creating separate databases for the root and client,
3. Creating sample tables for client and root and sample data for the root,

4. Starting SymmetricDS and registering the client with the root,
5. Sending an initial load to the client,
6. Causing a data push and data pull operation, and
7. Verifying information about the batches that were sent and received.

2.1. Installing SymmetricDS

First, we will install the SymmetricDS software and configure it with your database connection information:

1. Download the symmetric-ds-2.x.x-server.zip file from http://www.symmetricds.org/
2. Unzip the file in any directory you choose. This will create a `symmetric-ds-2.x.x-server` subdirectory, which corresponds to the version you downloaded.
3. Edit the database properties in the following property files for the root (central office) and client (store) nodes:

- `samples/root.properties`
- `samples/client.properties`

4. Set the following properties in *both* files to specify how to connect to the database:

```
# The class name for the JDBC Driver
db.driver=com.mysql.jdbc.Driver

# The JDBC URL used to connect to the database
db.url=jdbc:mysql://localhost/sample

# The user to login as who can create and update tables
db.user=symmetric

# The password for the user to login as
db.password=secret
```

5. Next, set the following property in the `client.properties` file to specify where the root node can be contacted:

```
# The HTTP URL of the root node to contact for registration
registration.url=http://localhost:8080/sync
```

For the tutorial, the client database starts out empty, and the node is not registered. Registration is the process where the node receives its configuration and stores it in its database. The configuration describes which database tables to synchronize and to which nodes. When an unregistered node starts up, it will register with the node specified by the registration URL. The

registration node centrally controls nodes on the network by allowing registration and returning configuration. In this tutorial, the registration node is the root node, which also participates in synchronization with other nodes.

2.2. Creating and Populating Your Databases

Important

You must first create the databases for your root and client nodes using the administration tools provided by your database vendor. Make sure the name of the databases you create match the settings in the properties files.

See Appendix C, *Database Notes* (p. 102) for compatibility with your specific database.

First, create the sample tables in the *root* node database, load the sample data, and load the sample configuration.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.

2. Create the sample tables in the root database by executing the following command:

   ../bin/sym -p root.properties --run-ddl create_sample.xml

   Note that the warning messages from the command are safe to ignore.

3. Next, create the SymmetricDS tables in the root node database. These tables will contain the configuration for synchronization. The following command uses the auto-creation feature to create all the necessary SymmetricDS system tables.

   ../bin/sym -p root.properties --auto-create

4. Finally, load the sample data and configuration into the root node database by executing:

   ../bin/sym -p root.properties --run-sql insert_sample.sql

We have now created the root database tables and populated them with sample data. Next, we create the sample tables in the *client* node database to prepare it for receiving data.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.

2. Create the sample tables in the client database by executing:

   ../bin/sym -p client.properties --run-ddl create_sample.xml

Note that the warning messages from the command are safe to ignore.

Please verify *both* databases by logging in and listing the tables.

1. Find the item tables that sync from root to client: item and item_selling_price.
2. Find the sales tables that sync from client to root: sale_transaction and sale_return_line_item.
3. Find the SymmetricDS system tables, which have a prefix of "sym_".
4. Validate the root item tables have sample data.

2.3. Starting SymmetricDS

Database setup and configuration for the tutorial is now complete. Time to put SymmetricDS into action. We will now start both SymmetricDS nodes and observe the logging output.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Start the root node server by executing:

   ../bin/sym -p root.properties --port 8080 --server

   The root node server starts up and creates all the triggers that were configured by the sample configuration. It listens on port 8080 for synchronization and registration requests.
3. Start the client node server by executing:

   ../bin/sym -p client.properties --port 9090 --server

   The client node server starts up and uses the auto-creation feature to create the SymmetricDS system tables. It begins polling the root node in order to register. Since registration is not yet open, the client node receives an authorization failure (HTTP response of 403).

Tip

If you want to change the port number used by SymmetricDS, you need to also set the sync.url runtime property to match. The default value is:

```
sync.url=http://localhost:8080/sync
```

2.4. Registering a Node

Next, we need to open registration for the client node so that it may receive its initial load of data and so that it may receive and send data from and to the root node. There are several ways to do this. We will use the administration feature on the root node.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.

2. Open registration for the client node server by executing:

   ../bin/sym -p root.properties --open-registration "store,1"

   The registration is now opened for a node group called "store" with an external identifier of "1". This information matches the settings in `client.properties` for the client node. Each node is assigned to a node group and is given an external ID that makes sense for the application. In this tutorial, we have retail stores that run SymmetricDS, so we named our node group "store" and we used numeric identifiers starting with "1". More information about node groups will be covered in the next chapter.

3. Watch the logging output of the client node to see it successfully register with the root node. The client is configured to attempt registration once per minute. Once registered, the root and client are enabled for synchronization!

2.5. Sending an Initial Load

Next, we will send an initial load of data to our store (that is, the client node), again using the root node administration feature.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.

2. Send an initial load of data to the client node server by executing:

   ../bin/sym -p root.properties --reload-node 1

   With this command, the root node queues up an initial load for the client node that will be sent the next time the client performs its pull. The initial load includes data for each table that is configured for synchronization.

3. Watch the logging output of both nodes to see the data transfer. The client is configured to pull data from the root every minute.

2.6. Pulling Data

Next, we will make a change to the item data in the central office (we'll add a new item), and observe the data being pulled down to the store.

1. Open an interactive SQL session with the *root* database.
2. Add a new item for sale:

   insert into item_selling_price (price_id, price) values (55, 0.65);

   insert into item (item_id, price_id, name) values (110000055, 55, 'Soft Drink');

   Once the statements are committed, the data change is captured by SymmetricDS and queued for the client node to pull.
3. Watch the logging output of both nodes to see the data transfer. The client is configured to pull data from the root every minute.
4. Verify that the new data arrives in the client database using another interactive SQL session.

2.7. Pushing Data

We will now simulate a sale at the store and observe how SymmetricDS pushes the sale transaction to the central office.

1. Open an interactive SQL session with the *client* database.
2. Add a new sale to the client database:

   insert into sale_transaction (tran_id, store, workstation, day, seq) values (1000, '1', '3', '2007-11-01', 100);

   insert into sale_return_line_item (tran_id, item_id, price, quantity) values (1000, 110000055, 0.65, 1);

   Once the statements are committed, the data change is captured and queued for the client node to push.
3. Watch the logging output of both nodes to see the data transfer. The client is configured to push data to the root every minute.

2.8. Verifying Outgoing Batches

Now that we have pushed and pulled data, we will demonstrate how you can obtain information about what data has been batched and sent. A batch is used for tracking and sending data changes to nodes. The sending node creates a batch and the receiving node acknowledges it. A batch in error is retried during synchronization attempts, but only after data changes in other channels are allowed to be sent. Channels

are categories assigned to tables for the purpose of independent synchronization and control. Batches for a channel are not created when a batch in the channel is in error status.

1. Open an interactive SQL session with either the root or client database.

2. Verify that the data change you made was captured:

   select * from sym_data order by data_id desc;

   Each row represents a row of data that was changed. The event_type is "I" for insert, "U" for update", or "D" for delete. For insert and update, the captured data values are listed in row_data. For update and delete, the primary key values are listed in pk_data.

3. Verify that the data change was routed to a node, using the data_id from the previous step:

   select * from sym_data_event where data_id = ?;

   When the batched flag is set, the data change is assigned to a batch using a batch_id that is used to track and synchronize the data. Batches are created and assigned during a push or pull synchronization.

4. Verify that the data change was batched, sent, and acknowledged, using the batch_id from the previous step:

   select * from sym_outgoing_batch where batch_id = ?;

   A batch represents a collection of changes to be sent to a node. The batch is created during a push or pull synchronization, when the status is set to "NE" for new. The receiving node acknowledges the batch with a status of "OK" for success or "ER" for error.

Understanding these three tables, along with a fourth table discussed in the next section, is key to diagnosing any synchronizaiton issues you might encounter. As you work with SymmetricDS, either when experimenting or starting to use SymmetricDS on your own data, spend time monitoring these tables to better understand how SymmetricDS works.

2.9. Verifying Incoming Batches

The receiving node keeps track of the batches it acknowledges and records statistics about loading the data. Duplicate batches are skipped by default, but this behavior can be changed with the `incoming.batches.skip.duplicates` runtime property.

1. Open an interactive SQL session with either the root or client database.

2. Verify that the batch was acknowledged, using a batch_id from the previous section:

   select * from sym_incoming_batch where batch_id = ?;

   A batch represents a collection of changes loaded by the node. The sending node that created the batch is recorded. The status is either "OK" for success or "ER" for error.

Chapter 3. Planning an Implementation

In the previous Chapter we presented a high level introduction to some basic concepts in SymmetricDS, some of the high-level features, and a tutorial demonstrating a basic, working example of SymmetricDS in action. This chapter will focus on the key considerations and decisions one must make when planning a SymmetricDS implementation. As needed, basic concepts will be reviewed or introduced throughout this Chapter. By the end of the chapter you should be able to proceed forward and implement your planned design. This Chapter will intentionally avoid discussing the underlying database tables that capture the configuration resulting from your analysis and design process. Implementation of your design, along with discussion of the tables backing each concept, is covered in Chapter 4, *Configuration* (p. 24).

When needed, we will rely on an example of a typical use of SymmetricDS in retail situations. This example retail deployment of SymmetricDS might include many point-of-sale workstations located at stores that may have intermittent network connection to a central location. These workstations might have point-sale-software that uses a local relational database. The database is populated with items, prices and tax information from a centralized database. The point-of-sale software looks up item information from the local database and also saves sale information to the same database. The persisted sales need to be propagated back to the centralized database.

3.1. Identifying Nodes

A *node* is a single instance of SymmetricDS. It can be thought of as a proxy for a database which manages the synchronization of data to and/or from its database. For our example retail application, the following would be SymmetricDS nodes:

- Each point-of-sale workstation.
- The central office database server.

Each node of SymmetricDS can be either embedded in another application, run stand-alone, or even run in the background as a service. If desired, nodes can be clustered to help disperse load if they send and/or receive large volumes of data to or from a large number of nodes.

Individual nodes are easy to identify when planning your implementation. If a database exists in your domain that needs to send or receive data, there needs to be a corresponding SymmetricDS instance (a node) responsible for managing the synchronization for that database.

3.2. Organizing Nodes

Nodes in SymmetricDS are organized into an overall node network, with connections based on what data needs to be synchronized where. The exact organization of your nodes will be very specific to your synchronization goals. As a starting point, lay out your nodes in diagram form and draw connections between nodes to represent cases in which data is to flow in some manner. Think in terms of what data is needed at which node, what data is in common to more than one node, etc. If it is helpful, you could also show data flow into and out of external systems. As you will discover later, SymmetricDS can publish data changes from a node as well using JMS.

Our retail example, as shown in Figure 3.1, represents a tree hierarchy with a single central office node connected by lines to one or more children nodes (the POS workstations). Information flows from the central office node to an individual register and vice versa, but never flows between registers.

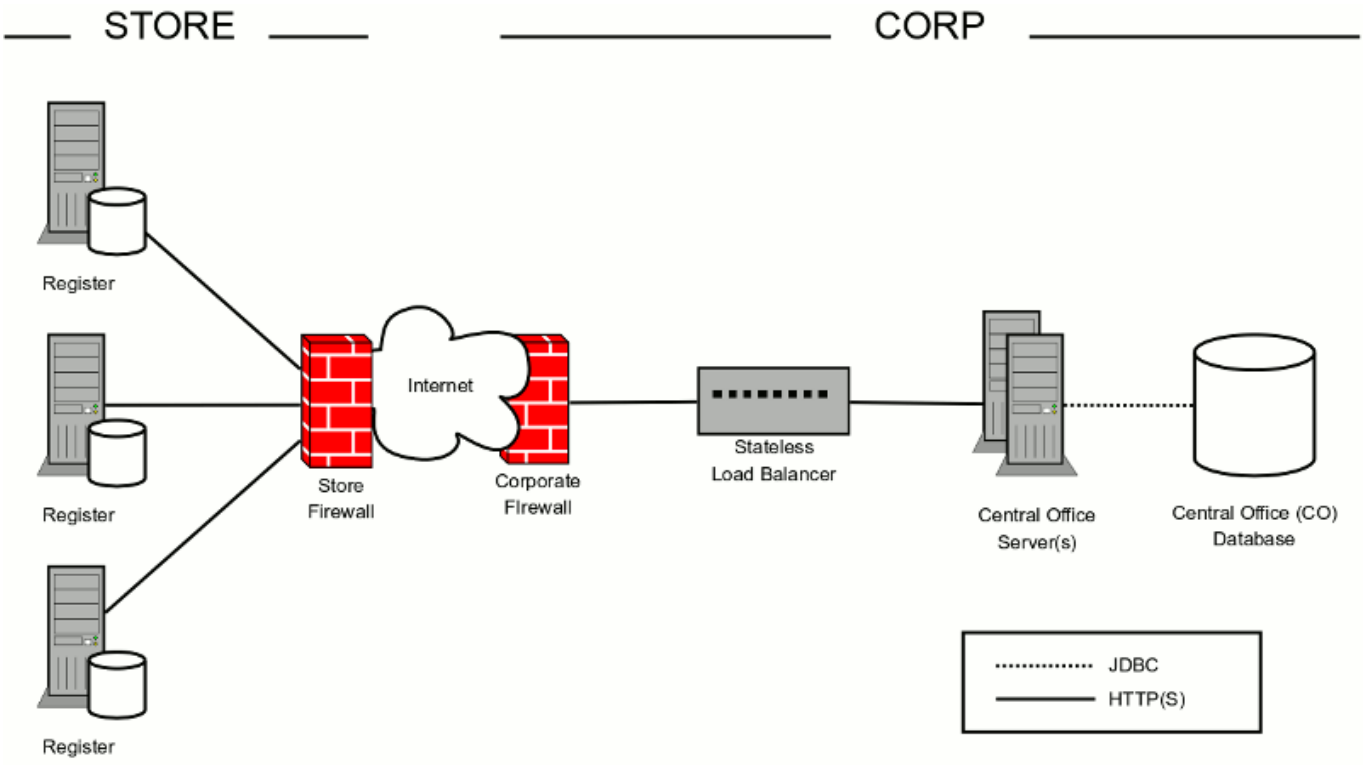


Figure 3.1. Two Tiered Retail Store Deployment Example

More complex organization can also be used. Consider, for example, if the same retail example is expanded to include store *servers* in each store to perform tasks such as opening the store for the day, reconciling registers, assigning employees, etc. One approach to this new configuration would be to create a three-tier hierarchy (see Figure 3.2). The highest tier, the centralized database, connects with each store server's database. The store servers, in turn, communicate with the individual point-of-sale workstations at the store. In this way data from each register could be accumulated at the store server, then sent on to the central office. Similarly, data from the central office can be staged in the store server and then sent on to each register, filtering the register's data based on which register it is.

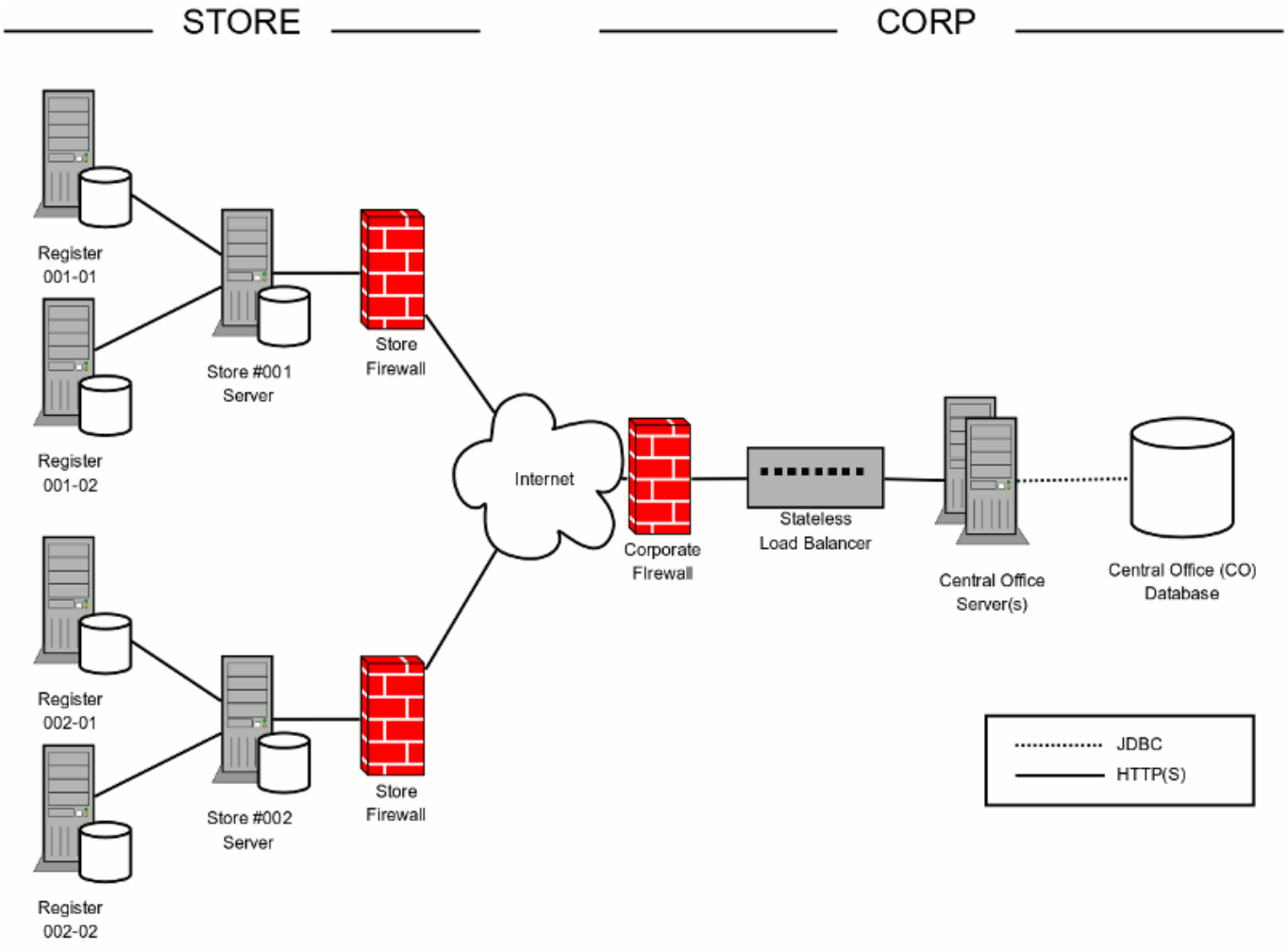


Figure 3.2. Three Tiered, In-Store Server, Retail Store Deployment Example

One final example, show in Figure 3.3, again extending our original two-tier retail use case, would be to organize stores by "region" in the world. This three tier architecture would introduce new regional servers (and corresponding regional databases) which would consolidate information specific to stores the regional server is responsible for. The tiers in this case are therefore the central office server, regional servers, and individual store registers.

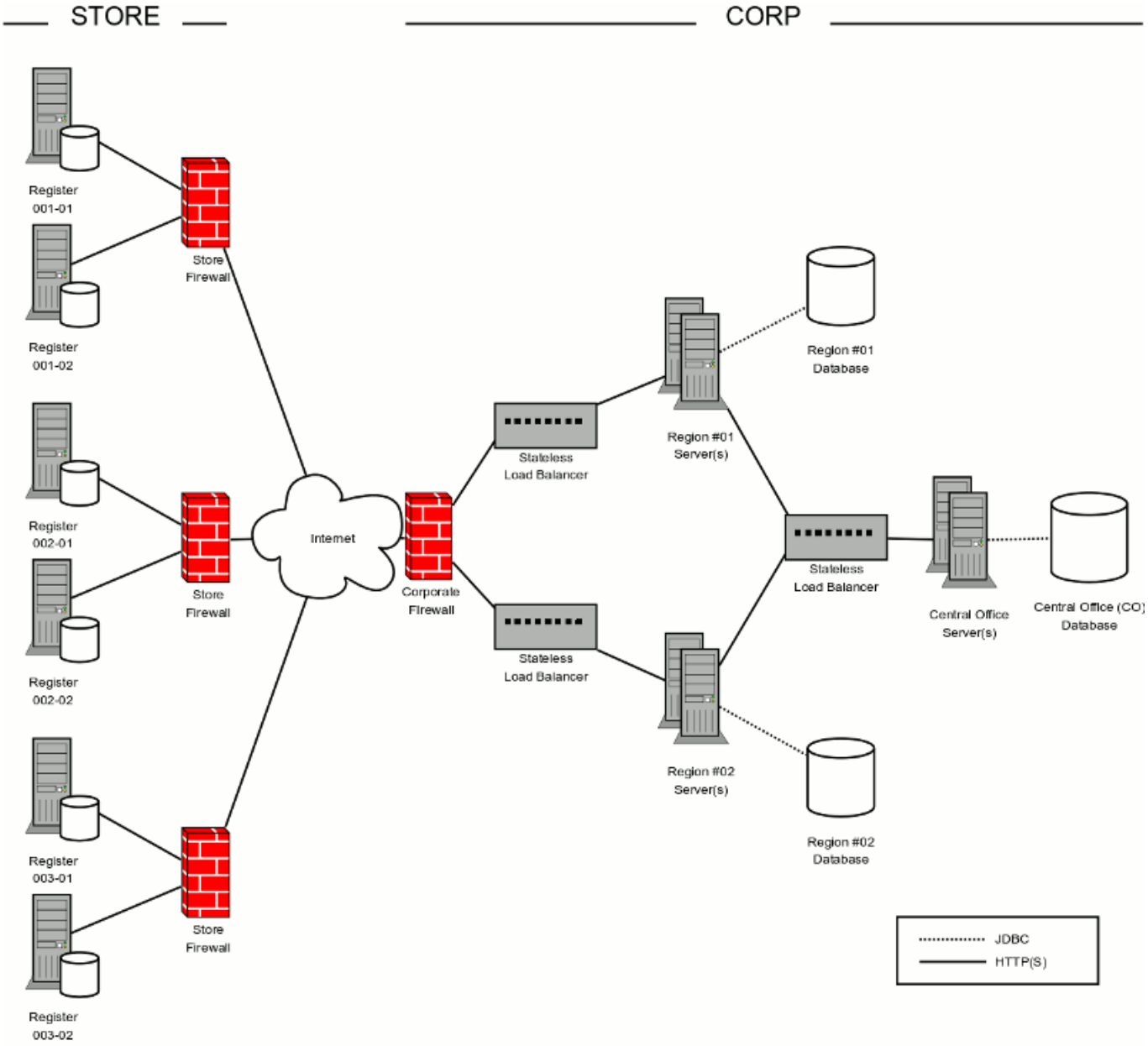


Figure 3.3. Three Tiered, Regional Server, Retail Store Deployment Example

These are just three common examples of how one might organize nodes in SymmetricDS. While the examples above were for the retail industry, the organization, they could apply to a variety of application domains.

3.3. Defining Node Groups

Once the organization of your SymmetricDS nodes has been chosen, you will need to *group* your nodes based on which nodes share common functionality. This is accomplished in SymmetricDS through the concept of a *Node Group*. Frequently, an individual tier in your network will represent one Node Group. Much of SymmetricDS' functionality is specified by Node Group and not an individual node. For

example, when it comes time to decide where to route data captured by SymmetricDS, the routing is configured by *Node Group*.

For the examples above, we might define Node Groups of:

- "workstation", to represent each point-of-sale workstation
- "corp" or "central-office" to represent the centralized node.
- "store" to represent the store server that interacts with store workstations and sends and receives data from a central office server.
- "region" to represent the a regional server that interacts with store workstations and sends and receives data from a central office server.

Considerable thought should be given to how you define the Node Groups. Groups should be created for each set of nodes that synchronize common tables in a similar manner. Also, give your Node Groups meaningful names, as they will appear in many, many places in your implementation of SymmetricDS.

Note that there are other mechanisms in SymmetricDS to route to individual nodes or smaller subsets of nodes within a Node Group, so do not choose Node Groups based on needing only subsets of data at specific nodes. For example, although you could, you would not want to create a Node Group for each store even though different tax rates need to be routed to each store. Each store needs to synchronize the same tables to the same groups, so 'store' would be a good choice for a Node Group.

3.4. Linking Nodes

Now that Node Groups have been chosen, the next step in planning is to document the individual links between Node Groups. These *Node Group Links* establish a source Node Group, a target Node Group, and a *data event action*, namely whether the data changes are *pushed* or *pulled*. The push method causes the source Node Group to connect to the target, while a pull method causes it to wait for the target to connect to it.

For our retail store example, there are two Node Group Links defined. For the first link, the "store" Node Group pushes data to the "corp" central office Node Group. The second defines a "corp" to "store" link as a pull. Thus, the store nodes will periodically pull data from the central office, but when it comes time to send data to the central office a store node will do a push.

3.5. Choosing Data Channels

When SymmetricDS captures data changes in the database, the changes are captured in the order in which they occur. In addition, that order is preserved when synchronizing the data to other nodes. Frequently, however, you will have cases where you have different "types" of data with differing priorities. Some data might, for example, need priority for synchronization despite the normal order of events. For example, in a retail environment, users may be waiting for inventory documents to update while a promotional sale event updates a large number of items.

SymmetricDS supports this by allowing tables being synchronized to be grouped together into *Channels*

of data. A number of controls to the synchronization behavior of SymmetricDS are controlled at the Channel level. For example, Channels provide a processing order when synchronizing, a limit on the amount of data that will be batched together, and isolation from errors in other channels. By categorizing data into channels and assigning them to TRIGGERs, the user gains more control and visibility into the flow of data. In addition, SymmetricDS allows for synchronization to be enabled, suspended, or scheduled by Channels as well. The frequency of synchronization can also be controlled at the channel level.

Choosing Channels is fairly straightforward and can be changed over time, if needed. Think about the differing "types" of data present in your application, the volume of data in the various types, etc. What data is considered must-have and can't be delayed due to a high volume load of another type of data? For example, you might place employee-related data, such as clocking in or out, on one channel, but sales transactions on another. We will define which tables belong to which channels in the next sections.

Important

Be sure that, when defining Channels, all tables related by foreign keys are included in the same channel.

3.6. Defining Data Changes to be Captured and Routed

At this point, you have designed the node-related aspects of your implementation, namely choosing nodes, grouping the nodes based on functionality, defining which node groups send and receive data to which others (and by what method). You have defined data Channels based on the types and priority of data being synchronized. The largest remaining task prior to starting your implementation is to define and document what data changes are to be captured (by defining SymmetricDS *Triggers*), and to decide to which node(s) the data changes are to be *routed* to and under what conditions. We will also, in this section, discuss the concept of an *initial load* of data into a SymmetricDS node.

3.6.1. Defining Triggers

SymmetricDS uses *database triggers* to capture and record changes to be synchronized to other nodes. Based on the configuration you provide, SymmetricDS creates the needed database triggers automatically for you. There is a great deal of flexibility in terms of defining the exact conditions under which a data change is captured. SymmetricDS triggers are defined in a table named TRIGGER. Each trigger you define is for a particular table associated. Each trigger can also specify:

- whether to install a trigger for updates, inserts, and/or deletes
- conditions on which an insert, update, and/or delete fires
- a list of columns that should not be synchronized from this table
- a SQL select statement that can be used to hold data needed for routing (known as External Data)

As you define your triggers, consider which data changes are relevant to your application and which ones

ar not. Consider under what special conditions you might want to route data, as well. For our retail example, we likely want to have triggers defined for updating, inserting, and deleting pricing information in the central office so that the data can be routed down to the stores. Similarly, we need triggers on sales transaction tables such that sales information can be sent back to the central office.

3.6.2. Defining Routers

The triggers that have been defined in the previous section only define *when* data changes are to be captured for synchronization. They do not define *where* the data changes are to be sent to. Routers, plus a mapping between Triggers and Routers (TRIGGER_ROUTER), define the process for determining which nodes receive the data changes.

Before we discuss Routers and Trigger Routers, we should probably take a break and discuss the process SymmetricDS uses to keep track of the changes and routing. As we stated, SymmetricDS relies on auto-created database triggers to capture and record relevant data changes into a table, the DATA table. After the data is captured, a background process chooses the nodes that the data will be synchronized to. This is called *routing* and it is performed by the Routing Job. Note that the Routing Job does not actually send any data. It just organizes and records the decisions on where to send data in a "staging" table called DATA_EVENT and OUTGOING_BATCH.

Now we are ready to discuss Routers. The router itself is what defines the configuration of where to send a data change. Each Router you define can be associated with or assigned to any number of Triggers through a join table that defines the relationship. Routers are defined the SymmetricDS table named ROUTER. For each router you define, you will need to specify:

- the target table on the destination node to route the data
- the source node group and target node group for the nodes to route the data to
- a router *type* and router *expression*
- whether to route updates, inserts, and/or deletes

For now, do not worry about the specific routing types. They will be covered later. For your design simply make notes of the information needed and decisions to determine the list of nodes to route to. You will find later that there is incredible flexibility and functionality available in routers. For example, you will find you can:

- send the changes to all nodes that belong to the target node group defined in the router.
- compare old or new column values to a constant value or the value of a node's identity.
- execute a SQL expression against the database to select nodes to route to. This SQL expression can be passed values of old and new column values.
- execute a Bean Shell expression in order to select nodes to route to. The Bean Shell expression can use the the old and new column values.
- publish data changes directly to a messaging solution instead of transmitting changes to registered

nodes. (This router must be configured manually in XML as an extension point.)

For each of your Triggers, decide which Router matches the behavior needed for that Trigger. These Trigger Router combinations will be used to define a mapping between your Triggers and Routers when you implement your design.

3.6.3. Mapping Triggers to Routers

The mapping between Triggers and Routers, found in the table TRIGGER_ROUTER, defines configuration specific to a particular Trigger and Router combination.

3.6.3.1. Planning Initial Loads

SymmetricDS provides the ability to "load" or "seed" a node's database with specific sets of data from its parent node. This concept is known as an *Initial Load* of data and is used to start off most synchronization scenarios. The Trigger Router mapping defines how initial loads can occur, so now is a good time to plan how your *Initial Loads* will work. Using our retail example, consider a new store being opened. Initially, you would like to pre-populate a store database with all the item, pricing, and tax data for that specific store. This is achieved through an initial load. A part of your planning, be sure to consider which tables, if any, will need to be loaded initially. SymmetricDS can also perform an initial load on a table with just a subset of data. Initial Loads are further discussed in Section 4.6.3.1, Initial Load (p. 35).

3.6.3.2. Circular References and "Ping Back"

When routing data, SymmetricDS by default checks each data change and will not route a data change back to a node if it originated the change to begin with. This prevents the possibility of data changes resulting in an infinite loop of changes under certain circumstances. You may find that, for some reason, you need SymmetricDS to go ahead and send the data back to the originating node - a "ping back". As part of the planning process, consider whether you have a special case for needing ping back. Ping Back control is further discussed in Section 4.6.3.3, Enabling "Ping Back" (p. 37).

3.6.4. Planning for Registering Nodes

Our final step in planning an implementation of SymmetricDS involves deciding how a new node is connected to, or *registered* with a parent node for the first time.

The following are some options on ways you might register nodes:

- The tutorial uses the command line utility to register each individual node.
- A JMX interface provides the same interface that the command line utility does. JMX can be invoked programatically or via a web console.
- Both the utility and the JMX method register a node by inserting into two tables. A script can be written to directly register nodes by directly inserting into the database.
- SymmetricDS can be configured to auto register nodes. This means that any node that asks for a

registration will be given one.

3.7. Planning Data Transformations

SymmetricDS also provides the abilty to *transform* synchronized data instead of simply synchronizing it. Your application might, for example require a particular column in your source data to be mapped to two different target tables with possibly different column names. Or, you might need to "merge" one or more columns of data from two indepdentent tables into one table on the target. Or, you may want to set default column values on a target table based on a particular event on the source database. All of these operations, and many more, can be accomplished using SymmetricDS' transformation capabilities.

As you plan your SymmetricDS implementation, make notes of cases where a data transformation is needed. Include details such as when the transformation might occur (is it only on an insert, or a delete?), which tables or columns play a part, etc. Complete details of all the transformation features, including how to configure a transformation, are discussed in Section 4.8, Transforming Data (p. 38).

Chapter 4. Configuration

Chapter 3 introduced numerous concepts and the analysis and design needed to create an implementation of SymmetricDS. This chapter re-visits each analysis step and documents how to turn a SymmetricDS design into reality through configuration of the various SymmetricDS tables. In addition, several advanced configuration options, not presented previously, will also be covered.

4.1. Node Properties

To get a SymmetricDS node running, it needs to be given an identity and it needs to know how to connect to the database it will be synchronizing. A typical way to specify this is to place properties in the `symmetric.properties` file. When started up, SymmetricDS reads the configuration and state from the database. If the configuration tables are missing, they are created automatically (auto creation can be disabled). Basic configuration is described by inserting into the following tables (the complete data model is defined in Appendix A, *Data Model* (p. 71)).

- NODE_GROUP - specifies the tiers that exist in a SymmetricDS network
- NODE_GROUP_LINK - links two node groups together for synchronization
- CHANNEL - grouping and priority of synchronizations
- TRIGGER - specifies tables, channels, and conditions for which changes in the database should be captured
- ROUTER - specifies the routers defined for synchronization, along with other routing details
- TRIGGER_ROUTER - provides mappings of routers and triggers

During start up, triggers are verified against the database, and database triggers are installed on tables that require data changes to be captured. The Route, Pull and Push Jobs begin running to synchronize changes with other nodes.

Each node requires properties that allow it to connect to a database and register with a parent node. To give a node its identity, the following properties are used:

group.id
The node group that this node is a member of. Synchronization is specified between node groups, which means you only need to specify it once for multiple nodes in the same group.

external.id
The external id for this node has meaning to the user and provides integration into the system where it is deployed. For example, it might be a retail store number or a region number. The external id can be used in expressions for conditional and subset data synchronization. Behind the scenes, each node has a unique sequence number for tracking synchronization events. That makes it possible to assign the same external id to multiple nodes, if desired.

sync.url
The URL where this node can be contacted for synchronization. At startup and during each heartbeat, the node updates its entry in the database with this URL.

When a new node is first started, it is has no information about synchronizing. It contacts the registration server in order to join the network and receive its configuration. The configuration for all nodes is stored on the registration server, and the URL must be specified in the following property:

registration.url
The URL where this node can connect for registration to receive its configuration. The registration server is part of SymmetricDS and is enabled as part of the deployment.

Important

Note that a *registration server node* is defined as one whose `registration.url` is either (a) blank, or (b) identical to its `sync.url`.

When deploying to an application server, it is common for database connection pools to be found in the Java naming directory (JNDI). In this case, set the following property:

db.jndi.name
The name of the database connection pool to use, which is registered in the JNDI directory tree of the application server. It is recommended that this DataSource is NOT transactional, because SymmetricDS will handle its own transactions.

For a deployment where the database connection pool should be created using a JDBC driver, set the following properties:

db.driver
The class name of the JDBC driver.

db.url
The JDBC URL used to connect to the database.

db.user
The database username, which is used to login, create, and update SymmetricDS tables.

db.password
The password for the database user.

4.2. Node

A *node*, a single instance of SymmetricDS, is defined in the NODE table. Two other tables play a direct role in defining a node, as well The first is NODE_IDENTITY. The *only* row in this table is inserted in the database when the node first *registers* with a parent node. In the case of a root node, the row is

entered by the user. The row is used by a node instance to determine its node identity.

The following SQL statements set up a top-level registration server as a node identified as "00000" in the "corp" node group.

```
insert into SYM_NODE
  (node_id, node_group_id, external_id, sync_enabled)
values
  ('00000', 'corp', '00000', 1);

insert into SYM_NODE_IDENTITY values ('00000');
```

The second table, NODE_SECURITY has rows created for each *child* node that registers with the node, assuming auto-registration is enabled. If auto registration is not enabled, you must create a row in NODE and NODE_SECURITY for the node to be able to register. You can also, with this table, manually cause a node to re-register or do a re-initial load by setting the corresponding columns in the table itself. Registration is discussed in more detail in Section 4.7, Opening Registration (p. 38).

4.3. Node Group

Node Groups are straightforward to configure and are defined in the NODE_GROUP table. The following SQL statements would create node groups for "corp" and "store" based on our retail store example.

```
insert into SYM_NODE_GROUP
  (node_group_id, description)
values
  ('store', 'A retail store node');

insert into SYM_NODE_GROUP
  (node_group_id, description)
values
  ('corp', 'A corporate node');
```

4.4. Node Group Link

Similarly, Node Group links are established using a data event action of 'P' for Push and 'W' for Pull ("wait"). The following SQL statements links the "corp" and "store" node groups for synchronization. It configures the "store" nodes to push their data changes to the "corp" nodes, and the "corp" nodes to send changes to "store" nodes by waiting for a pull.

```
insert into SYM_NODE_GROUP_LINK
  (source_node_group, target_node_group, data_event_action)
values
  ('store', 'corp', 'P');

insert into SYM_NODE_GROUP_LINK
```

```
  (source_node_group, target_node_group, data_event_action)
values
  ('corp', 'store', 'W');
```

4.5. Channel

By categorizing data into channels and assigning them to TRIGGERs, the user gains more control and visibility into the flow of data. In addition, SymmetricDS allows for synchronization to be enabled, suspended, or scheduled by channels as well. The frequency of synchronization and order that data gets synchronized is also controlled at the channel level.

The following SQL statements setup channels for a retail store. An "item" channel includes data for items and their prices, while a "sale_transaction" channel includes data for ringing sales at a register.

```
insert into SYM_CHANNEL
  (channel_id, processing_order, max_batch_size, max_batch_to_send,
   extract_period_millis, batch_algorithm, enabled, description)
values
  ('item', 10, 1000, 10,  0, 'default', 1, 'Item and pricing data');

insert into SYM_CHANNEL
  (channel_id, processing_order, max_batch_size, max_batch_to_send,
   extract_period_millis, batch_algorithm, enabled, description)
values
  ('sale_transaction', 1, 1000, 10,  60000, 'transactional', 1,
   'retail sale transactions from register');
```

Batching is the grouping of data, by channel, to be transferred and committed at the client together. There are three different out-of-the-box batching algorithms which may be configured in the batch_algorithm column on channel.

default
All changes that happen in a transaction are guaranteed to be batched together. Multiple transactions will be batched and committed together until there is no more data to be sent or the max_batch_size is reached.

transactional
Batches will map directly to database transactions. If there are many small database transactions, then there will be many batches. The max_batch_size column has no effect.

nontransactional
Multiple transactions will be batched and committed together until there is no more data to be sent or the max_batch_size is reached. The batch will be cut off at the max_batch_size regardless of whether it is in the middle of a transaction.

There are also several size-related parameters that can be set by channel. They include:

max_batch_size

Specifies the maximum number of data events to process within a batch for this channel.

max_batch_to_send
Specifies the maximum number of batches to send for a given channel during a 'synchronization' between two nodes. A 'synchronization' is equivalent to a push or a pull. For example, if there are 12 batches ready to be sent for a channel and max_batch_to_send is equal to 10, then only the first 10 batches will be sent even though 12 batches are ready.

max_data_to_route
Specifices the maximum number of data rows to route for a channel at a time.

Based on your particular synchronization requirements, you can also specify whether old, new, and primary key data should be read and included during routing for a given channel. These are controlled by the columns use_old_data_to_route, use_row_data_to_route, and use_pk_data_to_route, respectively. By default, they are all 1 (true).

Finally, if data on a particular channel contains big lobs, you can set the column contains_big_lob to 1 (true) to provide SymmetricDS the hint that the channel contains big lobs. Some databases have shortcuts that SymmetricDS can take advantage of if it knows that the lob columns in DATA aren't going to contain large lobs. The definition of how large a 'big' lob is varies from database to database.

4.6. Triggers and Routers

4.6.1. Trigger

SymmetricDS captures synchronization data using database triggers. SymmetricDS' Triggers are defined in the TRIGGER table. Each record is used by SymmetricDS when generating database triggers. Database triggers are only generated when a trigger is associated with a ROUTER whose `source_node_group_id` matches the node group id of the current node.

When determining whether a data change has occurred or not, by defalt the triggers will record a change even if the data was updated to the same value(s) they were originally. For example, a data change will be captured if an update of one column in a row updated the value to the same value it already was. There is a global property, `trigger.update.capture.changed.data.only.enabled` (false by default), that allows you to override this behavior. When set to true, SymmetricDS will only capture a change if the data has truly changed (i.e., when the new column data is not equal to the old column data).

Important
The property `trigger.update.capture.changed.data.only.enabled` is currently only supported in the MySQL and Oracle dialects.

The following SQL statement defines a trigger that will capture data for a table named "item" whenever data is inserted, updated, or deleted. The trigger is assigned to a channel also called 'item'.

```
insert into SYM_TRIGGER
    (trigger_id,source_table_name,channel_id,last_update_time,create_time)
```

```
values
  ('item', 'item', 'item', current_timestamp, current_timestamp);
```

Two lobs-related settings are also available on TRIGGER:

use_stream_lobs
Specifies whether to capture lob data as the trigger is firing or to stream lob columns from the source tables using callbacks during extraction. A value of 1 indicates to stream from the source via callback; a value of 0, lob data is captured by the trigger.

use_capture_lobs
Provides a hint as to whether this trigger will capture big lobs data. If set to 1 every effort will be made during data capture in trigger and during data selection for initial load to use lob facilities to extract and store data in the database.

Important

Note that many databases allow for multiple triggers of the same type to be defined. Each database defines the order in which the triggers fire differently. If you have additional triggers beyond those SymmetricDS installs on your table, please consult your database documentation to determine if there will be issues with the ordering of the triggers.

4.6.2. Router

Routers provided in the base implementation currently include:

- Default Router - a router that sends all data to all nodes that belong to the target node group defined in the router.
- Column Match Router - a router that compares old or new column values to a constant value or the value of a node's external_id or node_id.
- Lookup Router - a router which can be configured to determine routing based on an existing or ancillary table specifically for the purpose of routing data.
- Subselect Router - a router that executes a SQL expression against the database to select nodes to route to. This SQL expression can be passed values of old and new column values.
- Scripted Router - a router that executes a Bean Shell script expression in order to select nodes to route to. The script can use the the old and new column values.
- Xml Publishing Router - a router the publishes data changes directly to a messaging solution instead of transmitting changes to registered nodes. This router must be configured manually in XML as an extension point.

The mapping between the set of triggers and set of routers is many-to-many. This means that one trigger can capture changes and route to multiple locations. It also means that one router can be defined an

associated with many different triggers.

4.6.2.1. Default Router

The simplest router is a router that sends all the data that is captured by its associated triggers to all the nodes that belong to the target node group defined in the router. A router is defined as a row in the ROUTER table. It is then linked to triggers in the TRIGGER_ROUTER table.

The following SQL statement defines a router that will send data from the 'corp' group to the 'store' group.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id,
    create_time, last_update_time)
values
  ('corp-2-store','corp', 'store', current_timestamp, current_timestamp);
```

The following SQL statement maps the 'corp-2-store' router to the item trigger.

```
insert into SYM_TRIGGER_ROUTER
  (trigger_id, router_id, initial_load_order,  create_time, last_update_time)
values
  ('item', 'corp-2-store', 1, current_timestamp, current_timestamp);
```

4.6.2.2. Column Match Router

Sometimes requirements may exist that require data to be routed based on the current value or the old value of a column in the table that is being routed. Column routers are configured by setting the `router_type` column on the ROUTER table to `column` and setting the `router_expression` column to an equality expression that represents the expected value of the column.

The first part of the expression is always the column name. The column name should always be defined in upper case. The upper case column name prefixed by OLD_ can be used for a comparison being done with the old column data value.

The second part of the expression can be a constant value, a token that represents another column, or a token that represents some other SymmetricDS concept. Token values always begin with a colon (:).

Consider a table that needs to be routed to all nodes in the target group only when a status column is set to 'OK.' The following SQL statement will insert a column router to accomplish that.

```
insert into SYM_ROUTER
(router_id, source_node_group_id, target_node_group_id, router_type,
 router_expression, create_time, last_update_time)
values
('corp-2-store-ok','corp', 'store', 'column',
 'STATUS=OK', current_timestamp, current_timestamp);
```

Consider a table that needs to be routed to all nodes in the target group only when a status column changes values. The following SQL statement will insert a column router to accomplish that. Note the use of OLD_STATUS, where the OLD_ prefix gives access to the old column value.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-status','corp', 'store', 'column',
    'STATUS!=:OLD_STATUS', current_timestamp, current_timestamp);
```

Consider a table that needs to be routed to only nodes in the target group whose STORE_ID column matches the external id of a node. The following SQL statement will insert a column router to accomplish that.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-id','corp', 'store', 'column',
    'STORE_ID=:EXTERNAL_ID', current_timestamp, current_timestamp);
```

Attributes on a NODE that can be referenced with tokens include:

- NODE_ID
- EXTERNAL_ID
- NODE_GROUP_ID

Consider a table that needs to be routed to a redirect node defined by its external id in the REGISTRATION_REDIRECT table. The following SQL statement will insert a column router to accomplish that.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-redirect','corp', 'store', 'column',
    'STORE_ID=:REDIRECT_NODE', current_timestamp, current_timestamp);
```

More than one column may be configured in a router_expression. When more than one column is configured, all matches are added to the list of nodes to route to. The following is an example where the STORE_ID column may contain the STORE_ID to route to or the constant of ALL which indicates that all nodes should receive the update.

```
```

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-multiple-matches','corp', 'store', 'column',
   'STORE_ID=ALL or STORE_ID=:EXTERNAL_ID', current_timestamp, current_timestamp);
```

The NULL keyword may be used to check if a column is null. If the column is null, then data will be routed to all nodes who qualify for the update. This following is an example where the STORE_ID column is used to route to a set of nodes who have a STORE_ID equal to their EXTERNAL_ID, or to all nodes if the STORE_ID is null.

```

insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-multiple-matches','corp', 'store', 'column',
   'STORE_ID=NULL or STORE_ID=:EXTERNAL_ID', current_timestamp, current_timestamp);
```

4.6.2.3. Lookup Table Router

A lookup table may contain the id of the node where data needs to be routed. This could be an existing table or an ancillary table that is added specifically for the purpose of routing data. Lookup table routers are configured by setting the `router_type` column on the ROUTER table to `lookuptable` and setting a list of configuration parameters in the `router_expression` column.

Each of the following configuration parameters are required.

LOOKUP_TABLE
This is the name of the lookup table.

KEY_COLUMN
This is the name of the column on the table that is being routed. It will be used as a key into the lookup table.

LOOKUP_KEY_COLUMN
This is the name of the column that is the key on the lookup table.

EXTERNAL_ID_COLUMN
This is the name of the column that contains the external_id of the node to route to on the lookup table.

Note that the lookup table will be read into memory and cached for the duration of a routing pass for a single channel.

Consider a table that needs to be routed to a specific store, but the data in the changing table only contains brand information. In this case, the STORE table may be used as a lookup table.

```

```

```
insert into SYM_ROUTER
(router_id, source_node_group_id, target_node_group_id, router_type,
 router_expression, create_time, last_update_time)
values
('corp-2-store-ok','corp', 'store', 'lookuptable',
 'LOOKUP_TABLE=STORE
KEY_COLUMN=BRAND_ID
LOOKUP_KEY_COLUMN=BRAND_ID
EXTERNAL_ID_COLUMN=STORE_ID', current_timestamp, current_timestamp);
```

4.6.2.4. Subselect Router

Sometimes routing decisions need to be made based on data that is not in the current row being synchronized. Consider an example where an Order table and a OrderLineItem table need to be routed to a specific store. The Order table has a column named order_id and STORE_ID. A store node has an external_id that is equal to the STORE_ID on the Order table. OrderLineItem, however, only has a foreign key to its Order of order_id. To route OrderLineItems to the same nodes that the Order will be routed to, we need to reference the master Order record.

There are two possible ways to route the OrderLineItem in SymmetricDS. One is to configure a 'subselect' router_type on the ROUTER table and the other is to configure an external_select on the TRIGGER table.

A 'subselect' is configured with a router_expression that is a SQL select statement which returns a result set of the node_ids that need routed to. Column tokens can be used in the SQL expression and will be replaced with row column data. The overhead of using this router type is high because the 'subselect' statement runs for each row that is routed. It should not be used for tables that have a lot of rows that are updated. It also has the disadvantage that if the Order master record is deleted, then no results would be returned and routing would not happen. The router_expression is appended to the following SQL statement in order to select the node ids.

```
select c.node_id from sym_node c where
  c.node_group_id=:NODE_GROUP_ID and c.sync_enabled=1 and
```

Consider a table that needs to be routed to all nodes in the target group only when a status column is set to 'OK.' The following SQL statement will insert a column router to accomplish that.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store','corp', 'store', 'subselect',
    'c.external_id in (select STORE_ID from order where order_id=:ORDER_ID)',
    current_timestamp, current_timestamp);
```

Alternatively, when using an external_select on the TRIGGER table, data is captured in the

EXTERNAL_DATA column of the DATA table at the time a trigger fires. The EXTERNAL_DATA can then be used for routing by using a router_type of 'column'. The advantage of this approach is that it is very unlikely that the master Order table will have been deleted at the time any DML accures on the OrderLineItem table. It also is a bit more effcient than the 'subselect' approach, although the triggers produced do run the extra external_select inline with application database updates.

In the following example, the STORE_ID is captured from the Order table in the EXTERNAL_DATA column. EXTERNAL_DATA is always available for routing as a virtual column in a 'column' router. The router is configured to route based on the captured EXTERNAL_DATA to all nodes whose external_id matches. Note that other supported node attribute tokens can also be used for routing.

```
insert into SYM_TRIGGER
  (trigger_id,source_table_name,channel_id,external_select,
    last_update_time,create_time)
values
  ('orderlineitem', 'orderlineitem', 'orderlineitem','select STORE_ID
    from order where order_id=$(curTriggerValue).$(curColumnPrefix)order_id',
    current_timestamp, current_timestamp);

insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-ext','corp', 'store', 'column',
    'EXTERNAL_DATA=:EXTERNAL_ID', current_timestamp, current_timestamp);
```

Note the syntax $(curTriggerValue).$(curColumnPrefix). This translates into "OLD_" or "NEW_" based on the DML type being run. In the case of Insert or Update, it's NEW_. For Delete, it's OLD_ (since there is no new data). In this way, you can access the DML-appropriate value for your select statement.

4.6.2.5. Scripted Router

When more flexibility is needed in the logic to choose the nodes to route to, then the a scripted router may be used. The currently available scripting language is Bean Shell. Bean Shell is a Java-like scripting language. Documentation for the Bean Shell scripting language can be found at http://www.beanshell.org.

The router_type for a Bean Shell scripted router is 'bsh'. The router_expression is a valid Bean Shell script that:

- adds node ids to the 'targetNodes' collection which is bound to the script
- returns a new collection of node ids
- returns a single node id
- returns true to indicate that all nodes should be routed or returns false to indicate that no nodes should be routed

Also bound to the script evaluation is a list of 'nodes'. The list of 'nodes' is a list of eligible Node objects. The current data column values and the old data column values are bound to the script evaluation as Java object representations of the column data. The columns are bound using the uppercase names of the

columns. Old values are bound to uppercase representations that are prefixed with 'OLD_'.

In the following example, the node_id is a combination of STORE_ID and WORKSTATION_NUMBER, both of which are columns on the table that is being routed.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-bsh','corp', 'store', 'bsh',
    'targetNodes.add(STORE_ID + "-" + WORKSTATION_NUMBER);',
    current_timestamp, current_timestamp);
```

The same could also be accomplished by simply returning the node id. The last line of a bsh script is always the return value.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-bsh','corp', 'store', 'bsh',
    'STORE_ID + "-" + WORKSTATION_NUMBER',
    current_timestamp, current_timestamp);
```

The following example will synchronize to all nodes if the FLAG column has changed, otherwise no nodes will be synchronized. Note that here we make use of OLD_, which provides access to the old column value.

```
insert into SYM_ROUTER
  (router_id, source_node_group_id, target_node_group_id, router_type,
    router_expression, create_time, last_update_time)
values
  ('corp-2-store-flag-changed','corp', 'store', 'bsh',
    'FLAG != null && !FLAG.equals(OLD_FLAG)',
    current_timestamp, current_timestamp);
```

4.6.3. Trigger / Router Mappings

Two important controls can be configured for a specific Trigger / Router combination: Initial Load and Ping Back. The parameters for these can be found in the Trigger / Router mapping table, TRIGGER_ROUTER.

4.6.3.1. Initial Load

An initial load is the process of seeding tables at a target node with data from its parent node. When a node connects and data is extracted, after it is registered and if an initial load was requested, each table

that is configured to synchronize to the target node group will be given a reload event in the order defined by the end user. A SQL statement is run against each table to get the data load that will be streamed to the target node. The selected data is filtered through the configured router for the table being loaded. If the data set is going to be large, then SQL criteria can optionally be provided to pair down the data that is selected out of the database.

An initial load can not occur until after a node is registered. An initial load is requested by setting the `initial_load_enabled` column on NODE_SECURITY to *1* on the row for the target node in the parent node's database. The next time the target node synchronizes, reload batches will be inserted. At the same time reload batches are inserted, all previously pending batches for the node are marked as successfully sent.

Important

Note that if the parent node that a node is registering with is *not* a registration server node (as can happen with a registration redirect or certain non-tree structure node configurations) the parent node's NODE_SECURITY entry must exist at the parent node and have a non-null value for column `initial_load_time`. Nodes can't be registered to non-registration-server nodes without this value being set one way or another (i.e., manually, or as a result of an initial load occuring at the parent node).

SymmetricDS recognizes that an initial load has completed when the `initial_load_time` column on the target node is set to a non-null value.

An initial load is accomplished by inserting reload batches in a defined order according to the `initial_load_order` column on TRIGGER_ROUTER. Initial load data is always queried from the source database table. All data is passed through the configured router to filter out data that might not be targeted at a node.

An efficient way to select a subset of data from a table for an initial load is to provide an `initial_load_select` clause on TRIGGER_ROUTER. This clause, if present, is applied as a `where` clause to the SQL used to select the data to be loaded. The clause may use "t" as an alias for the table being loaded, if needed. If an `initial_load_select` clause is provided, data will *not* be passed through the configured router during initial load. In cases where routing is done using a feature like Section 4.6.2.4, Subselect Router (p. 33) , an `initial_load_select` clause matching the subselect's criteria would be a more efficient approach.

One example of the use of an initial load select would be if you wished to only load data created more recently than the start of year 2011. Say, for example, the column `created_time` contains the creation date. Your `initial_load_select` would read `created_time > ts {'2011-01-01 00:00:00.0000'}` (using whatever timestamp format works for your database). This then gets applied as a `where` clause when selecting data from the table.

Important

When providing an `initial_load_select` be sure to test out the criteria against production data in a query browser. Do an explain plan to make sure you are properly using indexes.

4.6.3.2. Dead Triggers

Occasionally the decision of what data to load initially results in additional triggers. These triggers, known as *Dead Triggers*, are configured such that they do not capture any data changes. A "dead" Trigger is one that does not capture data changes. In other words, the `sync_on_insert`, `sync_on_update`, and `sync_on_delete` properties for the Trigger are all set to false. However, since the Trigger is specified, it *will* be included in the initial load of data for target Nodes.

Why might you need a Dead Trigger? A dead Trigger might be used to load a read-only lookup table, for example. It could also be used to load a table that needs populated with example or default data. Another use is a recovery load of data for tables that have a single direction of synchronization. For example, a retail store records sales transaction that synchronize in one direction by trickling back to the central office. If the retail store needs to recover all the sales transactions from the central office, they can be sent are part of an initial load from the central office by setting up dead Triggers that "sync" in that direction.

The following SQL statement sets up a non-syncing dead Trigger that sends the `sale_transaction` table to the "store" Node Group from the "corp" Node Group during an initial load.

```
insert into sym_trigger (TRIGGER_ID,SOURCE_CATALOG_NAME,
  SOURCE_SCHEMA_NAME,SOURCE_TABLE_NAME,CHANNEL_ID,
  SYNC_ON_UPDATE,SYNC_ON_INSERT,SYNC_ON_DELETE,
  SYNC_ON_INCOMING_BATCH,NAME_FOR_UPDATE_TRIGGER,
  NAME_FOR_INSERT_TRIGGER,NAME_FOR_DELETE_TRIGGER,
  SYNC_ON_UPDATE_CONDITION,SYNC_ON_INSERT_CONDITION,
  SYNC_ON_DELETE_CONDITION,EXTERNAL_SELECT,
  TX_ID_EXPRESSION,EXCLUDED_COLUMN_NAMES,
  CREATE_TIME,LAST_UPDATE_BY,LAST_UPDATE_TIME)
  values ('SALE_TRANSACTION_DEAD',null,null,
  'SALE_TRANSACTION','transaction',
  0,0,0,0,null,null,null,null,null,null,null,null,null,
  current_timestamp,'demo',current_timestamp);

insert into sym_router (ROUTER_ID,TARGET_CATALOG_NAME,TARGET_SCHEMA_NAME,
  TARGET_TABLE_NAME,SOURCE_NODE_GROUP_ID,TARGET_NODE_GROUP_ID,ROUTER_TYPE,
  ROUTER_EXPRESSION,SYNC_ON_UPDATE,SYNC_ON_INSERT,SYNC_ON_DELETE,
  CREATE_TIME,LAST_UPDATE_BY,LAST_UPDATE_TIME)
  values ('CORP_2_STORE',null,null,null,
  'corp','store',null,null,1,1,1,
  current_timestamp,'demo',current_timestamp);

insert into sym_trigger_router (TRIGGER_ID,ROUTER_ID,INITIAL_LOAD_ORDER,
  INITIAL_LOAD_SELECT,CREATE_TIME,LAST_UPDATE_BY,LAST_UPDATE_TIME)
  values ('SALE_TRANSACTION_DEAD','CORP_2_REGION',100,null,
   current_timestamp,'demo',current_timestamp);
```

4.6.3.3. Enabling "Ping Back"

As discussed in Section 3.6.3.2, Circular References and "Ping Back" (p. 22) SymmetricDS, by default, avoids circular data changes. When a trigger fires as a result of SymmetricDS itself (such as the case when sync on incoming batch is set), it records the originating source node of the data change in `source_node_id`. During routing, if routing results in sending the data back to the originating source node,

the data is not routed by default. If instead you wish to route the data back to the originating node, you can set the `ping_back_enabled` column for the needed particular trigger / router combination. This will cause the router to "ping" the data back to the originating node when it usually would not.

4.7. Opening Registration

Node registration is the act of setting up a new NODE and NODE_SECURITY so that when the new node is brought online it is allowed to join the system. Nodes are only allowed to register if rows exist for the node and the `registration_enabled` flag is set to 1. If the `auto.registration` SymmetricDS property is set to true, then when a node attempts to register, if registration has not already occurred, the node will automatically be registered.

SymmetricDS allows you to have multiple nodes with the same `external_id`. Out of the box, openRegistration will open a new registration if a registration already exists for a node with the same external_id. A new registration means a new node with a new `node_id` and the same `external_id` will be created. If you want to re-register the same node you can use the `reOpenRegistration()` JMX method which takes a `node_id` as an argument.

4.8. Transforming Data

New to SymmetricDS 2.4, SymmetricDS is now able to transform synchronized data by way of configuration (previously, for most cases a custom data loader would need to have been written). This transformation can take place on a source node or on a target node, as the data is being loaded or extracted. With this new feature you can, for example:

- Copy a column from a source table to two (or more) target table columns,
- Merge columns from two or more source tables into a single row in a target table,
- Insert constants in columns in target tables based on source data synchronizations,
- Insert multiple rows of data into a single target table based on one change in a source table,
- Apply a Bean Shell script to achieve a custom transform when loading into the target database.

These transformations can take place either on the target or on the source, and as data is either being extracted or loaded. In either case, the transformation is initiated due to existence of a source synchronization trigger. The source trigger creates the synchronization data, while the transformation configuration decides what to do with the sychronization data as it is either being extracted from the source or loaded into the target. You have the flexibility of defining different transformation behavior depending on whether the source change that triggered the synchronization was an Insert, Update, or Delete. In the case of Delete, you even have options on what exactly to do on the target side, be it a delete of a row, setting columns to specific values, or absolutely nothing at all.

A few key concepts are important to keep in mind to understand how SymmetricDS performs transformations. The first concept is that of the "source operation" or "source DML type", which is the type of operation that occurred to generate the synchronization data in the first place (i.e., an insert, a

delete, or an update). Your transformations can be configured to act differently based on the source DML type, if desired. When transforming, by default the DML action taken on the target matches that of the action taken on the row in the source (although this behavior can be altered through configuration if needed). If the source DML type is an Insert, for example, the resulting transformation DML(s) will be Insert(s).

Another important concept is the way in which transforms are applied. Each source operation may map to one or more transforms and result in one or more operations on the target tables. Each of these target operations are performed as independent operations in sequence and must be "complete" from a SQL perspective. In other words, you must define columns for the transformation that are sufficient to fill in any primary key or other required data in the target table if the source operation was an Insert, for example.

Finally, please note that the tranformation engine relies on a source trigger / router existing to supply the source data for the transformation. The transform configuration will never be used if the source table and target node group does not have a defined trigger / router combination for that source table and target node group.

4.8.1. Transform Configuration Tables

SymmetricDS stores its transformation configuration in two configuration tables, TRANSFORM_TABLE and TRANSFORM_COLUMN. Defining a transformation involves configuration in both tables, with the first table defining which source and destination tables are involved, and the second defining the columns involved in the transformation and the behavior of the data for those columns. We will explain the various options available in both tables and the various pre-defined transformation types.

To define a transformation, you will first define the source table and target table that applies to a particular transformation. The source and target tables, along with a unique identifier (the transform_id column) are defined in TRANSFORM_TABLE. In addition, you will specify the source_node_group_id and target_node_group_id to which the transform will apply, along with whether the transform should occur on the Extract step or the Load step (transform_point). All of these values are required.

Three additional configuration settings are also defined at the source-target table level: the order of the transformations, the behavior when deleting, and whether an update should always be attempted first. More specifically,

- transform_order: For a single source operation that is mapped to a transformation, there could be more than one target operation that takes place. You may control the order in which the target operations are applied through a configuration parameter defined for each source-target table combination. This might be important, for example, if the foreign key relationships on the target tables require you to execute the transformations in a particular order.

- delete_action: When a source operation of Delete takes place, there are three possible ways to handle the transformation at the target. The options include:

  - NONE - The delete results in no target changes.

  - DEL_ROW - The delete results in a delete of the row as specified by the pk columns defined in

the transformation configuration.

- UPDATE_COL - The delete results in an Update operation on the target which updates the specific rows and columns based on the defined transformation.

- update_first: This option overrides the default behavior for an Insert operation. Instead of attempting the Insert first, SymmetricDS will always perform an Update first and then fall back to an Insert if that fails. Note that, by default, fall back logic *always* applies for Insert and Updates. Here, all you a specifying is whether to always do an Update first, which can have performance benefits under certain situations you may run into.

For each transformation defined in TRANSFORM_TABLE, the columns to be transformed (and how they are transformed) are defined in TRANSFORM_COLUMN. This column-level table typically has several rows for each transformation id, each of which defines the source column name, the target column name, as well as the following details:

- include_on: Defines whether this entry applies to source operations of Insert (I), Update (U), or Delete (D), or any source operation.

- pk: Indicates that this mapping is used to define the "primary key" for identifying the target row(s) (which may or may not be the true primary key of the target table). This is used to define the "where" clause when an Update or Delete on the target is occurring. At least one row marked as a pk should be present for each transform_id.

- transform_type, transform_expression: Specifies how the data is modified, if at all. The available transform types are discussed below, and the default is 'copy', which just copies the data from source to target.

- transform_order: In the event there are more than one columns to transform, this defines the relative order in which the transformations are applied.

4.8.2. Transformation Types

There are several pre-defined transform types available in SymmetricDS. Additional ones can be defined by creating and configuring an extension point which implements the `IColumnTransform` interface. The pre-defined transform types include the following (the transform_type entry is shown in parentheses):

- Copy Column Transform ('copy'): This transformation type copies the source column value to the target column. This is the default behavior.

- Constant Transform ('const'): This transformation type allows you to map a constant value to the given target column. The constant itself is placed in transform_expression.

- Variable Transform ('variable'): This transformation type allows you to map a built-in variable to the given target column. The variable name is placed in transform_expression. The following variables are available: `system_date` is the current system date, and `system_timestamp` is the current system date and time.

- Additive Transform ('additive'): This transformation type is used for numeric data. It computes the change between the old and new values on the source and then adds (or subtracts) the value from the existing value in the target column. For example, if the source column changed from a 2 to a 4, and the target column is currently 10, the effect of the transform will be to change the target column to a value of 12 (10+(4-2) => 12).

- Substring Transform ('substr'): This transformation computes a substring of the source column data and uses the substring as the target column value. The transform_expression can be a single integer (`n`, the beginning index), or a pair of comma-separated integers (`n,m` - the beginning and ending index). The transform behaves as the Java substring function would using the specified values in transform_expression.

- Lookup Transform ('lookup'): This transformation allows for the lookup of a column value from a single row using a SQL statement provided in the transform_expression. The SQL statement can access the current source row using parameters with a name of the source column name in upper case prefixed by a colon. For example, a SQL statement that looks up `customer_id` from another table using the `id` column of the current source row is `select customer_id from customers where group_id = :ID`.

- Multiplier Transform ('multiply'): This transformation allows for the creation of multiple rows in the target table based on the transform_expression. This transform type can only be used on a primary key column. The transform_expression is a SQL statement that returns the list to be used to create the multiple targets.

- Shell Script Transform ('bsh'): This transformation allows you to provide a Bean Shell script in transform_expression and executes the script at the time of transformation. Some variables are provided to the script: `COLUMN_NAME` is a variable for a source column in the row, where the variable name is the column name in uppercase; `currentValue` is the value of the current source column; `oldValue` is the old value of the source column for an updated row; `jdbcTemplate` is a Spring JdbcTemplate object for querying or updating the database.

- Variable Transform ('variable'): This transformation allows you to place a dynamic variable (such as the current database time) into the target column. The only transform_expression value currently supported is `system_timestamp`.

- Identity Transform ('identity'): This transformation allows you to insert into a identity column by computing a new identity, not copying the actual identity value from the source.

Chapter 5. Advanced Topics

This chapter focuses on a variety of topics, including deployment options, jobs, clustering, encryptions, synchronization control, and configuration of SymmetricDS.

5.1. Advanced Synchronization

5.1.1. Bi-Directional Synchronization

SymmetricDS allows tables to be synchronized bi-directionally. Note that an outgoing synchronization does not process changes during an incoming synchronization on the same node unless the trigger was created with the `sync_on_incoming_batch` flag set. If the `sync_on_incoming_batch` flag is set, then update loops are prevented by a feature that is available in most database dialects. More specifically, during an incoming synchronization the source `node_id` is put into a database session variable that is available to the database trigger. Data events are not generated if the target `node_id` on an outgoing synchronization is equal to the source `node_id`.

By default, only the columns that changed will be updated in the target system.

More complex conflict resolution strategies can be accomplished by using the `IDataLoaderFilter` extension point which has access to both old and new data.

5.1.2. Multi-Tiered Synchronization

As shown in Section 3.2, Organizing Nodes (p. 15) , there may be scenarios where data needs to flow through multiple tiers of nodes that are organized in a tree-like network with each tier requiring a different subset of data. For example, you may have a system where the lowest tier may by a computer or device located in a store. Those devices may connect to a server located physically at that store. Then the store server may communicate with a corporate server for example. In this case, the three tiers would be device, store, and corporate. Each tier is typically represented by a node group. Each node in the tier would belong to the node group representing that tier.

A node will always push and pull data to other node groups according to the node group link configuration. A node can only pull and push data to other nodes that are represented `node` table in its database and having `sync_enabled = 1`. Because of this, a tree-like hierarchy of nodes can be created by having only a subset of nodes belonging to the same node group represented at the different branches of the tree.

If auto registration is turned *off*, then this setup must occur manually by opening registration for the desired nodes at the desired parent node and by configuring each node's `registration.url` to be the parent node's URL. The parent node is always tracked by the setting of the parent's `node_id` in the `created_at_node_id` column of the new node. When a node registers and downloads its configuration it is always provided the configuration for nodes that might register with the node itself based on the Node Group Links defined in the parent node.

5.1.2.1. Registration Redirect

When deploying a multi-tiered system it may be advantageous to have only one registration server, even though the parent node of a registering node could be any of a number of nodes in the system. In SymmetricDS the parent node is always the node that a child registers with. The REGISTRATION_REDIRECT table allows a single node, usually the root server in the network, to redirect registering nodes to their true parents. It does so based on a mapping found in the table of the external id (`registrant_external_id`) to the parent's node id (`registration_node_id`).

For example, if it is desired to have a series of regional servers that workstations at retail stores get assigned to based on their `external_id`, the store number, then you might insert into REGISTRATION_REDIRECT the store number as the `registrant_external_id` and the `node_id` of the assigned region as the `registration_node_id`. When a workstation at the store registers, the root server send an HTTP redirect to the `sync_url` of the node that matches the `registration_node_id`.

Important

Please see Section 4.6.3.1, Initial Load (p. 35) for important details around initial loads and registration when using registration redirect.

5.2. Jobs

The SymmetricDS software allows for outgoing and incoming changes to be synchronized to/from other databases. The node that initiates a synchronization connection is the client, and the node receiving a connection is the host. Because synchronization is configurable to push or pull in either direction, the same node can act as either a client or a host in different circumstances.

The SymmetricDS software consists of a series of background jobs, managers, Servlets, and services wired together via dependency injection using the Spring Framework.

As a client, the node runs the router job, push job and pull job on a timer thread. The router job uses services to create batches that are targeted at certain nodes. The push job uses services to extract and stream data to another node (that is, it pushes data). The response from a push is a list of batch acknowlegements to indicate that data was loaded. The pull job uses services to load data that is streamed from another node (*i.e.*, it pulls data). After loading data, a second connection is made to send a list of batch acknowlegements.

As a host, the node waits for incoming connections that pull, push, or acknowledge data changes. The push Servlet uses services to load data that is pushed from a client node. After loading data, it responds with a list of batch acknowledgements. The pull Servlet uses services to extract, and stream data back to the client node. The ack Servlet uses services to update the status of data that was loaded at a client node. The router job batches and routes data.

By default, data is extracted from the source database into memory until a threshold size is reached. If the threshold size is reached, data is streamed to a temporary file in the JVM's default temporary directory. Next, the data is streamed to the target node across the transport layer. The receiving node will cache the

data in memory until the threshold size is reached, writing to a temporary file if necessary. At last, the data is loaded into the target database by the data loader. This step by step approach allows for extract time, transport time, and load time to all be measured independently. It also allows database resources to be used most optimally.

The transport manager handles the incoming and outgoing streams of data between nodes. The default transport is based on a simple implementation over HTTP. An internal transport is also provided. It is possible to add other implementations, such as a socket-based transport manager.

Node communication over HTTP is represented in the following figure.

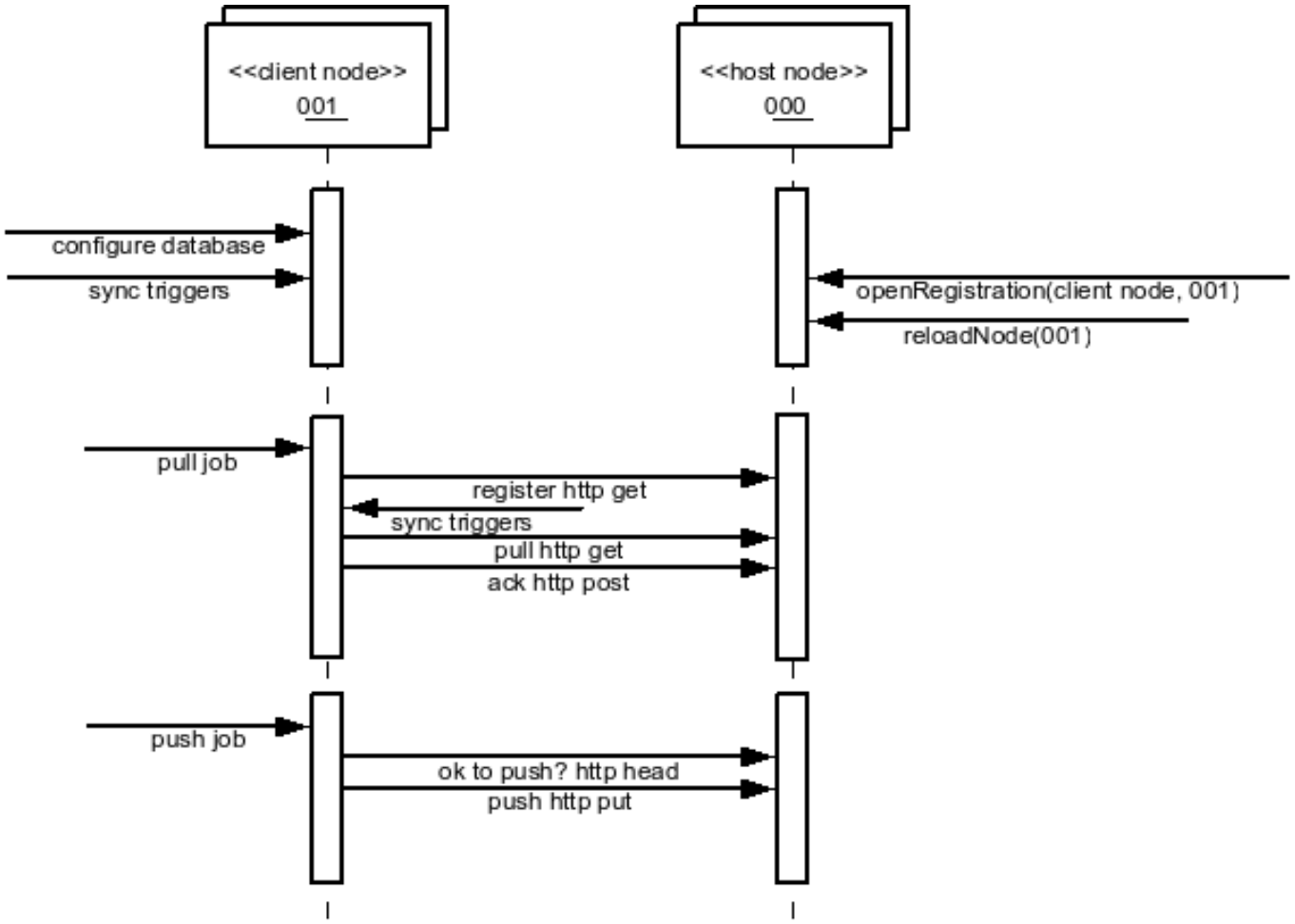


Figure 5.1. Node Communication

The `StandaloneSymmetricEngine` is wrapper API that can be used to directly start the client services only. The `SymmetricWebServer` is a wrapper API that can be used to directly start *both* the client and host services inside a Jetty web container. The `SymmetricLauncher` provides command line tools to work with and start SymmetricDS.

5.2.1. Route Job

5.2.1.1. Overview

The SymmetricDS-created database triggers cause data to be capture in the DATA table. The next step in the synchronization process is to process the change data to determine which nodes, if any, the data should be routed to. This step is performed by the *Route Job*. In addition to determining which nodes data will be sent to, the Route Job is also responsible for determing how much data will be batched together for transport. It is a single background task that inserts into DATA_EVENT and OUTGOING_BATCH.

At a high level, the Route Job is straightforward. It collects a list of data ids from DATA which haven't yet been routed (see Section 5.2.1.2, Data Gaps (p. 45) for much more detail about this step), one channel at a time, up to a limit specified by the channel configuration (`max_data_to_route`, on CHANNEL). The data is then batched based on the `batch_algorithm` defined for the channel and as documented in Section 4.5, Channel (p. 27) . Note that, for the default batching algorithm, there may actually be more than `max_data_to_route` included depending on the transaction boundaries. The mapping of data to specific nodes, organized into batches, is then recorded in OUTGOING_BATCH with a status of "RT" in each case (representing the fact that the Route Job is still running). Once the routing algorithms and batching are completed, the batches are organized with their corresponding data ids and saved in DATA_EVENT. Once DATA_EVENT is updated, the rows in OUTGOING_BATCH are updated to a status of New "NE".

5.2.1.2. Data Gaps

On the surface, the first Route Job step of collecting unrouted data ids seems simple: assign sequential data ids for each data row as it's inserted and keep track of which data id was last routed and start from there. The difficulty arises, however, due to the fact that there can be multiple transactions inserting into DATA simultaneously. As such, a given section of rows in the DATA table may actually contain "gaps" in the data ids when the Route Job is executing. Most of these gaps are only temporarily and fill in at some point after routing and need to be picked up with the next run of the Route Job. Thus, the Route Job needs to remember to route the filled-in gaps. Worse yet, some of these gaps are actually permanent and result from a transaction that is rolled back for some reason. In this case, the Route Job must continue to watch for the gap to fill in and, at some point, eventually gives up and assumes the gap is permanent and can be skipped. All of this must be done in some fashion that guarantees that gaps are routed when they fill in while also keeping routing as efficient as possible.

SymmetricDS handles the issue of data gaps by making use of a table, DATA_GAP, to record gaps found in the data ids. In fact, this table completely defines the entire range of data tha can be routed at any point in time. For a brand new instance of SymmetricDS, this table is empty and SymmetricDS creates a gap starting from data id of zero and ending with a very large number (defined by `routing.largest.gap.size`). At the start of a Route Job, the list of valid gaps (gaps with status of 'GP') is collected, and each gap is evaluated in turn. If a gap is sufficiently old (as defined by `routing.stale.dataid.gap.time.ms`, the gap is marked as skipped (status of 'SK') and will no longer be evaluated in future Route Jobs (note that the 'last' gap (the one with the highest starting data id) is never skipped). If not skipped, then DATA_EVENT is searched for data ids present in the gap. If one or more data ids is found in DATA_EVENT, then the current gap is marked with a status of OK, and new gap(s) are created to represent the data ids still missing in the gap's range. This process is done for all gaps. If the very last gap contained data, a new gap starting from the highest data id and ending at (highest data id + `routing.largest.gap.size`) is then created. This process has resulted in an updated list of gaps which may contain new data to be routed.

5.2.2. Controlling Synchronization Frequency

The frequency of data synchronization is controlled by the coordination of a series of asynchronous events.

The Route Job determines which nodes data will be sent to, as well as how much data will be batched together for transport. When the `start.route.job` SymmetricDS property is set to `true`, the frequency that routing occurs is controlled by the `job.routing.period.time.ms`. Each time data is routed, the DATA_REF table is updated with the id of the last contiguous data row to have been processed. This is done so the query to find unrouted data is optimal.

After data is routed, it awaits transport to the target nodes. Transport can occur when a client node is configured to pull data or when the host node is configured to push data. These events are controlled by the *Push* and the *Pull Jobs*. When the `start.pull.job` SymmetricDS property is set to `true`, the frequency that data is pulled is controlled by the `job.pull.period.time.ms`. When the `start.push.job` SymmetricDS property is set to `true`, the frequency that data is pushed is controlled by the `job.push.period.time.ms`. Data is extracted by channel from the source database's DATA table at an interval controlled by the `extract_period_millis` column on the CHANNEL table. The `last_extract_time` is always recorded, by channel, on the NODE_CHANNEL_CTL table for the host node's id. When the Pull and Push Job run, if the extract period has not passed according to the last extract time, then the channel will be skipped for this run. If the `extract_period_millis` is set to zero, data extraction will happen every time the jobs run.

SymmetricDS also provides the ability to configure windows of time when synchronization is allowed. This is done using the NODE_GROUP_CHANNEL_WINDOW table. A list of allowed time windows can be specified for a node group and a channel. If one or more windows exist, then data will only be extracted and transported if the time of day falls within the window of time specified. The configured times are always for the target node's local time. If the `start_time` is greater than the `end_time`, then the window crosses over to the next day.

All data loading may be disabled by setting the `dataloader.enable` property to false. This has the effect of not allowing incoming synchronizations, while allowing outgoing synchronizations. All data extractions may be disabled by setting the `dataextractor.enable` property to false. These properties can be controlled by inserting into the root server's PARAMETER table. These properties affect every channel with the exception of the 'config' channel.

5.2.3. Sync Triggers Job

SymmetricDS examines the current configuration, corresponding database triggers, and the underlying tables to determine if database triggers need created or updated. The change activity is recorded on the TRIGGER_HIST table with a reason for the change. The following reasons for a change are possible:

- N - New trigger that has not been created before
- S - Schema changes in the table were detected
- C - Configuration changes in Trigger
- T - Trigger was missing

A configuration entry in Trigger without any history in Trigger Hist results in a new trigger being created (N). The Trigger Hist stores a hash of the underlying table, so any alteration to the table causes the trigger to be rebuilt (S). When the `last_update_time` is changed on the Trigger entry, the configuration change causes the trigger to be rebuilt (C). If an entry in Trigger Hist is missing the corresponding database trigger, the trigger is created (T).

The process of examining triggers and rebuilding them is automatically run during startup and each night by the SyncTriggersJob. The user can also manually run the process at any time by invoking the `syncTriggers()` method over JMX. The SyncTriggersJob is enabled by default to run at 15 minutes past midnight. If SymmetricDS is being run from a collection of servers (multiple instances of the same Node running against the same database), then locking should be enable to prevent database contention. The following runtime properties control the behavior of the process.

start.synctriggers.job
Whether the sync triggers job is enabled for this node. [Default: true]

job.synctriggers.aftermidnight.minutes
If scheduled, the sync triggers job will run nightly. This is how long after midnight that job will run. [Default: 15]

cluster.lock.during.sync.triggers
Indicate if the sync triggers job is clustered and requires a lock before running. [Default: false]

5.3. JMS Publishing

With the proper configuration SymmetricDS can publish XML messages of captured data changes to JMS during routing or transactionally while data loading synchronized data into a target database. The following explains how to publish to JMS during synchronization to the target database.

The XmlPublisherDataLoaderFilter is a IDataLoaderFilter that may be configured to publish specific tables as an XML message to a JMS provider. See Chapter 6, *Extending SymmetricDS* (p. 59) for information on how to configure an extension point. If the publish to JMS fails, the batch will be marked in error, the loaded data for the batch will be rolled back and the batch will be retried during the next synchronization run.

The following is an example extension point configuration that will publish four tables in XML with a root tag of *'sale'*. Each XML message will be grouped by the batch and the column names identified by the groupByColumnNames property which have the same values.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="configuration-publishingFilter"
      class="org.jumpmind.symmetric.integrate.XmlPublisherDataLoaderFilter">
        <property name="xmlTagNameToUseForGroup" value="sale"/>
```

```
        <property name="tableNamesToPublishAsGroup">
            <list>
               <value>SALE_TX</value>
               <value>SALE_LINE_ITEM</value>
               <value>SALE_TAX</value>
               <value>SALE_TOTAL</value>
            </list>
        </property>
        <property name="groupByColumnNames">
            <list>
               <value>STORE_ID</value>
               <value>BUSINESS_DAY</value>
               <value>WORKSTATION_ID</value>
               <value>TRANSACTION_ID</value>
            </list>
        </property>
        <property name="publisher">
           <bean class="org.jumpmind.symmetric.integrate.SimpleJmsPublisher">
               <property name="jmsTemplate" ref="definedSpringJmsTemplate"/>
           </bean>
        </property>
    </bean>
</beans>
```

The publisher property on the XmlPublisherDataLoaderFilter takes an interface of type IPublisher. The implementation demonstrated here is an implementation that publishes to JMS using Spring's JMS template. Other implementations of IPublisher could easily publish the XML to other targets like an HTTP server, the file system or secure copy it to another server.

The above configuration will publish XML similiar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<sale xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  id="0012010-01-220031234" nodeid="00001" time="1264187704155">
  <row entity="SALE_TX" dml="I">
    <data key="STORE_ID">001</data>
    <data key="BUSINESS_DAY">2010-01-22</data>
    <data key="WORKSTATION_ID">003</data>
    <data key="TRANSACTION_ID">1234</data>
    <data key="CASHIER_ID">010110</data>
  </row>
  <row entity="SALE_LINE_ITEM" dml="I">
    <data key="STORE_ID">001</data>
    <data key="BUSINESS_DAY">2010-01-22</data>
    <data key="WORKSTATION_ID">003</data>
    <data key="TRANSACTION_ID">1234</data>
    <data key="SKU">9999999</data>
    <data key="PRICE">10.00</data>
    <data key="DESC" xsi:nil="true"/>
  </row>
  <row entity="SALE_LINE_ITEM" dml="I">
    <data key="STORE_ID">001</data>
    <data key="BUSINESS_DAY">2010-01-22</data>
    <data key="WORKSTATION_ID">003</data>
    <data key="TRANSACTION_ID">1234</data>
    <data key="SKU">9999999</data>
    <data key="PRICE">10.00</data>
    <data key="DESC" xsi:nil="true"/>
  </row>
  <row entity="SALE_TAX" dml="I">
```

```
    <data key="STORE_ID">001</data>
    <data key="BUSINESS_DAY">2010-01-22</data>
    <data key="WORKSTATION_ID">003</data>
    <data key="TRANSACTION_ID">1234</data>
    <data key="AMOUNT">1.33</data>
  </row>
  <row entity="SALE_TOTAL" dml="I">
    <data key="STORE_ID">001</data>
    <data key="BUSINESS_DAY">2010-01-22</data>
    <data key="WORKSTATION_ID">003</data>
    <data key="TRANSACTION_ID">1234</data>
    <data key="AMOUNT">21.33</data>
  </row>
</sale>
```

To publish JMS messages during routing the same pattern is valid, with the exception that the extension point would be the XmlPublisherDataRouter and the router would be configured by setting the `router_type` of a ROUTER to the Spring bean name of the registered extension point. Of course, the router would need to be linked through TRIGGER_ROUTERs to each TRIGGER table that needs published.

5.4. Deployment Options

An instance of SymmetricDS can be deployed in several ways:

- Web application archive (WAR) deployed to an application server

  This option means packaging a WAR file and deploying to your favorite web server, like Apache Tomcat. It's a little more work, but you can configure the web server to do whatever you need. SymmetricDS can also be embedded in an existing web application, if desired.

- Standalone service that embeds Jetty web server

  This option means running the *sym* command line, which launches the built-in Jetty web server. This is a simple option because it is already provided, but you lose the flexibility to configure the web server any further.

- Embedded as a Java library in an application

  This option means you must write a wrapper Java program that runs SymmetricDS. You would probably use Jetty web server, which is also embeddable. You could bring up an embedded database like Derby or H2. You could configure the web server, database, or SymmetricDS to do whatever you needed, but it's also the most work of the three options discussed thus far.

- Grails Application

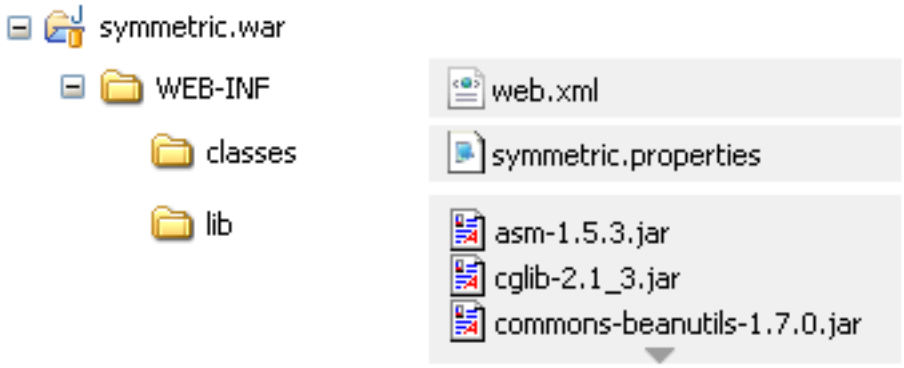
  A Grails SymmetricDS plugin is provided at the default Grails plugin site. This option ends up being a WAR deployment, but allows for the use of the Grails SDK for configuring and building the deployment. The plugin also provides Gorm (Hibernate) access to many of the core database tables.

The deployment model you choose depends on how much flexibility you need versus how easy you want it to be. Both Jetty and Tomcat are excellent, scalable web servers that compete with each other and have great performance. Most people choose either the *Standalone* or *Web Archive* with Tomcat 5.5 or 6. Deploying to Tomcat is a good middle-of-the-road decision that requires a little more work for more flexibility.

Next, we will go into a little more detail on the first three deployment options listed above.

5.4.1. Web Archive

As a web application archive, a WAR is deployed to an application server, such as Tomcat, Jetty, or JBoss. The structure of the archive will have a `web.xml` file in the `WEB-INF` folder, an appropriately configured `symmetric.properties` file in the `WEB-INF/classes` folder, and the required JAR files in the `WEB-INF/lib` folder.



A war file can be generated using the standalone installation's sym utility and the --create-war option. The command requires the name of the war file to generate. It essentially packages up the web directory, the conf directory and includes an optional properties file. Note that if a properties file is included, it will be copied to WEB-INF/classes/symmetric.properties. This is the same location conf/symmetric.properties would have been copied to. The generated war distribution uses the same web.xml as the standalone deployment.

../bin/sym -p my-symmetric-ds.properties --create-war /some/path/to/symmetric-ds.war

The `web.base.servlet.path` property in `symmetric.properties` can be set if the SymmetricServlet needs to coexist with other Servlets. By default, the value is blank. If you set it to, say, `web.base.servlet.path=sync` for exmaple, `registration.url` would be `http://server:port/sync`.

5.4.2. Standalone

A standalone service can use the `sym` command line options to start a server. An embedded instance of Jetty is used to service web requests for all the servlets.

```
/symmetric/bin/sym --properties root.properties --port 8080 --server
```

This example starts the SymmetricDS server on port 8080 with the startup properties found in the `root.properties` file.

5.4.3. Embedded

A Java application with the SymmetricDS Java Archive (JAR) libraries in its classpath can use the `SymmetricWebServer` to start the server.

```
import org.jumpmind.symmetric.SymmetricWebServer;

public class StartSymmetricEngine {

    /**
     * Start an engine that is configured by two properties files. One is
     * packaged with the application and contains overridden properties that are
     * specific to the application. The other is found in the application's
     * working directory. It can be used to setup environment specific
     * properties.
     */
    public static void main(String[] args) throws Exception {

        // Specify the properties file and the web directory
        SymmetricWebServer node = new SymmetricWebServer(
                                    "classpath://my-application.properties", "web");

        // indicates that the start() method should return
        node.setJoin(false);

        // this will create the database, sync triggers, start jobs running
        node.start(8080);

        // this will stop the node
        node.stop();
    }

}
```

This example starts the SymmetricDS server on port 8080 with a startup properties file that is found on the application's classpath to provide properties that override the default values. Note if the properties file should be found on the file system the property file name would be prepended with file://.

The second parameter to the web server is the name of the directory where the web.xml for the application can be found. You may use the web.xml from the standalone deployment. In this example you would place the web.xml in the web/WEB-INF/ directory.

5.5. Running SymmetricDS as a Service

SymmetricDS can be configured to start and run as a service in both Windows and *nix platforms.

5.5.1. Running as a Windows Service

SymmetricDS uses the Java Service Wrapper product from Tanuki Software to run in the background as a Windows system service. The Java Service Wrapper executable is named `sym_service.exe` so it can be easily identified from a list of running processes. To install the service, use the provided script:

```
bin\install_service.bat
```

The service configuration is found in `conf/sym_service.conf`. Edit this file if you want to change the default port number (8080), initial memory size (256 MB), log file size (10 MB), or other settings. When started, the server will look in the `conf` directory for the `symmetric.properties` file and the `log4j.xml` file. Logging for standard out, error, and application are written to the `logs` directory.

Most configuration changes do not require the service to be re-installed. To un-install the service, use the provided script:

```
bin\uninstall_service.bat
```

Use the **net** command to start and stop the service:

```
net start symmetric
net stop symmetric
```

5.5.2. Running as a *nix Service

SymmetricDS uses the Java Service Wrapper product from Tanuki Software to run in the background as a Unix system service. The Java Service Wrapper executable is named `sym_service` so it can be easily identified from a list of running processes. The service configuration is found in `conf/sym_service.conf`. Edit this file if you want to change the default port number (8080), initial memory size (256 MB), log file size (10 MB), or other settings.

An init script is provided to work with standard Unix run configuration levels. The `sym_service.initd` file follows the Linux Standard Base specification, which should work on many systems, including Fedora and Debian-based distributions. To install the script, copy it into the system init directory:

```
cp bin/sym_service.initd /etc/init.d/sym_service
```

Edit the init script to set the SYM_HOME variable to the directory where SymmetricDS is located. The init script calls the `sym_service` executable.

To enable the service to run automatically when the system is started:

```
/sbin/chkconfig --add sym_service
```

To disable the service from running automatically:

```
/sbin/chkconfig --del sym_service
```

On Suse Linux install the service by calling:

```
/usr/lib/lsb/install_initd sym_service
```

Remove the service by calling:

```
/usr/lib/lsb/remove_initd sym_service
```

Use the **service** command to start, stop, and query the status of the service:

```
/sbin/service sym_service start
/sbin/service sym_service stop
/sbin/service sym_service status
```

Alternatively, call the init.d script directly:

```
/etc/init.d/sym_service start
/etc/init.d/sym_service stop
/etc/init.d/sym_service status
```

5.6. Clustering

A single SymmetricDS node may be clustered across a series of instances, creating a web farm. A node might be clustered to provide load balancing and failover, for example.

When clustered, a hardware load balancer is typically used to round robin client requests to the cluster. The load balancer should be configured for stateless connections. Also, the `sync.url` (discussed in Section 4.1, Node Properties (p. 24)) SymmetricDS property should be set to the URL of the load balancer.

If the cluster will be running any of the SymmetricDS jobs, then the `cluster.lock.enabled` property should be set to `true`. By setting this property to true, SymmetricDS will use a row in the LOCK table as a semaphore to make sure that only one instance at a time runs a job. When a lock is acquired, a row is updated in the lock table with the time of the lock and the server id of the locking job. The lock time is set back to null when the job is finished running. Another instance of SymmetricDS cannot aquire a lock until the locking instance (according to the server id) releases the lock. If an instance is terminated while the lock is still held, an instance with the same server id is allowed to reaquire the lock. If the locking instance remains down, the lock can be broken after a period of time, specified by the `cluster.lock.timeout.ms` property, has expired. Note that if the job is still running and the lock expires, two jobs could be running at the same time which could cause database deadlocks.

By default, the locking server id is the hostname of the server. If two clustered instances are running on

the same server, then the `cluster.server.id` property may be set to indicate the name that the instance should use for its server id.

When deploying SymmetricDS to an application server like Tomcat or JBoss, no special session clustering needs to be configured for the application server.

5.7. Encrypted Passwords

The `db.user` and `db.password` properties will accept encrypted text, which protects against casual observation. The text is prefixed with `enc:` to indicate that it is encrypted. To encrypt text, use the following command:

```
sym -e secret
```

The text is encrypted using a secret key named "sym.secret" that is retrieved from a keystore file. By default, the keystore is located in `security/keystore`. The location and filename of the keystore can be overridden by setting the "sym.keystore.file" system property. If the secret key is not found, the system will generate and install a secret key for use with Triple DES cipher.

Generate a new secret key for encryption using the `keytool` command that comes with the JRE. If there is an existing key in the keystore, first remove it:

```
keytool -keystore keystore -storepass changeit -storetype jceks \
   -alias sym.secret -delete
```

Then generate a secret key, specifying a cipher algorithm and key size. Commonly used algorithms that are supported include aes, blowfish, desede, and rc4.

```
keytool -keystore keystore -storepass changeit -storetype jceks \
   -alias sym.secret -genseckey -keyalg aes -keysize 128
```

If using an alternative provider, place the provider JAR file in the SymmetricDS `lib` folder. The provider class name should be installed in the JRE security properties or specified on the command line. To install in the JRE, edit the JRE `lib/security/java.security` file and set a `security.provider.i` property for the provider class name. Or, the provider can be specified on the command line instead. Both `keytool` and `sym` accept command line arguments for the provider class name. For example, using the Bouncy Castle provider, the command line options would look like:

```
keytool -keystore keystore -storepass changeit -storetype jceks \
   -alias sym.secret -genseckey -keyalg idea -keysize 56 \
   -providerClass org.bouncycastle.jce.provider.BouncyCastleProvider \
   -providerPath ..\lib\bcprov-ext.jar
```

```
sym -jcep org.bouncycastle.jce.provider.BouncyCastleProvider -e secret
```

To customize the encryption, write a Java class that implements the ISecurityService or extends the default SecurityService, and place the class on the classpath in either `lib` or `web/WEB-INF/lib` folders. Then, in the `symmetric.properties` specify your class name for the security service.

```
security.service.class.name=org.jumpmind.symmetric.service.impl.SecurityService
```

Remember to specify your properties file when encrypting passwords, so it will use your custom ISecurityService.

```
sym -p symmetric.properties -e secret
```

5.8. Secure Transport

By specifying the "https" protocol for a URL, SymmetricDS will communicate over Secure Sockets Layer (SSL) for an encrypted transport. The following properties need to be set with "https" in the URL:

sync.url
This is the URL of the current node, so if you want to force other nodes to communicate over SSL with this node, you specify "https" in the URL.

registration.url
This is the URL where the node will connect for registration when it first starts up. To protect the registration with SSL, you specify "https" in the URL.

For incoming HTTPS connections, SymmetricDS depends on the webserver where it is deployed, so the webserver must be configured for HTTPS. As a standalone deployment, the "sym" launcher command provides options for enabling HTTPS support.

5.8.1. Sym Launcher

The "sym" launch command uses Jetty as an embedded web server. Using command line options, the web server can be told to listen for HTTP, HTTPS, or both.

sym --port 8080 --server

sym --secure-port 8443 --secure-server

sym --port 8080 --secure-port 8443 --mixed-server

5.8.2. Tomcat

If you deploy SymmetricDS to Apache Tomcat, it can be secured by editing the `TOMCAT_HOME/conf/server.xml` configuration file. There is already a line that can be uncommented and changed to the following:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
  maxThreads="150" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS"
  keystoreFile="/symmetric-ds-1.x.x/security/keystore" />
```

5.8.3. Keystores

When SymmetricDS connects to a URL with HTTPS, Java checks the validity of the certificate using the built-in trusted keystore located at `JRE_HOME/lib/security/cacerts`. The "sym" launcher command overrides the trusted keystore to use its own trusted keystore instead, which is located at `security/cacerts`. This keystore contains the certificate aliased as "sym" for use in testing and easing deployments. The trusted keystore can be overridden by specifying the `javax.net.ssl.trustStore` system property.

When SymmetricDS is run as a secure server with the "sym" launcher, it accepts incoming requests using the key installed in the keystore located at `security/keystore`. The default key is provided for convenience of testing, but should be re-generated for security.

5.8.4. Generating Keys

To generate new keys and install a server certificate, use the following steps:

1. Open a command prompt and navigate to the `security` subdirectory of your SymmetricDS installation on the server to which communication will be secured (typically the "root" or "central office" server).

2. Delete the old key pair and certificate.

   keytool -keystore keystore -delete -alias sym

   keytool -keystore cacerts -delete -alias sym

```
Enter keystore password:  changeit
```

3. Generate a new key pair. Note that the first name/last name (the "CN") must match the fully qualified hostname the client will be using to communcate to the server.

   keytool -keystore keystore -alias sym -genkey -keyalg RSA -validity 10950

```
Enter keystore password:  changeit
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  SymmetricDS
What is the name of your organization?
  [Unknown]:  JumpMind
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=SymmetricDS, O=JumpMind, L=Unknown, ST=Unknown, C=Unknown
correct?
  [no]:  yes

Enter key password for <sym>
```

```
        (RETURN if same as keystore password):
```

4. Export the certificate from the private keystore.

   keytool -keystore keystore -export -alias sym -rfc -file sym.cer

5. Install the certificate in the trusted keystore.

   keytool -keystore cacerts -import -alias sym -file sym.cer

6. Copy the cacerts file that is generated by this process to the `security` directory of each client's SymmetricDS installation.

5.9. Basic Authentication

SymmetricDS supports basic authentication for client and server nodes. To configure a client node to use basic authentication when communicating with a server node, specify the following startup parameters:

http.basic.auth.username
username for client node basic authentication. [Default:]

http.basic.auth.password
password for client node basic authentication. [Default:]

The SymmetricDS Standalone and Embedded Server also support basic authentication. This feature is enabled by specifying the basic authentication username and password using the following startup parameters:

embedded.webserver.basic.auth.username
username for basic authentication for an embedded server or standalone server node. [Default:]

embedded.webserver.basic.auth.password
password for basic authentication for an embedded server or standalone server node. [Default:]

If the server node is deployed to Tomcat or another application server as a WAR or EAR file, then basic authentication is setup with the standard configuration in the WEB.xml file.

5.10. Multi-Server Mode

SymmetricDS supports running multiple SymmetricDS instances that leverage the same web server in the same process. This mode can be turned on in the `web/WEB-INF/web.xml` file. By default, `multiServerMode` is turned off.

```
```

```
    <context-param>
        <param-name>multiServerMode</param-name>
        <param-value>true</param-value>
    </context-param>
```

When `multiServerMode` is turned on, SymmetricDS will initialize itself with an instance of a node for each properties file found in the `engines` directory. Each node will inherit common properties from `conf/symmetric.properties`. Each properties file must specify the minimum required properties to define a single node. In addition, the properties file is required to also specify a property, `engine.name`, that provides a unique name for the node's engine.

Chapter 6. Extending SymmetricDS

SymmetricDS may be extended via a plug-in like architecture where extension point interfaces may be implemented by a custom class and registered with the synchronization engine. All supported extension points extend the IExtensionPoint interface. The currently available extension points are documented in the following sections.

When the synchronization engine starts up, a Spring post processor searches the Spring ApplicationContext for any registered classes which implement IExtensionPoint. An IExtensionPoint designates whether it should be auto registered or not. If the extension point is to be auto registered then the post processor registers the known interface with the appropriate service.

The INodeGroupExtensionPoint interface may be optionally implemented to designate that auto registered extension points should only be auto registered with specific node groups.

```
/**
 * Only apply this extension point to the 'root' node group.
 */
 public String[] getNodeGroupIdsToApplyTo() {
     return new String[] { "root" };
 }
```

SymmetricDS will look for Spring configured extensions in the application Classpath by importing any Spring XML configuration files found matching the following pattern: `META-INF/services/symmetric-*-ext.xml`. When packaged in a jar file the `META-INF` directory should be at the root of the jar file. When packaged in a war file, the `META-INF` directory should be in the `WEB-INF/classes` directory.

6.1. IParameterFilter

Parameter values can be specified in code using a parameter filter. Note that there can be only one parameter filter per engine instance. The IParameterFilter replaces the deprecated IRuntimeConfig from prior releases.

```
public class MyParameterFilter
    implements IParameterFilter, INodeGroupExtensionPoint {

    /**
     * Only apply this filter to stores
     */
    public String[] getNodeGroupIdsToApplyTo() {
        return new String[] { "store" };
    }

    public String filterParameter(String key, String value) {
        // look up a store number from an already existing properties file.
        if (key.equals(ParameterConstants.EXTERNAL_ID)) {
            return StoreProperties.getStoreProperties().
              getProperty(StoreProperties.STORE_NUMBER);
        }
        return value;
```

```
    }

    public boolean isAutoRegister() {
        return true;
    }

}
```

6.2. IDataLoaderFilter

Data can be filtered as it is loaded into the target database. It can also be filtered when it is extracted from the source database. As data is loaded into the target database, a filter can change the data in a column or save it somewhere else. It can also specify by the return value of the function call that the data loader should continue on and load the data (by returning true) or ignore it (by returning false). One possible use of the filter might be to route credit card data to a secure database and blank it out as it loads into a less-restricted reporting database.

An IDataLoaderContext is passed to each of the callback methods. A new context is created for each synchronization. The context provides methods to lookup column indexes by column name, get table meta data, and access to old data if the `sync_column_level` flag is enabled. The context also provides a means to share data during a synchronization between different rows of data that are committed in a database transaction and are in the same channel. It does so by providing a context cache which can be populated by the extension point.

Many times the IDataLoaderFilter will be combined with the IBatchListener. The XmlPublisherFilter (in the `org.jumpmind.symmetric.ext` package) is a good example of using the combination of the two extension points in order to create XML messages to be published to JMS.

A class implementing the IDataLoaderFilter interface is injected onto the DataLoaderService in order to receive callbacks when data is inserted, updated, or deleted.

```
public MyFilter implements IDataLoaderFilter {

    public boolean isAutoRegister() {
        return true;
    }

    public boolean filterInsert(IDataLoaderContext context,
        String[] columnValues) {
        return true;
    }

    public boolean filterUpdate(IDataLoaderContext context,
        String[] columnValues, String[] keyValues) {
        return true;
    }

    public void filterDelete(IDataLoaderContext context,
        String[] keyValues) {
        return true;
    }

}
```

The filter class is specified as a Spring-managed bean. A custom Spring XML file is specified as follows in a jar at `META-INF/services/symmetric-myfilter-ext.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="myFilter" class="com.mydomain.MyFilter"/>

</beans>
```

6.3. ITableColumnFilter

Implement this extension point to filter out specific columns from use by the dataloader. Only one column filter may be added per target table.

6.4. IBatchListener

This extension point is called whenever a batch has completed loading but before the transaction has committed.

6.5. IAcknowledgeEventListener

Implement this extension point to receive callback events when a batch is acknowledged. The callback for this listener happens at the point of extraction.

6.6. IReloadListener

Implement this extension point to listen in and take action before or after a reload is requested for a Node. The callback for this listener happens at the point of extraction.

6.7. IExtractorFilter

This extension point is called after data has been extracted, but before it has been streamed. It has the ability to inspect each row of data to take some action and indicate, if necessary, that the row should not be streamed.

6.8. ISyncUrlExtension

This extension point is used to select an appropriate URL based on the URI provided in the `sync_url` column of `sym_node`.

To use this extension point configure the sync_url for a node with the protocol of ext://beanName. The beanName is the name you give the extension point in the extension xml file.

6.9. INodeIdGenerator

This extension point allows SymmetricDS users to implement their own algorithms for how node ids and passwords are generated or selected during the registration process. There may be only one node generator per SymmetricDS instance.

6.10. ITriggerCreationListener

Implement this extension point to get status callbacks during trigger creation.

6.11. IBatchAlgorithm

Implement this extension point and set the name of the Spring bean on the batch_algorithm column of the Channel table to use. This extension point gives fine grained control over how a channel is batched.

6.12. IDataRouter

Implement this extension point and set the name of the Spring bean on the router_type column of the Router table to use. This extension point gives the ability to programatically decide which nodes data should be routed to.

6.13. IHeartbeatListener

Implement this extension point to get callbacks during the heartbeat job.

6.14. IOfflineClientListener

Implement this extension point to get callbacks for offline events on client nodes.

6.15. IOfflineServerListener

Implement this extension point to get callbacks for offline events detected on a server node during monitoring of client nodes.

6.16. INodePasswordFilter

Implement this extension point to intercept the saving and rendering of the node password.

Chapter 7. Administration

7.1. Solving Synchronization Issues

By design, whenever SymmetricDS encounters an issue with a synchronization, the batch containing the error is marked as being in an error state, and all subsequent batches *for that particular channel to that particular node* are held and not synchronized until the error batch is resolved. SymmetricDS will retry the batch in error until the situation creating the error is resolved (or the data for the batch itself is changed).

7.1.1. Analyzing the Issue

The first step in analyzing the cause of a failed batch is to locate information about the data in the batch, starting with either OUTGOING_BATCH or INCOMING_BATCH. We'll use outgoing batches for the examples below. To locate batches in error, use:

```
select * from sym_outgoing_batch where error_flag=1;
```

Several useful pieces of information are available from this query:

- The batch number of the failed batch, available in column `BATCH_ID`.
- The node to which the batch is being sent, available in column `NODE_ID`.
- The channel to which the batch belongs, available in column `CHANNEL_ID`. All subsequent batches on this channel to this node will be held until the error condition is resolved.
- The specific data id in the batch which is causing the failure, available in column `FAILED_DATA_ID`.
- Any SQL message, SQL State, and SQL Codes being returned during the synchronization attempt, available in columns `SQL_MESSAGE`, `SQL_STATE`, and `SQL_CODE`, respectively.

Note
Using the `error_flag` on the batch table, as shown above, is more reliable than using the `status` column. The status column can change from 'ER' to a different status temporarily as the batch is retried.

Note
The query above will also show you any recent batches that were originally in error and were changed to be manually skipped. See the end of Section 7.1.2, Resolving the Issue (p. 65) for more details.

To get a full picture of the batch, you can query for information representing the complete list of all data changes associated with the failed batch by joining DATA and DATA_EVENT, such as:

```
select * from sym_data where data_id in
        (select data_id from sym_data_event where batch_id='XXXXXX');
```

where XXXXXX is the batch id of the failing batch.

This query returns a wealth of information about each data change in a batch, including:

- The table involved in each data change, available in column TABLE_NAME,
- The event type (Update [U], Insert [I], or Delete [D]), available in column EVENT_TYPE,
- A comma separated list of the new data and (optionally) the old data, available in columns ROW_DATA and OLD_DATA, respectively.
- The primary key data, available in column PK_DATA
- The channel id, trigger history information, transaction id if available, and other information.

More importantly, if you narrow your query to just the failed data id you can determine the exact data change that is causing the failure:

```
select * from sym_data where data_id in
        (select failed_data_id from sym_outgoing_batch where batch_id='XXXXX');
```

where XXXXXX is the batch that is failing.

The queries above usually yield enough information to be able to determine why a particular batch is failing. Common reasons a batch might be failing include:

- The schema at the destination has a column that is not nullable yet the source has the column defined as nullable and a data change was sent with the column as null.
- A foreign key constraint at the destination is preventing an insertion or update, which could be caused from data being deleted at the destination or the foreign key constraint is not in place at the source.
- The data size of a column on the destination is smaller than the data size in the source, and data that is too large for the destination has been synced.

7.1.2. Resolving the Issue

Once you have decided upon the cause of the issue, you'll have to decide the best course of action to fix the issue. If, for example, the problem is due to a database schema mismatch, one possible solution would be to alter the destination database in such a way that the SQL error no longer occurs. Whatever approach you take to remedy the issue, once you have made the change, on the next push or pull SymmetricDS will retry the batch and the channel's data will start flowing again.

If you have instead decided that the batch itself is wrong, or does not need synchronized, or you wish to remove a particular data change from a batch, you do have the option of changing the data associated

with the batch directly.

Warning

Be cautious when using the following two approaches to resolve synchronization issues. By far, the best approach to solving a synchronization error is to resolve what is truly causing the error at the destination database. Skipping a batch or removing a data id as discussed below should be your solution of last resort, since doing so results in differences between the source and destination databases.

Now that you've read the warning, if you *still* want to change the batch data itself, you do have several options, including:

- Causing SymmetricDS to skip the batch completely. This is accomplished by setting the batch's status to 'OK', as in:

```
update sym_outgoing_batch set status='OK' where batch_id='XXXXXX'
```

  where XXXXXX is the failing batch. On the next pull or push, SymmetricDS will skip this batch since it now thinks the batch has already been synchronized. Note that you can still distinguish between successful batches and ones that you've artificially marked as 'OK', since the `error_flag` column on the failed batch will still be set to '1' (in error).

- Removing the failing data id from the batch by deleting the corresponding row in DATA_EVENT. Eliminating the data id from the list of data ids in the batch will cause future synchronization attempts of the batch to no longer include that particular data change as part of the batch. For example:

```
delete from sym_data_event where batch_id='XXXXXX' and data_id='YYYYYY'
```

  where XXXXXX is the failing batch and YYYYYY is the data id to longer be included in the batch.

7.2. Changing Triggers

A trigger row may be updated using SQL to change a synchronization definition. SymmetricDS will look for changes each night or whenever the Sync Triggers Job is run (see below). For example, a change to place the table `price_changes` into the price channel would be accomplished with the following statement:

```
update SYM_TRIGGER
set channel_id = 'price',
    last_update_by = 'jsmith',
    last_update_time = current_timestamp
where source_table_name = 'price_changes';
```

All configuration should be managed centrally at the registration node. If enabled, configuration changes will be synchronized out to client nodes. When trigger changes reach the client nodes the Sync Triggers Job will run automatically.

Centrally, the trigger changes will not take effect until the Sync Triggers Job runs. Instead of waiting for the Sync Triggers Job to run overnight after making a Trigger change, you can invoke the syncTriggers() method over JMX or simply restart the SymmetricDS server. A complete record of trigger changes is kept in the table TRIGGER_HIST, which was discussed in Section 5.2.3, Sync Triggers Job (p. 46) .

7.3. Re-synchronizing Data

There may be times where you find you need to re-send or re-synchronize data when the change itself was not captured. This could be needed, for example, if the data changes occurred prior to SymmetricDS placing triggers on the data tables themselves, or if the data at the destination was accidentally deleted, or for some other reason. Two approaches are commonly taken to re-send the data, both of which are discussed below.

Important

Be careful when re-sending data using either of these two techniques. Be sure you are only sending the rows you intend to send and, more importantly, be sure to re-send the data in a way that won't cause foreign key constraint issues at the destination. In other words, if more than one table is involved, be sure to send any tables which are referred to by other tables by foreign keys first. Otherwise, the channel's synchronization will block because SymmetricDS is unable to insert or update the row because the foreign key relationship refers to a non-existent row in the destination!

One possible approach would be to "touch" the rows in individual tables that need re-sent. By "touch", we mean to alter the row data in such a way that SymmetricDS detects a data change and therefore includes the data change in the batching and synchronizing steps. Note that you have to change the data in some meaningful way (e.g., update a time stamp); setting a column to its current value is not sufficient (by default, if there's not an actual data value change SymmetricDS won't treat the change as something which needs synched.

A second approach would be to take advantage of SymmetricDS built-in functionality by simulating a partial "initial load" of the data. The approach is to manually create "reload" events in DATA for the necessary tables, thereby resending the desired rows for the given tables. Again, foreign key constraints must be kept in mind when creating these reload events. These reload events are created in the source database itself, and the necessary table, trigger-router combination, and channel are included to indicate the direction of synchronization.

To create a reload event, you create a DATA row, using:

- data_id: null
- table_name: name of table to be sent
- event_type: 'R', for reload
- row_data: a "where" clause (minus the word 'where') which defines the subset of rows from the table to be sent. To send all rows, one can use 1=1 for this value.

- pk_data: null
- old_data: null
- trigger_hist_id: use the id of the most recent entry (i.e., max(trigger_hist_id)) in TRIGGER_HIST for the trigger-router combination for your table and router.
- channel_id: the channel in which the table is routed
- transaction_id: pick a value, for example '1'
- source_node_id: null
- external_data: null
- create_time: current_timestamp

By way of example, take our retail hands-on tutorial covered in Chapter 2, *Hands-on Tutorial* (p. 7) . Let's say we need to re-send a particular sales transaction from the store to corp over again because we lost the data in corp due to an overzealous delete. For the tutorial, all transaction-related tables start with `sale_`, use the `sale_transaction` channel, and are routed using the `store_corp_identity` router. In addition, the trigger-routers have been set up with an initial load order based on the necessary foreign key relationships (i.e., transaction tables which are "parents" have a lower initial load order than those of their "children"). An insert statement that would create the necessary "reload" events (three in this case, one for each table) would be as follows (where MISSING_ID is changed to the needed transaciton id):

```
insert into sym_data (
    select null, t.source_table_name, 'R', 'tran_id=''MISSING-ID''', null, null,
            h.trigger_hist_id, t.channel_id, '1', null, null, current_timestamp
        from sym_trigger t inner join sym_trigger_router tr on
            t.trigger_id=tr.trigger_id inner join sym_trigger_hist h on
            h.trigger_hist_id=(select max(trigger_hist_id) from sym_trigger_hist
                where trigger_id=t.trigger_id)
    where channel_id='sale_transaction' and
        tr.router_id like 'store_corp_identity' and
        (t.source_table_name like 'sale_%')
    order by tr.initial_load_order asc);
```

This insert statement generates three rows, one for each configured sale table. It uses the most recent trigger history id for the corresponding table. Finally, it takes advantage of the initial load order for each trigger-router to create the three rows in the correct order (the order corresponding to the order in which the tables would have been initial loaded).

7.4. Changing Configuration

The configuration of your system as defined in the `sym_*` tables may be modified at runtime. By default, any changes made to the `sym_*` tables (with the exception of `sym_node`) should be made at the registration

server. The changes will be synchronized out to the leaf nodes by SymmetricDS triggers that are automatically created on the tables.

If this behavior is not desired, the feature can be turned off using a parameter. Custom triggers may be added to the `sym_*` tables when the auto syncing feature is disabled.

7.5. Logging Configuration

The standalone SymmetricDS installation uses Log4J for logging. The configuration file is `conf/log4j.xml`. The `log4j.xml` file has hints as to what logging can be enabled for useful, finer-grained logging.

SymmetricDS proxies all of its logging through Commons Logging. When deploying to an application server, if Log4J is not being leveraged, then the general rules for for Commons Logging apply.

7.6. Java Management Extensions

Monitoring and administrative operations can be performed using Java Management Extensions (JMX). SymmetricDS uses MX4J to expose JMX attributes and operations that can be accessed from the built-in web console, Java's jconsole, or an application server. By default, the web management console can be opened from the following address:

```
http://localhost:31416/
```

Using the Java jconsole command, SymmetricDS is listed as a local process named SymmetricLauncher. In jconsole, SymmetricDS appears under the MBeans tab under then name defined by the `engine.name` property. The default value is SymmetricDS.

The management interfaces under SymmetricDS are organized as follows:

- Node - administrative operations
- Incoming - statistics about incoming batches
- Outgoing - statistics about outgoing batches
- Parameters - access to properties set through the parameter service
- Notifications - setting thresholds and receiving notifications

7.7. Temporary Files

SymmetricDS creates temporary extraction and data load files with the CSV payload of a synchronization when the value of the `stream.to.file.threshold.bytes` SymmetricDS property has been reached. Before reaching the threshold, files are streamed to/from memory. The default threshold value is 32,767 bytes. This feature may be turned off by setting the `stream.to.file.enabled` property to false.

SymmetricDS creates these temporary files in the directory specified by the `java.io.tmpdir` Java System property. When SymmmetricDS starts up, stranded temporary files are aways cleaned up. Files will only be stranded if the SymmetricDS engine is force killed.

The location of the temporary directory may be changed by setting the Java System property passed into the Java program at startup. For example,

```
-Djava.io.tmpdir=/home/.symmetricds/tmp
```

7.8. Database Purging

Purging is the act of cleaning up captured data that is no longer needed in SymmetricDS's runtime tables. Data is purged through delete statements by the *Purge Job*. Only data that has been successfully synchronized will be purged. Purged tables include:

- DATA
- DATA_EVENT
- OUTGOING_BATCH
- INCOMING_BATCH
- DATA_GAP
- NODE_HOST_STATS
- NODE_HOST_CHANNEL_STATS
- NODE_HOST_JOB_STATS

The purge job is enabled by the `start.purge.job` SymmetricDS property. The job runs periodically according to the `job.purge.period.time.ms` property. The default period is to run every ten minutes.

Two retention period properties indicate how much history SymmetricDS will retain before purging. The `purge.retention.minutes` property indicates the period of history to keep for synchronization tables. The default value is 5 days. The `statistic.retention.minutes` property indicates the period of history to keep for statistics. The default value is also 5 days.

The purge properties should be adjusted according to how much data is flowing through the system and the amount of storage space the database has. For an initial deployment it is recommended that the purge properties be kept at the defaults, since it is often helpful to be able to look at the captured data in order to triage problems and profile the synchronization patterns. When scaling up to more nodes, it is recomended that the purge parameters be scaled back to 24 hours or less.

Appendix A. Data Model

What follows is the complete SymmetricDS data model. Note that all tables are prepended with a configurable prefix so that multiple instances of SymmetricDS may coexist in the same database. The default prefix is *sym_*.

SymmetricDS configuration is entered by the user into the data model to control the behavior of what data is synchronized to which nodes.

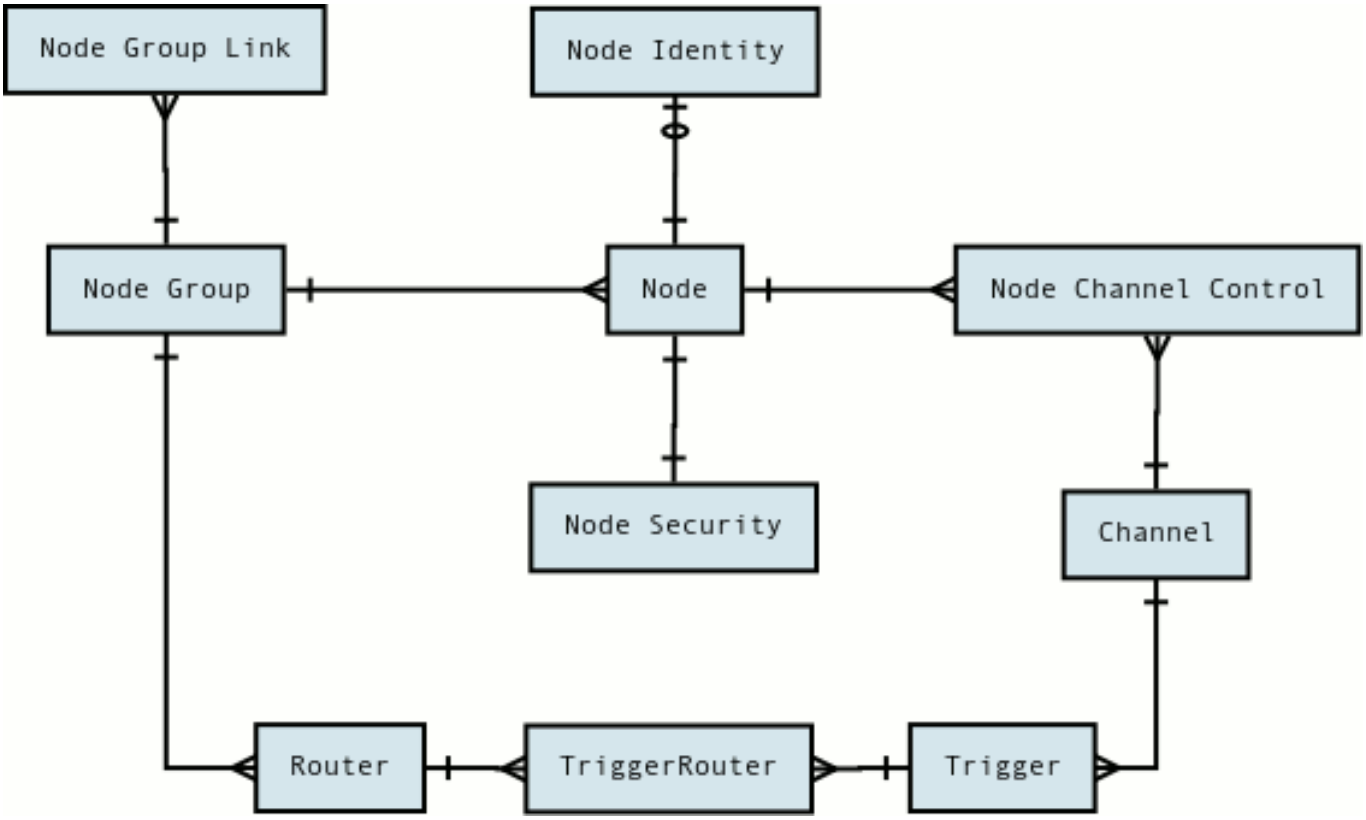


Figure A.1. Configuration Data Model

At runtime, the configuration is used to capture data changes and route them to nodes. The data changes are placed together in a single unit called a batch that can be loaded by another node. Outgoing batches are delivered to nodes and acknowledged. Incoming batches are received and loaded. History is recorded for batch status changes and statistics.

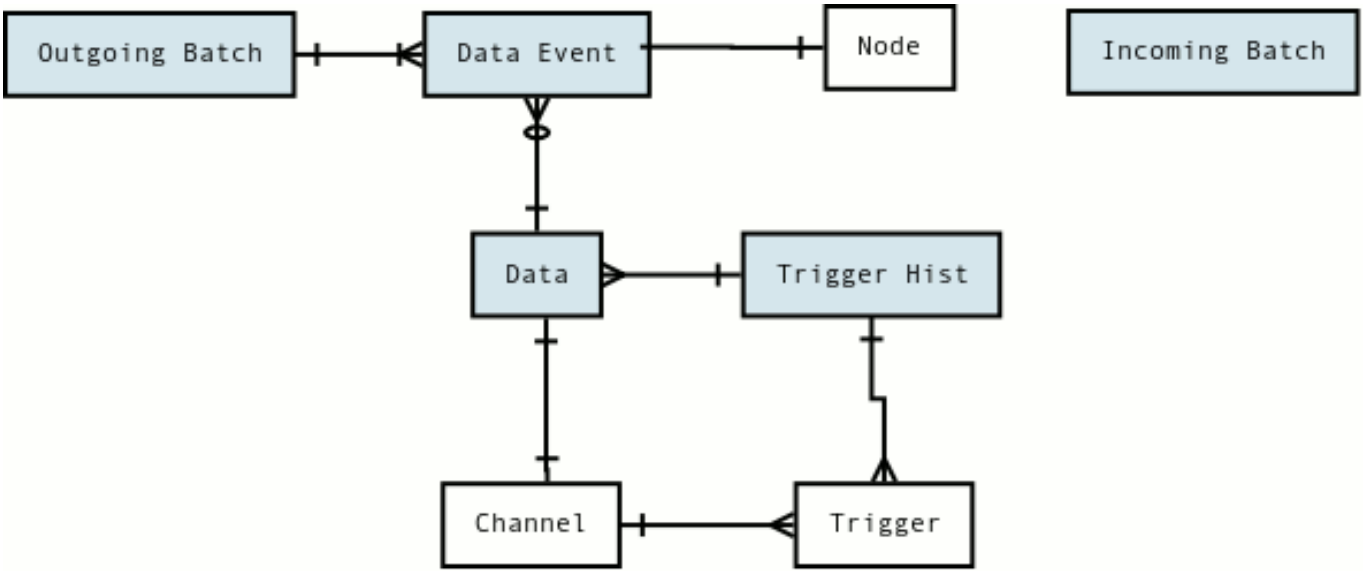


Figure A.2. Runtime Data Model

A.1. NODE

Representation of an instance of SymmetricDS that synchronizes data with one or more additional nodes. Each node has a unique identifier (nodeId) that is used when communicating, as well as a domain-specific identifier (externalId) that provides context within the local system.

Table A.1. NODE

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	A unique identifier for a node.
NODE_GROUP_ID	VARCHAR (50)			X	The node group that this node belongs to, such as 'store'.
EXTERNAL_ID	VARCHAR (50)			X	A domain-specific identifier for context within the local system. For example, the retail store number.
SYNC_ENABLED	INTEGER (1)	0			Indicates whether this node should be sent synchronization. Disabled nodes are ignored by the triggers, so no entries are made in data_event for the node.
SYNC_URL	VARCHAR (255)				The URL to contact the node for synchronization.
SCHEMA_VERSION	VARCHAR (50)				The version of the database schema this node manages. Useful for specifying synchronization by version.
SYMMETRIC_VERSION	VARCHAR				The version of SymmetricDS running at this

Name	Type / Size	Default	PK FK	not null	Description
	(50)				node.
DATABASE_TYPE	VARCHAR (50)				The database product name at this node as reported by JDBC.
DATABASE_VERSION	VARCHAR (50)				The database product version at this node as reported by JDBC.
HEARTBEAT_TIME	TIMESTAMP				The last timestamp when the node sent a heartbeat, which is attempted every ten minutes by default.
TIMEZONE_OFFSET	VARCHAR (6)				The timezone offset in RFC822 format at the time of the last heartbeat.
BATCH_TO_SEND_COUNT	INTEGER	0			The number of outgoing batches that have not yet been sent. This field is updated as part of the heartbeat job.
BATCH_IN_ERROR_COUNT	INTEGER	0			The number of outgoing batches that are in error at this node. This field is updated as part of the heartbeat job.
CREATED_AT_NODE_ID	VARCHAR (50)				The node_id of the node where this node was created. This is typically filled automatically with the node_id found in node_identity where registration was opened for the node.
DEPLOYMENT_TYPE	VARCHAR (50)				An indicator as to the type of SymmetricDS software that is running. Possible values are, but not limited to: engine, standalone, war, professional, mobile

A.2. NODE_SECURITY

Security features like node passwords and open registration flag are stored in the node_security table.

Table A.2. NODE_SECURITY

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	Unique identifier for a node.
NODE_PASSWORD	VARCHAR (50)			X	The password used by the node to prove its identity during synchronization.
REGISTRATION_ENABLED	INTEGER (1)	0			Indicates whether registration is open for this node. Re-registration may be forced for a node if this is set back to '1' in a parent database for the node_id that should be re-registred.
REGISTRATION_TIME	TIMESTAMP				The timestamp when this node was last registered.

Name	Type / Size	Default	PK FK	not null	Description
INITIAL_LOAD_ENABLED	INTEGER (1)	0			Indicates whether an initial load will be sent to this node.
INITIAL_LOAD_TIME	TIMESTAMP				The timestamp when this node started the initial load.
CREATED_AT_NODE_ID	VARCHAR (50)			X	The node_id of the node where this node was created. This is typically filled automatically with the node_id found in node_identity where registration was opened for the node.

A.3. NODE_IDENTITY

After registration, this table will have one row representing the identity of the node. For a root node, the row is entered by the user.

Table A.3. NODE_IDENTITY

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	Unique identifier for a node.

A.4. NODE_GROUP

A category of Nodes that synchronizes data with one or more NodeGroups. A common use of NodeGroup is to describe a level in a hierarchy of data synchronization.

Table A.4. NODE_GROUP

Name	Type / Size	Default	PK FK	not null	Description
NODE_GROUP_ID	VARCHAR (50)		PK	X	Unique identifier for a node group, usually named something meaningful, like 'store' or 'warehouse'.
DESCRIPTION	VARCHAR (255)				A description of this node group.

A.5. NODE_GROUP_LINK

A source node_group sends its data updates to a target NodeGroup using a pull, push, or custom

technique.

Table A.5. NODE_GROUP_LINK

Name	Type / Size	Default	PK FK	not null	Description
SOURCE_NODE_GROUP_ID	VARCHAR (50)		PK	X	The node group where data changes should be captured.
TARGET_NODE_GROUP_ID	VARCHAR (50)		PK	X	The node group where data changes will be sent.
DATA_EVENT_ACTION	CHAR (1)	W		X	The notification scheme used to send data changes to the target node group. (P = Push, W = Wait for Pull)

A.6. NODE_HOST

Representation of an physical workstation or server that is hosting the SymmetricDS software. In a clustered environment there may be more than one entry per node in this table.

Table A.6. NODE_HOST

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	A unique identifier for a node.
HOST_NAME	VARCHAR (60)		PK	X	The host name of a workstation or server. If more than one instance of SymmetricDS runs on the same server, then this value can be a 'server id' specified by -Druntime.symmetric.cluster.server.id
IP_ADDRESS	VARCHAR (50)				The ip address for the host.
OS_USER	VARCHAR (50)				The user SymmetricDS is running under
OS_NAME	VARCHAR (50)				The name of the OS
OS_ARCH	VARCHAR (50)				The hardware architecture of the OS
OS_VERSION	VARCHAR (50)				The version of the OS
AVAILABLE_PROCESSORS	INTEGER	0			The number of processors available to use.
FREE_MEMORY_BYTES	BIGINT	0			The amount of free memory available to the JVM.

Name	Type / Size	Default	PK FK	not null	Description
TOTAL_MEMORY_BYTES	BIGINT	0			The amount of total memory available to the JVM.
MAX_MEMORY_BYTES	BIGINT	0			The max amount of memory available to the JVM.
JAVA_VERSION	VARCHAR (50)				The version of java that SymmetricDS is running as.
JAVA_VENDOR	VARCHAR (255)				The vendor of java that SymmetricDS is running as.
SYMMETRIC_VERSION	VARCHAR (50)				The version of SymmetricDS running at this node.
TIMEZONE_OFFSET	VARCHAR (6)				The timezone offset in RFC822 format at the time of the last heartbeat.
HEARTBEAT_TIME	TIMESTAMP				The last timestamp when the node sent a heartbeat, which is attempted every ten minutes by default.
LAST_RESTART_TIME	TIMESTAMP			X	Timestamp when this instance was last restarted.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.

A.7. NODE_HOST_CHANNEL_STATS

Table A.7. NODE_HOST_CHANNEL_STATS

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	A unique identifier for a node.
HOST_NAME	VARCHAR (60)		PK	X	The host name of a workstation or server. If more than one instance of SymmetricDS runs on the same server, then this value can be a 'server id' specified by -Druntime.symmetric.cluster.server.id
CHANNEL_ID	VARCHAR (20)		PK	X	The channel_id of the channel that data changes will flow through.
START_TIME	TIMESTAMP		PK	X	
END_TIME	TIMESTAMP		PK	X	
DATA_ROUTED	BIGINT	0			Indicate the number of data rows that have been routed during this period.
DATA_UNROUTED	BIGINT	0			
DATA_EVENT_INSERTED	BIGINT	0			Indicate the number of data rows that have been

Name	Type / Size	Default	PK FK	not null	Description
					routed during this period.
DATA_EXTRACTED	BIGINT	0			
DATA_BYTES_EXTRACTED	BIGINT	0			
DATA_EXTRACTED_ERRORS	BIGINT	0			
DATA_BYTES_SENT	BIGINT	0			
DATA_SENT	BIGINT	0			
DATA_SENT_ERRORS	BIGINT	0			
DATA_LOADED	BIGINT	0			
DATA_BYTES_LOADED	BIGINT	0			
DATA_LOADED_ERRORS	BIGINT	0			

A.8. NODE_HOST_STATS

Table A.8. NODE_HOST_STATS

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	A unique identifier for a node.
HOST_NAME	VARCHAR (60)		PK	X	The host name of a workstation or server. If more than one instance of SymmetricDS runs on the same server, then this value can be a 'server id' specified by -Druntime.symmetric.cluster.server.id
START_TIME	TIMESTAMP		PK	X	
END_TIME	TIMESTAMP		PK	X	
RESTARTED	BIGINT	0		X	Indicate that a restart occurred during this period.
NODES_PULLED	BIGINT	0			
TOTAL_NODES_PULL_TIME	BIGINT	0			
NODES_PUSHED	BIGINT	0			
TOTAL_NODES_PUSH_TIME	BIGINT	0			
NODES_REJECTED	BIGINT	0			
NODES_REGISTERED	BIGINT	0			
NODES_LOADED	BIGINT	0			
NODES_DISABLED	BIGINT	0			

Name	Type / Size	Default	PK FK	not null	Description
PURGED_DATA_ROWS	BIGINT	0			
PURGED_DATA_EVENT_ROWS	BIGINT	0			
PURGED_BATCH_OUTGOING_ROWS	BIGINT	0			
PURGED_BATCH_INCOMING_ROWS	BIGINT	0			
TRIGGERS_CREATED_COUNT	BIGINT				
TRIGGERS_REBUILT_COUNT	BIGINT				
TRIGGERS_REMOVED_COUNT	BIGINT				

A.9. NODE_HOST_JOB_STATS

Table A.9. NODE_HOST_JOB_STATS

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	A unique identifier for a node.
HOST_NAME	VARCHAR (60)		PK	X	The host name of a workstation or server. If more than one instance of SymmetricDS runs on the same server, then this value can be a 'server id' specified by -Druntime.symmetric.cluster.server.id
JOB_NAME	VARCHAR (50)		PK	X	
START_TIME	TIMESTAMP		PK	X	
END_TIME	TIMESTAMP		PK	X	
PROCESSED_COUNT	BIGINT	0			

A.10. CHANNEL

This table represents a category of data that can be synchronized independently of other channels. Channels allow control over the type of data flowing and prevents one type of synchronization from contending with another.

Table A.10. CHANNEL

Name	Type / Size	Default	PK FK	not null	Description
CHANNEL_ID	VARCHAR (20)		PK	X	A unique identifer, usually named something meaningful, like 'sales' or 'inventory'.
PROCESSING_ORDER	INTEGER	1		X	Order of sequence to process channel data.
MAX_BATCH_SIZE	INTEGER	1000		X	The maximum number of Data Events to process within a batch for this channel.
MAX_BATCH_TO_SEND	INTEGER	60		X	The maximum number of batches to send during a 'synchronization' between two nodes. A 'synchronization' is equivalent to a push or a pull. If there are 12 batches ready to be sent for a channel and max_batch_to_send is equal to 10, then only the first 10 batches will be sent.
MAX_DATA_TO_ROUTE	INTEGER	100000		X	The maximum number of data rows to route for a channel at a time.
EXTRACT_PERIOD_MILLIS	INTEGER	0		X	The minimum number of milliseconds allowed between attempts to extract data for targeted at a node_id.
ENABLED	INTEGER (1)	1		X	Indicates whether channel is enabled or not.
USE_OLD_DATA_TO_ROUTE	INTEGER (1)	1		X	Indicates whether to read the old data during routing.
USE_ROW_DATA_TO_ROUTE	INTEGER (1)	1		X	Indicates whether to read the row data during routing.
USE_PK_DATA_TO_ROUTE	INTEGER (1)	1		X	Indicates whether to read the pk data during routing.
CONTAINS_BIG_LOB	INTEGER (1)	0		X	Provides SymmetricDS a hint as to whether this channel will contain big lobs data. Some databases have shortcuts that SymmetricDS can take advantage of if it knows that the lob columns in sym_data aren't going to contain large lobs. The definition of how big a 'large' lob is will differ from database to database.
BATCH_ALGORITHM	VARCHAR (50)	default		X	The algorithm to use when batching data on this channel. Possible values are: 'default', 'transactional', and 'nontransactional'
DESCRIPTION	VARCHAR (255)				Description on the type of data carried in this channel.

A.11. NODE_CHANNEL_CTL

Used to ignore or suspend a channel. A channel that is ignored will have its data_events batched and they will immediately be marked as 'OK' without sending them. A channel that is suspended is skipped when batching data_events.

Table A.11. NODE_CHANNEL_CTL

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	Unique identifier for a node.
CHANNEL_ID	VARCHAR (20)		PK	X	The name of the channel_id that is being controlled.
SUSPEND_ENABLED	INTEGER (1)	0			Indicates if this channel is suspended, which prevents its Data Events from being batched.
IGNORE_ENABLED	INTEGER (1)	0			Indicates if this channel is ignored, which marks its Data Events as if they were actually processed.
LAST_EXTRACT_TIME	TIMESTAMP				Record the last time data was extract for a node and a channel.

A.12. NODE_GROUP_CHANNEL_WINDOW

An optional window of time for which a node group and channel will be active.

Table A.12. NODE_GROUP_CHANNEL_WINDOW

Name	Type / Size	Default	PK FK	not null	Description
NODE_GROUP_ID	VARCHAR (50)		PK	X	The node_group_id that this window applies to.
CHANNEL_ID	VARCHAR (20)		PK	X	The channel_id that this window applies to.
START_TIME	TIME		PK	X	The start time for the active window.
END_TIME	TIME		PK	X	The end time for the active window. Note that if the end_time is less than the start_time then the window crosses a day boundary.
ENABLED	INTEGER (1)	0		X	Enable this window. If this is set to '0' then this window is ignored.

A.13. TRIGGER

Configures database triggers that capture changes in the database. Configuration of which triggers are generated for which tables is stored here. Triggers are created in a node's database if the source_node_group_id of a router is mapped to a row in this table.

Table A.13. TRIGGER

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_ID	VARCHAR (50)		PK	X	Unique identifier for a trigger.
SOURCE_CATALOG_NAME	VARCHAR (50)				Optional name for the catalog the configured table is in.
SOURCE_SCHEMA_NAME	VARCHAR (50)				Optional name for the schema a configured table is in.
SOURCE_TABLE_NAME	VARCHAR (50)			X	The name of the source table that will have a trigger installed to watch for data changes.
CHANNEL_ID	VARCHAR (20)			X	The channel_id of the channel that data changes will flow through.
SYNC_ON_UPDATE	INTEGER (1)	1		X	Whether or not to install an update trigger.
SYNC_ON_INSERT	INTEGER (1)	1		X	Whether or not to install an insert trigger.
SYNC_ON_DELETE	INTEGER (1)	1		X	Whether or not to install an delete trigger.
SYNC_ON_INCOMING_BATCH	INTEGER (1)	0		X	Whether or not an incoming batch that loads data into this table should cause the triggers to capture data_events. Be careful turning this on, because an update loop is possible.
NAME_FOR_UPDATE_TRIGGER	VARCHAR (50)				Override the default generated name for the update trigger.
NAME_FOR_INSERT_TRIGGER	VARCHAR (50)				Override the default generated name for the insert trigger.
NAME_FOR_DELETE_TRIGGER	VARCHAR (50)				Override the default generated name for the delete trigger.
SYNC_ON_UPDATE_CONDITION	LONGVARCHAR				Specify a condition for the update trigger firing using an expression specific to the database.
SYNC_ON_INSERT_CONDITION	LONGVARCHAR				Specify a condition for the insert trigger firing using an expression specific to the database.
SYNC_ON_DELETE_CONDITION	LONGVARCHAR				Specify a condition for the delete trigger firing using an expression specific to the database.
EXTERNAL_SELECT	LONGVARCHAR				Specify a SQL select statement that returns a single result. It will be used in the generated database trigger to populate the EXTERNAL_DATA field on the data table.
TX_ID_EXPRESSION	LONGVARCHAR				Override the default expression for the transaction identifier that groups the data changes that were committed together.
EXCLUDED_COLUMN_NAMES	LONGVARCHAR				Specify a comma-delimited list of columns that should not be synchronized from this table. Note that if a primary key is found in this list, it will be ignored.

Name	Type / Size	Default	PK FK	not null	Description
USE_STREAM_LOBS	INTEGER (1)	0		X	Specifies whether to capture lob data as the trigger is firing or to stream lob columns from the source tables using callbacks during extraction. A value of 1 indicates to stream from the source via callback; a value of 0, lob data is captured by the trigger.
USE_CAPTURE_LOBS	INTEGER (1)	0		X	Provides a hint as to whether this trigger will capture big lobs data. If set to 1 every effort will be made during data capture in trigger and during data selection for initial load to use lob facilities to extract and store data in the database.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.14. ROUTER

Configure a type of router from one node group to another. Note that routers are mapped to triggers through trigger_routers.

Table A.14. ROUTER

Name	Type / Size	Default	PK FK	not null	Description
ROUTER_ID	VARCHAR (50)		PK	X	Unique description of a specific router
TARGET_CATALOG_NAME	VARCHAR (50)				Optional name for the catalog a target table is in. Only use this if the target table is not in the default catalog.
TARGET_SCHEMA_NAME	VARCHAR (50)				Optional name of the schema a target table is in. On use this if the target table is not in the default schema.
TARGET_TABLE_NAME	VARCHAR (50)				Optional name for a target table. Only use this if the target table name is different than the source.
SOURCE_NODE_GROUP_ID	VARCHAR (50)			X	Routers with this node_group_id will install triggers that are mapped to this router.
TARGET_NODE_GROUP_ID	VARCHAR (50)			X	The node_group_id for nodes to route data to. Note that routing can be further narrowed down by the configured router_type and router_expression.
ROUTER_TYPE	VARCHAR				The name of a specific type of router. Out of

Name	Type / Size	Default	PK FK	not null	Description
	(50)				the box routers are 'default','column','bsh', and 'subselect.' Custom routers can be configured as extension points.
ROUTER_EXPRESSION	LONGVARCHAR				An expression that is specific to the type of router that is configured in router_type. See the documentation for each router for more details.
SYNC_ON_UPDATE	INTEGER (1)	1		X	Flag that indicates that this router should route updates.
SYNC_ON_INSERT	INTEGER (1)	1		X	Flag that indicates that this router should route inserts.
SYNC_ON_DELETE	INTEGER (1)	1		X	Flag that indicates that this router should route deletes.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.15. TRIGGER_ROUTER

Map a trigger to a router.

Table A.15. TRIGGER_ROUTER

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_ID	VARCHAR (50)		PK	X	The id of a trigger.
ROUTER_ID	VARCHAR (50)		PK	X	The id of a router.
INITIAL_LOAD_ORDER	INTEGER	1		X	Order sequence of this table when an initial load is sent to a node.
INITIAL_LOAD_SELECT	LONGVARCHAR				Optional expression that can be used to pair down the data selected from a table during the initial load process.
PING_BACK_ENABLED	INTEGER (1)	0		X	When enabled, the node will route data that originated from a node back to that node. This attribute is only effective if sync_on_incoming_batch is set to 1.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.

Name	Type / Size	Default	PK FK	not null	Description
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.16. PARAMETER

Provides a way to manage most SymmetricDS settings in the database.

Table A.16. PARAMETER

Name	Type / Size	Default	PK FK	not null	Description
EXTERNAL_ID	VARCHAR (50)		PK	X	Target the parameter at a specific external id. To target all nodes, use the value of 'ALL.'
NODE_GROUP_ID	VARCHAR (50)		PK	X	Target the parameter at a specific node group id. To target all groups, use the value of 'ALL.'
PARAM_KEY	VARCHAR (80)		PK	X	The name of the parameter.
PARAM_VALUE	LONGVARCHAR				The value of the parameter.

A.17. REGISTRATION_REDIRECT

Provides a way for a centralized registration server to redirect registering nodes to their prospective parent node in a multi-tiered deployment.

Table A.17. REGISTRATION_REDIRECT

Name	Type / Size	Default	PK FK	not null	Description
REGISTRANT_EXTERNAL_ID	VARCHAR (50)		PK	X	Maps the external id of a registration request to a different parent node.
REGISTRATION_NODE_ID	VARCHAR (50)			X	The node_id of the node that a registration request should be redirected to.

A.18. REGISTRATION_REQUEST

Audits when a node registers or attempts to register.

Table A.18. REGISTRATION_REQUEST

Name	Type / Size	Default	PK FK	not null	Description
NODE_GROUP_ID	VARCHAR (50)				The node group that this node belongs to, such as 'store'.
EXTERNAL_ID	VARCHAR (50)				A domain-specific identifier for context within the local system. For example, the retail store number.
STATUS	CHAR (2)			X	The current status of the registration attempt. Valid statuses are NR (not registered), IG (ignored), OK (sucessful)
HOST_NAME	VARCHAR (60)			X	The host name of a workstation or server. If more than one instance of SymmetricDS runs on the same server, then this value can be a 'server id' specified by -Druntime.symmetric.cluster.server.id
IP_ADDRESS	VARCHAR (21)			X	The ip address for the host.
ATTEMPT_COUNT	INTEGER	0			The number of registration attempts.
REGISTERED_NODE_ID	VARCHAR (50)				A unique identifier for a node.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.19. TRIGGER_HIST

A history of a table's definition and the trigger used to capture data from the table. When a database trigger captures a data change, it references a trigger_hist entry so it is possible to know which columns the data represents. trigger_hist entries are made during the sync trigger process, which runs at each startup, each night in the syncTriggersJob, or any time the syncTriggers() JMX method is manually invoked. A new entry is made when a table definition or a trigger definition is changed, which causes a database trigger to be created or rebuilt.

Table A.19. TRIGGER_HIST

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_HIST_ID	INTEGER		PK	X	Unique identifier for a trigger_hist entry
TRIGGER_ID	VARCHAR (50)			X	Unique identifier for a trigger
SOURCE_TABLE_NAME	VARCHAR (50)			X	The name of the source table that will have a trigger installed to watch for data changes.

Name	Type / Size	Default	PK FK	not null	Description
SOURCE_CATALOG_NAME	VARCHAR (50)				The catalog name where the source table resides.
SOURCE_SCHEMA_NAME	VARCHAR (50)				The schema name where the source table resides.
NAME_FOR_UPDATE_TRIGGER	VARCHAR (50)			X	The name used when the insert trigger was created.
NAME_FOR_INSERT_TRIGGER	VARCHAR (50)			X	The name used when the update trigger was created.
NAME_FOR_DELETE_TRIGGER	VARCHAR (50)			X	The name used when the delete trigger was created.
TABLE_HASH	BIGINT			X	A hash of the table definition, used to detect changes in the definition.
TRIGGER_ROW_HASH	BIGINT			X	A hash of the trigger definition. If changes are detected to the values that affect a trigger definition, then the trigger will be regenerated.
COLUMN_NAMES	LONGVARCHAR			X	The column names defined on the table. The column names are stored in comma-separated values (CSV) format.
PK_COLUMN_NAMES	LONGVARCHAR			X	The primary key column names defined on the table. The column names are stored in comma-separated values (CSV) format.
LAST_TRIGGER_BUILD_REASON	CHAR (1)			X	The following reasons for a change are possible: New trigger that has not been created before (N); Schema changes in the table were detected (S); Configuration changes in Trigger (C); Trigger was missing (T).
ERROR_MESSAGE	LONGVARCHAR				Record any errors or warnings that occurred when attempting to build the trigger.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
INACTIVE_TIME	TIMESTAMP				The date and time when a trigger was inactivated.

A.20. DATA

The captured data change that occurred to a row in the database. Entries in data are created by database triggers.

Table A.20. DATA

Name	Type / Size	Default	PK FK	not null	Description
DATA_ID	INTEGER		PK	X	Unique identifier for a data.

Name	Type / Size	Default	PK FK	not null	Description
TABLE_NAME	VARCHAR (50)			X	The name of the table in which a change occurred that this entry records.
EVENT_TYPE	CHAR (1)			X	The type of event captured by this entry. For triggers, this is the change that occurred, which is 'I' for insert, 'U' for update, or 'D' for delete. Other events include: 'R' for reloading the entire table (or subset of the table) to the node; 'S' for running dynamic SQL at the node, which is used for adhoc administration.
ROW_DATA	LONGVARCHAR				The captured data change from the synchronized table. The column values are stored in comma-separated values (CSV) format.
PK_DATA	LONGVARCHAR				The primary key values of the captured data change from the synchronized table. This data is captured for updates and deletes. The primary key values are stored in comma-separated values (CSV) format.
OLD_DATA	LONGVARCHAR				The captured data values prior to the update. The column values are stored in CSV format.
TRIGGER_HIST_ID	INTEGER			X	The foreign key to the trigger_hist entry that contains the primary key and column names for the table being synchronized.
CHANNEL_ID	VARCHAR (20)				The channel that this data belongs to, such as 'prices'
TRANSACTION_ID	VARCHAR (255)				An optional transaction identifier that links multiple data changes together as the same transaction.
SOURCE_NODE_ID	VARCHAR (50)				If the data was inserted by a SymmetricDS data loader, then the id of the source node is record so that data is not re-routed back to it.
EXTERNAL_DATA	VARCHAR (50)				A field that can be populated by a trigger that uses the EXTERNAL_SELECT
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.21. DATA_REF

Used only when routing.data.reader.type is set to 'ref.' Table that tracks the last known data_id that has been processed. This table is used so that joins to find unprocessed data can be better optimized.

Table A.21. DATA_REF

Name	Type / Size	Default	PK FK	not null	Description
REF_DATA_ID	INTEGER		PK	X	The data_id that can be used to limit the search of the data table to rows that are greater than this data_id value.
REF_TIME	TIMESTAMP				The time when the ref_data_id was recorded. It is used as the base time to calculate timeouts for gaps in the data_ids.

A.22. DATA_GAP

Used only when routing.data.reader.type is set to 'gap.' Table that tracks gaps in the data table so that they may be processed efficiently, if data shows up. Gaps can show up in the data table if a database transaction is rolled back.

Table A.22. DATA_GAP

Name	Type / Size	Default	PK FK	not null	Description
START_ID	INTEGER		PK	X	The first missing data_id from the data table where a gap is detected. This could be the last data_id inserted plus one.
END_ID	INTEGER		PK	X	The last missing data_id from the data table where a gap is detected. If the start_id is the last data_id inserted plus one, then this field is filled in with a -1.
STATUS	CHAR (2)				GP, SK, or FL. GP means there is a detected gap. FL means that the gap has been filled. SK means that the gap has been skipped either because the gap expired or because no database transaction was detected which means that no data will be committed to fill in the gap.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.
LAST_UPDATE_HOSTNAME	VARCHAR (255)				The host who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.23. DATA_EVENT

Represents routing of a data row to one or more nodes. Entries in data_event are created by database triggers.

Table A.23. DATA_EVENT

Name	Type / Size	Default	PK FK	not null	Description
DATA_ID	INTEGER		PK	X	Id of the data to be routed.
BATCH_ID	INTEGER	-1	PK	X	The node_id of the node that is to receive the data.
ROUTER_ID	VARCHAR (50)		PK	X	The router_id of the router that routed this data_event.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.24. OUTGOING_BATCH

Used for tracking the sending a collection of data to a node in the system. A new outgoing_batch is created and given a status of 'NE'. After sending the outgoing_batch to its target node, the status becomes 'SE'. The node responds with either a success status of 'OK' or an error status of 'ER'. An error while sending to the node also results in an error status of 'ER' regardless of whether the node sends that acknowledgement.

Table A.24. OUTGOING_BATCH

Name	Type / Size	Default	PK FK	not null	Description
BATCH_ID	INTEGER		PK	X	A unique id for the batch.
NODE_ID	VARCHAR (50)				The node that this batch is targeted at.
CHANNEL_ID	VARCHAR (20)				The channel that this batch is part of.
STATUS	CHAR (2)				The current status of the Batch can be currently routing (RE), newly created and ready for replication (NE), being queried from the database (QE), sent to a Node (SE), ready to be loaded (LD) and acknowledged as successful (OK) or error (ER).
LOAD_FLAG	INTEGER (1)	0			A flag that indicates that this batch is part of an initial load.
ERROR_FLAG	INTEGER (1)	0			A flag that indicates that this batch was in error during the last synchornization attempt.
BYTE_COUNT	BIGINT	0		X	The number of bytes that were sent as part of this batch.
EXTRACT_COUNT	BIGINT	0		X	The number of times this an attempt to extract this batch occurred.
SENT_COUNT	BIGINT	0		X	The number of times this batch was sent. A batch can be sent multiple times if an ACK is not received.
LOAD_COUNT	BIGINT	0		X	The number of times an attempt to load this

Name	Type / Size	Default	PK FK	not null	Description
					batch occurred.
DATA_EVENT_COUNT	BIGINT	0		X	The number of data_events that are part of this batch.
RELOAD_EVENT_COUNT	BIGINT	0		X	The number of reload events that are part of this batch.
INSERT_EVENT_COUNT	BIGINT	0		X	The number of insert events that are part of this batch.
UPDATE_EVENT_COUNT	BIGINT	0		X	The number of update events that are part of this batch.
DELETE_EVENT_COUNT	BIGINT	0		X	The number of delete events that are part of this batch.
OTHER_EVENT_COUNT	BIGINT	0		X	The number of other event types that are part of this batch. This includes any events types that are not a reload, insert, update or delete event type.
ROUTER_MILLIS	BIGINT	0		X	The number of milliseconds spent creating this batch.
NETWORK_MILLIS	BIGINT	0		X	The number of milliseconds spent transfering this batch across the network.
FILTER_MILLIS	BIGINT	0		X	The number of milliseconds spent in filters processing data.
LOAD_MILLIS	BIGINT	0		X	The number of milliseconds spent loading the data into the target database.
EXTRACT_MILLIS	BIGINT	0		X	The number of milliseconds spent extracting the data out of the source database.
SQL_STATE	VARCHAR (10)				For a status of error (ER), this is the XOPEN or SQL 99 SQL State.
SQL_CODE	INTEGER	0		X	For a status of error (ER), this is the error code from the database that is specific to the vendor.
SQL_MESSAGE	LONGVARCHAR				For a status of error (ER), this is the error message that describes the error.
FAILED_DATA_ID	BIGINT	0		X	For a status of error (ER), this is the data_id that was being processed when the batch failed.
LAST_UPDATE_HOSTNAME	VARCHAR (255)				The host name of the process that last did work on this batch.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a process last updated this entry.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.25. INCOMING_BATCH

The incoming_batch is used for tracking the status of loading an outgoing_batch from another node. Data is loaded and commited at the batch level. The status of the incoming_batch is either successful (OK) or error (ER).

Table A.25. INCOMING_BATCH

Name	Type / Size	Default	PK FK	not null	Description
BATCH_ID	INTEGER (50)		PK	X	The id of the outgoing_batch that is being loaded.
NODE_ID	VARCHAR (50)		PK	X	The node_id of the source of the batch being loaded.
CHANNEL_ID	VARCHAR (20)				The channel_id of the batch being loaded.
STATUS	CHAR (2)				The current status of the batch can be loading (LD), successfully loaded (OK), in error (ER) or skipped (SK)
ERROR_FLAG	INTEGER (1)	0			A flag that indicates that this batch was in error during the last synchornization attempt.
NETWORK_MILLIS	BIGINT	0		X	The number of milliseconds spent transfering this batch across the network.
FILTER_MILLIS	BIGINT	0		X	The number of milliseconds spent in filters processing data.
DATABASE_MILLIS	BIGINT	0		X	The number of milliseconds spent loading the data into the target database.
FAILED_ROW_NUMBER	BIGINT	0		X	For a status of error (ER), this is the data_id that was being processed when the batch failed.
BYTE_COUNT	BIGINT	0		X	The number of bytes that were sent as part of this batch.
STATEMENT_COUNT	BIGINT	0		X	The number of statements run to load this batch.
FALLBACK_INSERT_COUNT	BIGINT	0		X	The number of times an update was turned into an insert because the data was not already in the target database.
FALLBACK_UPDATE_COUNT	BIGINT	0		X	The number of times an insert was turned into an update because a data row already existed in the target database.
MISSING_DELETE_COUNT	BIGINT	0		X	THe number of times a delete did not effect the database because the row was already deleted.
SKIP_COUNT	BIGINT	0		X	The number of times a batch was sent and skipped because it had already been loaded according to incoming_batch
SQL_STATE	VARCHAR (10)				For a status of error (ER), this is the XOPEN or SQL 99 SQL State.
SQL_CODE	INTEGER	0		X	For a status of error (ER), this is the error code

Name	Type / Size	Default	PK FK	not null	Description
					from the database that is specific to the vendor.
SQL_MESSAGE	LONGVARCHAR				For a status of error (ER), this is the error message that describes the error.
LAST_UPDATE_HOSTNAME	VARCHAR (255)				The host name of the process that last did work on this batch.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a process last updated this entry.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.26. LOCK

Contains semaphores that are set when processes run, so that only one server can run a process at a time. Enable this feature by using the cluster.lock.during.xxxx parameters.

Table A.26. LOCK

Name	Type / Size	Default	PK FK	not null	Description
LOCK_ACTION	VARCHAR (50)		PK	X	The process that needs a lock.
LOCKING_SERVER_ID	VARCHAR (255)				The name of the server that currently has a lock. This is typically a host name, but it can be overridden using the -Druntime.symmetric.cluster.server.id=name System property.
LOCK_TIME	TIMESTAMP				The time a lock is aquired. Use the cluster.lock.timeout.ms to specify a lock timeout period.
LAST_LOCK_TIME	TIMESTAMP				Timestamp when a process last updated this entry.
LAST_LOCKING_SERVER_ID	VARCHAR (255)				The server id of the process that last did work on this batch.

A.27. TRANSFORM_TABLE

Defines a data loader transformation which can be used to map arbitrary tables and columns to other tables and columns.

Table A.27. TRANSFORM_TABLE

Name	Type / Size	Default	PK FK	not null	Description
TRANSFORM_ID	VARCHAR (50)		PK	X	Unique identifier of a specific transform.
SOURCE_NODE_GROUP_ID	VARCHAR (50)		PK	X	The node group where data changes should be captured.
TARGET_NODE_GROUP_ID	VARCHAR (50)		PK	X	The node group where data changes will be sent.
TRANSFORM_POINT	VARCHAR (10)			X	The point during the transport of captured data that a transform happens. Support values are EXTRACT or LOAD.
SOURCE_CATALOG_NAME	VARCHAR (128)				Optional name for the catalog the configured table is in.
SOURCE_SCHEMA_NAME	VARCHAR (128)				Optional name for the schema a configured table is in.
SOURCE_TABLE_NAME	VARCHAR (128)			X	The name of the source table that will be transformed.
TARGET_CATALOG_NAME	VARCHAR (128)				Optional name for the catalog a target table is in. Only use this if the target table is not in the default catalog.
TARGET_SCHEMA_NAME	VARCHAR (128)				Optional name of the schema a target table is in. Only use this if the target table is not in the default schema.
TARGET_TABLE_NAME	VARCHAR (128)				Optional name for a target table. Use this if the target table name is different than the source.
UPDATE_FIRST	INTEGER (1)	0			If true, the target actions are attempted as updates first, regardless of whether the source operation was an insert or an update.
DELETE_ACTION	VARCHAR (10)			X	An action to take upon delete of a row.
TRANSFORM_ORDER	INTEGER	1		X	Specifies the order in which to apply transforms if more than one target operation occurs.

A.28. TRANSFORM_COLUMN

Defines the column mappings and optional data transformation for a data loader transformation

Table A.28. TRANSFORM_COLUMN

Name	Type / Size	Default	PK FK	not null	Description
TRANSFORM_ID	VARCHAR (50)		PK	X	Unique identifier of a specific transform.
INCLUDE_ON	CHAR (1)	*	PK	X	Indicates whether this mapping is included

Name	Type / Size	Default	PK FK	not null	Description
					during an insert (I), update (U), delete (D) operation at the target based on the dml type at the source. A value of * represents the fact that you want to map the column for all operations.
TARGET_COLUMN_NAME	VARCHAR (128)		PK	X	Name of the target column.
SOURCE_COLUMN_NAME	VARCHAR (128)				Name of the source column.
PK	INTEGER (1)	0			Indicates whether this mapping defines a primary key to be used to identify the target row. At least one row must be defined as a pk for each transform_id.
TRANSFORM_TYPE	VARCHAR (50)	copy			The name of a specific type of transform. Custom transformers can be configured as extension points.
TRANSFORM_EXPRESSION	LONGVARCHAR				An expression that is specific to the type of transform that is configured in transform_type. See the documentation for each transformer for more details.
TRANSFORM_ORDER	INTEGER	1		X	Specifies the order in which to apply transforms if more than one target operation occurs.

Appendix B. Parameters

There are two kinds of parameters that can be used to configure the behavior of SymmetricDS: *Startup Parameters* and *Runtime Parameters*. Startup Parameters are required to be in a system property or a property file, while Runtime Parameters can also be found in the Parameter table from the database. Parameters are re-queried from their source at a configured interval and can also be refreshed on demand by using the JMX API. The following table shows the source of parameters and the hierarchy of precedence.

Table B.1. Parameter Locations

Location	Required	Description
symmetric-default.properties	Y	Packaged inside symmetric-ds.jar file. This file has all the default settings along with descriptions.
symmetric.properties	N	Provided by the end user on the classpath. The first symmetric.properties found on the classpath will be used.
symmetric.properties	N	Provided by the end user in the current system user's user.home directory.
named properties file 1	N	Provided by the end user as a Java system property (i.e. -Dsymmetric.override.properties.file.1=file://my.properties) or in the constructor of a `SymmetricEngine` .
named properties file 2	N	Provided by the end user as a Java system property (i.e. -Dsymmetric.override.properties.file.2=classpath://my.properties) or in the constructor of a `SymmetricEngine` .
Java System Properties	N	Any SymmetricDS property can be passed in as a -D property to the runtime. It will take precedence over any properties file property.
Parameter table	N	A table which contains SymmetricDS parameters. Parameters can be targeted at a specific node group and even at a specific external id. These settings will take precedence over all of the above.
IParameterFilter	N	An extension point which allows parameters to be sourced from another location or customized. These settings will take precedence over all of the above.

B.1. Startup Parameters

Startup parameters are read once from properties files and apply only during start up. The following properties are used:

db.jndi.name
The name of the database connection pool to use, which is registered in the JNDI directory tree of the application server. It is recommended that this DataSource is NOT transactional, because SymmetricDS will handle its own transactions. If NOT using a JNDI connection pool, you must provide information about the database connection using the `db.driver` , `db.url` , `db.user` , and `db.password` properties instead, which will create a pool of connections using the `db.pool` properties.

[Default:]

db.driver
The class name of the JDBC driver. If `db.jndi.name` is set, this property is ignored.
[Default: com.mysql.jdbc.Driver]

db.url
The JDBC URL used to connect to the database. If `db.jndi.name` is set, this property is ignored.
[Default: jdbc:mysql://localhost/symmetric]

db.user
The database username, which is used to login, create, and update SymmetricDS tables. To use an encrypted username, see Section 5.7, Encrypted Passwords (p. 54) . If `db.jndi.name` is set, this property is ignored. [Default: symmetric]

db.password
The password for the database user. To use an encrypted password, see Section 5.7, Encrypted Passwords (p. 54) . If `db.jndi.name` is set, this property is ignored. [Default:]

db.pool.initial.size
The initial size of the connection pool. If `db.jndi.name` is set, this property is ignored. [Default: 5]

db.pool.max.active
The maximum number of connections that will be allocated in the pool. If `db.jndi.name` is set, this property is ignored. [Default: 10]

db.pool.max.wait.millis
This is how long a request for a connection from the datasource will wait before giving up. If `db.jndi.name` is set, this property is ignored. [Default: 30000]

db.pool.min.evictable.idle.millis
This is how long a connection can be idle before it will be evicted. If `db.jndi.name` is set, this property is ignored. [Default: 120000]

db.spring.bean.name
The name of a Spring bean to use as the DataSource. If you want to use a different DataSource other than the provided DBCP version that SymmetricDS uses out of the box, you may set this to be the Spring bean name of your DataSource.

db.sql.query.timeout.seconds
The timeout in seconds for queries running on the database. [Default: 300]

db.tx.timeout.seconds
This is how long the default transaction time is. This needs to be fairly big to account for large data loads. [Default: 7200]

db.jdbc.streaming.results.fetch.size
This is the default fetch size for streaming result sets into memory from the database.
[Default: 1000]

db.default.schema

This is the schema that will be used for metadata lookup. Some dialect automatically figure this out using database specific SQL to get the current schema. [Default:]

db.metadata.ignore.case
Indicates that case should be ignored when looking up references to tables using the metadata api. [Default: true]

auto.config.database
If this is true, the configuration and runtime tables used by SymmetricDS are automatically created during startup. [Default: true]

auto.upgrade
If this is true, when symmetric starts up it will try to upgrade tables to latest version. [Default: true]

auto.sync.configuration
If this is true, create triggers for the SymmetricDS configuration table that will synchronize changes to node groups that pull from the node where this property is set. [Default: true]

https.allow.self.signed.certs
If this is true, a Symmetric client node to accept self signed certificates. [Default: true]

http.basic.auth.username
If specified, a Symmetric client node will use basic authentication when communicating with its server node using the given user name. [Default:]

http.basic.auth.password
If specified, the password used for basic authentication. [Default:]

embedded.webserver.basic.auth.username
If specified, the username for basic authentication for an embedded server or standalone server node. Specifying the username and password is all that's needed to enable basic authentication for an embedded server or standalone server node. [Default:]

embedded.webserver.basic.auth.password
If specified, the password for basic authentication for an embedded server or standalone server node. [Default:]

https.verified.server.names
A list of comma separated server names that will always verify when using https. This is useful if the URL's hostname and the server's identification hostname don't match exactly using the default rules for the JRE. A special value of "all" may be specified to allow all hostnames to verify. [Default:]

sync.table.prefix
When symmetric tables are created and accessed, this is the prefix to use for the table name. [Default: sym]

engine.name
This is the engine name. This should be set if you have more than one engine running in the same JVM. It is used to name the JMX management bean. [Default: Default]

start.push.job

Whether the push job is enabled for this node. [Default: true]

start.pull.job
Whether the pull job is enabled for this node. [Default: true]

start.purge.job
Whether the purge job is enabled for this node. [Default: true]

start.synctriggers.job
Whether the sync triggers job is enabled for this node. [Default: true]

start.heartbeat.job
Whether the heartbeat job is enabled for this node. The heartbeat job simply inserts an event to update the heartbeat_time column on the node table for the current node. [Default: true]

start.watchdog.job
Whether the watchdog job is enabled for this node. The watchdog job monitors child nodes to detect if they are offline. Refer to Section 6.15, IOfflineServerListener (p. 62) for more information.
[Default: true]

job.purge.period.time.ms
This is how often the purge job will be run. [Default: 600000]

job.statflush.period.time.ms
This is how often accumulated statistics will be flushed out to the database from memory.
[Default: 600000]

web.base.servlet.path
The base servlet path for when embedding SymmetricDS with in another web application.
[Default:]

B.2. Runtime Parameters

Runtime parameters are read periodically from properties files or the database. The following properties are used:

auto.registration
If this is true, registration is opened automatically for nodes requesting it. [Default: false]

auto.reload
If this is true, a reload is automatically sent to nodes when they register. [Default: false]

auto.update.node.values.from.properties
Update the node row in the database from the local properties during a heartbeat operation.
[Default: true]

http.concurrent.workers.max
This is the number of HTTP concurrent push/pull requests symmetric will accept. This is controlled by the NodeConcurrencyFilter. The maximum number of database connections in the database pool

should be set to twice this number.[Default: 20]

offline.node.detection.period.minutes
This is the minimum number of minutes that a child node has been offline before taking action. Refer to Section 6.15, IOfflineServerListener (p. 62) for more information. [Default: 120]

outgoing.batches.peek.ahead.window.after.max.size
This is the maximum number of events that will be peeked at to look for additional transaction rows after the max batch size is reached. The more concurrency in your db and the longer the transaction takes the bigger this value might have to be. [Default: 100]

incoming.batches.skip.duplicates
Whether or not to skip duplicate batches that are received. A duplicate batch is identified by the batch ID already existing in the incoming batch table. If this happens, it means an acknowledgement was lost due to failure or there is a bug. Accepting a duplicate batch in this case can mean overwriting data with old data. Another cause of duplicates is when the batch sequence number is reset, which might happen in a lab environement. Skipping a duplicate batch in this case would prevent data changes from loading. Generally, in a production envionment, this setting should be true. [Default: true]

num.of.ack.retries
This is the number of times we will attempt to send an ACK back to the remote node when pulling and loading data. [Default: 5]

time.between.ack.retries.ms
This is the amount of time to wait between trying to send an ACK back to the remote node when pulling and loading data. [Default: 5000]

dataextractor.enabled
Enable or disable all data extraction at a node for all channels other than the config channel.
[Default: true]

dataloader.enabled
Enable or disable all data loading at a node for all channels other than the config channel.
[Default: true]

dataloader.enable.fallback.update
If an insert is received, but the row already exists, then try an update instead. [Default: true]

dataloader.enable.fallback.insert
If an update is received, but it affects no rows, then try to insert instead. [Default: true]

dataloader.allow.missing.delete
If a delete is received, but it affects no rows, then continue. [Default: true]

cluster.server.id
Set this if you want to give your server a unique name to be used to identify which server did what action. Typically useful when running in a clustered environment. This is currently used by the ClusterService when locking for a node. [Default:]

cluster.lock.timeout.ms
Time limit of lock before it is considered abandoned and can be broken. [Default: 1800000]

cluster.lock.enabled
[Default: false]

initial.load.delete.first
Set this if tables should be purged prior to an initial load. [Default: false]

initial.load.create.first
Set this if tables (and their indexes) should be created prior to an initial load. [Default: false]

http.timeout.ms
Sets both the connection and read timeout on the internal HttpUrlConnection. [Default: 600000s]

http.compression
Whether or not to use compression over HTTP connections. Currently, this setting only affects the push connection of the source node. Compression on a pull is enabled using a filter in the web.xml for the PullServlet. [Default: true]

web.compression.disabled
Disable compression from occurring on Servlet communication. This property only affects the outbound HTTP traffic streamed by the PullServlet and PushServlet. [Default: false]

compression.level
Set the compression level this node will use when compressing synchronization payloads. Valid values include: NO_COMPRESSION = 0, BEST_SPEED = 1, BEST_COMPRESSION = 9, DEFAULT_COMPRESSION = -1 [Default: -1]

compression.strategy
Set the compression strategy this node will use when compressing synchronization payloads. Valid values include: FILTERED = 1, HUFFMAN_ONLY = 2, DEFAULT_STRATEGY = 0 [Default: 0]

stream.to.file.enabled
Save data to the file system before transporting it to the client or loading it to the database if the number of bytes is past a certain threshold. This allows for better compression and better use of database and network resources. Statistics in the batch tables will be more accurate if this is set to true because each timed operation is independent of the others. [Default: true]

stream.to.file.threshold.bytes
If stream.to.file.enabled is true, then the threshold number of bytes at which a file will be written is controlled by this property. Note that for a synchronization the entire payload of the synchronization will be buffered in memory up to this number (at which point it will be written and continue to stream to disk) [Default: 32767]

job.random.max.start.time.ms
When starting jobs, symmetric attempts to randomize the start time to spread out load. This is the maximum wait period before starting a job. [Default: 10000]

purge.retention.minutes
This is the retention for how long synchronization data will be kept in the SymmetricDS synchronization tables. Note that data will be purged only if the purge job is enabled. [Default: 7200]

statistic.retention.minutes
This is the retention for how long statistic data will be kept in the SymmetricDS staistic table. Note that data will be purged only if the purge job is enabled. [Default: 7200]

job.route.period.time.ms
This is how often the route job will be run. [Default: 10000]

job.push.period.time.ms
This is how often the push job will be run. [Default: 60000]

job.pull.period.time.ms
This is how often the pull job will be run. [Default: 60000]

job.synctriggers.aftermidnight.minutes
If scheduled, the sync triggers job will run nightly. This is how long after midnight that job will run. [Default: 15]

schema.version
This is hook to give the user a mechanism to indicate the schema version that is being synchronized. This property is only valid if you use the default IRuntimeConfiguration implementation.
[Default: ?]

registration.url
The URL where this node can connect for registration to receive its configuration. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default:]

sync.url
The URL where this node can be contacting for synchronization.
[Default: http://localhost:8080/sync]

group.id
The node group id for this node. [Default: default]

external.id
The secondary identifier for this node that has meaning to the system where it is deployed. While the node id is a generated sequence number, the external ID could have meaning in the user's domain, such as a retail store number. [Default:]

transport.type
Specify the transport type. Supported values currently include: http, internal. [Default: http]

hsqldb.initialize.db
If using the HsqlDbDialect, this property indicates whether Symmetric should setup the embedded database properties or if an external application will be doing so. [Default: true]

Appendix C. Database Notes

Each database management system has its own characteristics that results in feature coverage in SymmetricDS. The following table shows which features are available by database.

Table C.1. Support by Database

Database	Versions supported	Transaction Identifier	Fallback Update	Conditional Sync	Update Loop Prevention	BLOB Sync
Oracle	8.1.7 and above	Y	Y	Y	Y	Y
MySQL	5.0.2 and above	Y	Y	Y	Y	Y
PostgreSQL	8.2.5 and above	Y (8.3 and above only)	Y	Y	Y	Y
SQL Server	2005	Y	Y	Y	Y	Y
HSQLDB	1.8	Y	Y	Y	Y	Y
HSQLDB	2.0	N	Y	Y	Y	Y
H2	1.x	Y	Y	Y	Y	Y
Apache Derby	10.3.2.1	Y	Y	Y	Y	Y
IBM DB2	9.5	N	Y	Y	Y	Y
Firebird	2.0	Y	Y	Y	Y	Y
Informix	11	N	Y	Y	Y	N
Interbase	9.0	N	Y	Y	Y	Y

C.1. Oracle

On Oracle Real Application Clusters (RAC), sequences should be ordered so data is processed in the correct order. To offset the performance cost of ordering, the sequences should also be cached.

```
alter sequence SEQ_SYM_DATA_DATA_ID cache 1000 order;
alter sequence SEQ_SYM_OUTGOIN_BATCH_BATCH_ID cache 1000 order;
alter sequence SEQ_SYM_TRIGGER_RIGGER_HIST_ID cache 1000 order;
alter sequence SEQ_SYM_TRIGGER_TRIGGER_ID cache 1000 order;
```

While BLOBs are supported on Oracle, the LONG data type is not. LONG columns cannot be accessed from triggers.

Note that while Oracle supports multiple triggers of the same type to be defined, the order in which the triggers occur appears to be arbitrary.

The SymmetricDS user generally needs privileges for connecting and creating tables (including indexes), triggers, sequences, and procedures (including packages and functions). The following is an example of the needed grant statements:

```
GRANT CONNECT TO SYMMETRIC;
GRANT RESOURCE TO SYMMETRIC;
GRANT CREATE ANY TRIGGER TO SYMMETRIC;
GRANT EXECUTE ON UTL_RAW TO SYMMETRIC;
```

Partitioning the DATA table by channel can help insert, routing and extraction performance on concurrent, high throughput systems. TRIGGERs should be organized to put data that is expected to be inserted concurrently on separate CHANNELs. The following is an example of partitioning. Note that both the table and the index should be partitioned. The default value allows for more channels to be added without having to modify the partitions.

```
CREATE TABLE SYM_DATA
(
    data_id INTEGER NOT NULL ,
    table_name VARCHAR2(50) NOT NULL,
    event_type CHAR(1) NOT NULL,
    row_data CLOB,
    pk_data CLOB,
    old_data CLOB,
    trigger_hist_id INTEGER NOT NULL,
    channel_id VARCHAR2(20),
    transaction_id VARCHAR2(1000),
    source_node_id VARCHAR2(50),
    external_data VARCHAR2(50),
    create_time TIMESTAMP
) PARTITION BY LIST (channel_id) (
PARTITION P_CONFIG VALUES ('config'),
PARTITION P_CHANNEL_ONE VALUES ('channel_one'),
PARTITION P_CHANNEL_TWO VALUES ('channel_two'),
...
PARTITION P_CHANNEL_N VALUES ('channel_n'),
PARTITION P_DEFAULT VALUES (DEFAULT));
```

```
CREATE UNIQUE INDEX IDX_D_CHANNEL_ID ON SYM_DATA (DATA_ID, CHANNEL_ID)  LOCAL
(
 PARTITION I_CONFIG,
 PARTITION I_CHANNEL_ONE,
 PARTITION I_CHANNEL_TWO,
 ...
 PARTITION I_CHANNEL_N,
 PARTITION I_DEFAULT
);
```

C.2. MySQL

MySQL supports several storage engines for different table types. SymmetricDS requires a storage engine that handles transaction-safe tables. The recommended storage engine is InnoDB, which is included by default in MySQL 5.0 distributions. Either select the InnoDB engine during installation or modify your server configuration. To make InnoDB the default storage engine, modify your MySQL server configuration file (`my.ini` on Windows, `my.cnf` on Unix):

```
default-storage_engine = innodb
```

Alternatively, you can convert tables to the InnoDB storage engine with the following command:

```
alter table t engine = innodb;
```

On MySQL 5.0, the SymmetricDS user needs the SUPER privilege in order to create triggers.

```
grant super on *.* to symmetric;
```

On MySQL 5.1, the SymmetricDS user needs the TRIGGER and CREATE ROUTINE privileges in order to create triggers and functions.

```
grant trigger on *.* to symmetric;
```

```
grant create routine on *.* to symmetric;
```

MySQL allows '0000-00-00 00:00:00' to be entered as a value for datetime and timestamp columns. JDBC can not deal with a date value with a year of 0. In order to work around this SymmetricDS can be configured to treat date and time columns as varchar columns for data capture and data load. To enable this feature set the `db.treat.date.time.as.varchar.enabled` property to `true`.

C.3. PostgreSQL

Starting with PostgreSQL 8.3, SymmetricDS supports the transaction identifier. Binary Large Object (BLOB) replication is supported for both byte array (BYTEA) and object ID (OID) data types.

In order to function properly, SymmetricDS needs to use session variables. On PostgreSQL, session variables are enabled using a custom variable class. Add the following line to the `postgresql.conf` file of PostgreSQL server:

```

custom_variable_classes = 'symmetric'
```

This setting is required, and SymmetricDS will log an error and exit if it is not present.

Before database triggers can be created by in PostgreSQL, the plpgsql language handler must be installed on the database. The following statements should be run by the administrator on the database:

```

```

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;

CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
    '$libdir/plpgsql' LANGUAGE C;

CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler
    VALIDATOR plpgsql_validator;
```

C.4. MS SQL Server

SQL Server was tested using the jTDS JDBC driver.

C.5. HSQLDB

HSQLDB was implemented with the intention that the database be run embedded in the same JVM process as SymmetricDS. Instead of dynamically generating static SQL-based triggers like the other databases, HSQLDB triggers are Java classes that re-use existing SymmetricDS services to read the configuration and insert data events accordingly.

The transaction identifier support is based on SQL events that happen in a 'window' of time. The trigger(s) track when the last trigger fired. If a trigger fired within X milliseconds of the previous firing, then the current event gets the same transaction identifier as the last. If the time window has passed, then a new transaction identifier is generated.

C.6. H2

The H2 database allows only Java-based triggers. Therefore the H2 dialect requires that the SymmetricDS jar file be in the database's classpath.

C.7. Apache Derby

The Derby database can be run as an embedded database that is accessed by an application or a standalone server that can be accessed from the network. This dialect implementation creates database triggers that make method calls into Java classes. This means that the supporting JAR files need to be in the classpath when running Derby as a standalone database, which includes symmetric-ds.jar and commons-lang.jar.

C.8. IBM DB2

The DB2 Dialect uses global variables to enable and disable node and trigger synchronization. These

variables are created automatically during the first startup. The DB2 JDBC driver should be placed in the "lib" folder.

Currently, the DB2 Dialect for SymmetricDS does not provide support for transactional synchronization. Large objects (LOB) are supported, but are limited to 16,336 bytes in size. The current features in the DB2 Dialect have been tested using DB2 9.5 on Linux and Windows operating systems.

There is currently a bug with the retrieval of auto increment columns with the DB2 9.5 JDBC drivers that causes some of the SymmetricDS configuration tables to be rebuilt when auto.config.database=true. The DB2 9.7 JDBC drivers seem to have fixed the issue. They may be used with the 9.5 database.

A system temporary tablespace with too small of a page size may cause the following trigger build errors:

```
SQL1424N Too many references to transition variables and transition table
columns or the row length for these references is too long. Reason
code="2". LINE NUMBER=1. SQLSTATE=54040
```

Simply create a system temporary tablespace that has a bigger page size. A page size of 8k will probably suffice.

C.9. Firebird

The Firebird Dialect requires the installation of a User Defined Function (UDF) library in order to provide functionality needed by the database triggers. SymmetricDS includes the required UDF library, called SYM_UDF, in both source form (as a C program) and as pre-compiled libraries for both Windows and Linux. The SYM_UDF library is copied into the UDF folder within the Firebird installation directory.

For Linux users:

cp databases/firebird/sym_udf.so /opt/firebird/UDF

For Windows users:

copy databases\firebird\sym_udf.dll C:\Program Files\Firebird\Firebird_2_0\UDF

The following limitations currently exist for this dialect:

- The outgoing batch does not honor the channel size, and all outstanding data events are included in a batch.
- Syncing of Binary Large Object (BLOB) is limited to 16K bytes per column.
- Syncing of character data is limited to 32K bytes per column.

C.10. Informix

The Informix Dialect was tested against Informix Dynamic Server 11.50, but older versions may also work. You need to download the Informix JDBC Driver (from the IBM Download Site) and put the `ifxjdbc.jar` and `ifxlang.jar` files in the SymmetricDS `lib` folder.

Make sure your database has logging enabled, which enables transaction support. Enable logging when creating the database, like this:

```
CREATE DATABASE MYDB WITH LOG;
```

Or enable logging on an existing database, like this:

```
ondblog mydb unbuf log
ontape -s -L 0
```

The following features are not yet implemented:

- Syncing of Binary and Character Large Objects (LOB) is disabled.
- There is no transaction ID recorded on data captured, so it is possible for data to be committed within different transactions on the target database. If transaction synchronization is required, either specify a custom transaction ID or configure the synchronization so data is always sent in a single batch. A custom transaction ID can be specified with the tx_id_expression on TRIGGER. The batch size is controlled with the max_batch_size on CHANNEL. The pull and push jobs have runtime properties to control their interval.

C.11. Interbase

The Interbase Dialect requires the installation of a User Defined Function (UDF) library in order to provide functionality needed by the database triggers. SymmetricDS includes the required UDF library, called SYM_UDF, in both source form (as a C program) and as pre-compiled libraries for both Windows and Linux. The SYM_UDF library is copied into the UDF folder within the Interbase installation directory.

For Linux users:

cp databases/interbase/sym_udf.so /opt/interbase/UDF

For Windows users:

copy databases\interbase\sym_udf.dll C:\CodeGear\InterBase\UDF

The Interbase dialect currently has the following limitations:

- Data capture is limited to 4 KB per row, including large objects (LOB).
- There is no transaction ID recorded on data captured. Either specify a tx_id_expression on the TRIGGER table, or set a max_batch_size on the CHANNEL table that will accommodate your

transactional data.

Appendix D. Data Format

The SymmetricDS Data Format is used to stream data from one node to another. The data format reader and writer are pluggable with an initial implementation using a format based on Comma Separated Values (CSV). Each line in the stream is a record with fields separated by commas. String fields are surrounded with double quotes. Double quotes and backslashes used in a string field are escaped with a backslash. Binary values are represented as a string with hex values in "\0xab" format. The absence of any value in the field indicates a null value. Extra spacing is ignored and lines starting with a hash are ignored.

The first field of each line gives the directive for the line. The following directives are used:

nodeid, {node_id}
Identifies which node the data is coming from. Occurs once in CSV file.

binary, {BASE64|NONE|HEX}
Identifies the type of decoding the loader needs to use to decode binary data in the pay load. This varies depending on what database is the source of the data.

channel, {channel_id}
Identifies which channel a batch belongs to. The SymmetricDS data loader expects the channel to be specified before the batch.

batch, {batch_id}
Uniquely identifies a batch. Used to track whether a batch has been loaded before. A batch of -9999 is considered a virtual batch and will be loaded, but will not be recorded in incoming_batch.

schema, {schema name}
The name of the schema that is being targeted.

catalog, {catalog name}
The name of the catalog that is being targeted.

table, {table name}
The name of the table that is being targeted.

keys, {column name...}
Lists the column names that are used as the primary key for the table. Only needs to occur after the first occurrence of the table.

columns, {column name...}
Lists all the column names (including key columns) of the table. Only needs to occur after the first occurrence of the table.

insert, {column value...}
Insert into the table with the values that correspond with the columns.

update, {new column value...},{old key value...}
Update the table using the old key values to set the new column values.

old, {old column value...}
Represent all the old values of the data. This data can be used for conflict resolution.

delete, {old key value...}
Delete from the table using the old key values.

sql, {sql statement}
Optional notation that instructs the data loader to run the accompanying SQL statement.

bsh, {bsh script}
Optional notation that instructs the data loader to run the accompanying BeanShell snippet.

create, {xml}
Optional notation that instructs the data loader to run the accompanying DdlUtils XML table definition in order to create a database table.

commit, {batch_id}
An indicator that the batch has been transmitted and the data can be committed to the database.

Example D.1. Data Format Stream

```
nodeid, 1001
channel, pricing
binary, BASE64
batch, 100
schema,
catalog,
table, item_selling_price
keys, price_id
columns, price_id, price, cost
insert, 55, 0.65, 0.55
schema,
catalog,
table, item
keys, item_id
columns, item_id, price_id, name
insert, 110000055, 55, "Soft Drink"
delete, 110000001
schema,
catalog,
table, item_selling_price
update, 55, 0.75, 0.65, 55
commit, 100
```

Appendix E. Version Numbering

The software is released with a version number based on the Apache Portable Runtime Project version guidelines. In summary, the version is denoted as three integers in the format of MAJOR.MINOR.PATCH. Major versions are incompatible at the API level, and they can include any kind of change. Minor versions are compatible with older versions at the API and binary level, and they can introduce new functions or remove old ones. Patch versions are perfectly compatible, and they are released to fix defects.