SymmetricDS User Guide

v3.6

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 2014-08-28 at 15:13:03.

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. System Requirements

SymmetricDS is written in Java and requires a Java SE Runtime Environment (JRE) or Java SE Development Kit (JDK) version 6.0 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
- MariaDB version 5.1 and above
- Oracle version 10g and above
- PostgreSQL version 8.2.5 and above
- Sql Server 2005 and above
- Sql Server Azure
- HSQLDB 2.x
- H2 1.x
- Apache Derby 10.3.2.1 and above
- IBM DB2 9.5
- Firebird 2.0 and above
- Interbase 2009 and above
- Greenplum 8.2.15 and above
- SQLite 3 and above
- Sybase Adaptive Server Enterprise 12.5 and above
- Sybase SQL Anywhere 9 and above

See Appendix C, *Database Notes* (p. 139), for compatibility notes and other details for your specific database.

1.2. Concepts

1.2.1. Nodes

SymmetricDS is a Java-based application that hosts a synchronization engine which acts as an agent for data synchronization between a single database instance and other synchronization engines in a network.

The SymmetricDS engine is referred to as a *node*. SymmetricDS is designed to be able to scale out to many thousands of nodes. The database connection is configured by providing a database connection string, database user, and database password in a properties file. SymmetricDS can synchronize any table that is accessible by the database connection, given that the database user has been assigned the appropriate database permissions.

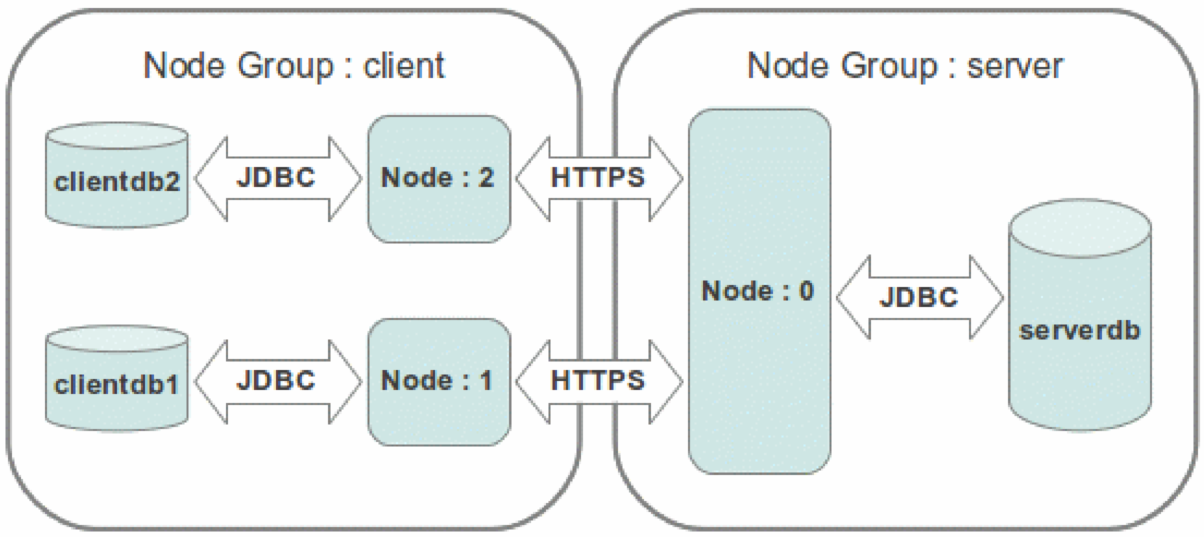


Figure 1.1. Simple Configuration

A SymmetricDS node is assigned an external id and a node group id. The external id is a meaningful, user-assigned identifier that is used by SymmetricDS to understand which data is destined for a given node. The node group id is used to identify groupings or tiers of nodes. It defines where the node fits into the overall node network. For example, one node group might be named “corporate” and represent an enterprise or corporate database. Another node group might be named “local_office” and represent databases located in different offices across a country. The external id for a “local_office” could be an office number or some other identifying alphanumeric string. A node is uniquely identified in a network by a node id that is automatically generated from the external id. If local office number 1 had two office databases and two SymmetricDS nodes, they would probably have an external id of “1” and node ids of “1-1” and “1-2.”

SymmetricDS can be deployed in a number of ways. The most common option is to deploy it as a stand alone process running as a service on your chosen server platform. When deployed in this manner SymmetricDS can act as either a client, a multi-tenant server or both depending on where the SymmetricDS database fits into the overall network of databases. Although it can run on the same server as its database, it is not required to do so. SymmetricDS can also be deployed as a web application in an application server such as Apache Tomcat, JBoss Application Server, IBM WebSphere, or others.

SymmetricDS was designed to be a simple, approachable, non-threatening tool for technology personnel. It can be thought of and dealt with as a web application, only instead of a browser as the client, other SymmetricDS engines are the clients. It has all the characteristics of a web application and can be tuned using the same principles that would be used to tune user facing web applications.

1.2.2. Change Data Capture

Changes are captured at a SymmetricDS enabled database by database triggers that are installed automatically by SymmetricDS based on configuration settings that you specify. The database triggers record data changes in the DATA table. The triggers are designed to be as noninvasive and as lightweight as possible. After SymmetricDS triggers are installed, changes are captured for any Data Manipulation Language (DML) statements performed by external applications. Note that no additional libraries or changes are needed by the applications that use the database and SymmetricDS does not have to be online for data to be captured.

Database tables that need to be replicated are configured in a series of SymmetricDS configuration tables. The configuration for the entire network of nodes is typically managed at a central node in the network, known as the registration server node. The registration server node is almost always the same node as the root node in a tree topology. When configuring “leaf” nodes, one of the start-up parameters is the URL of the registration server node. If the “leaf” node has not yet registered, it contacts the registration server and requests to join the network. Upon acceptance, the node downloads its configuration. After a node is registered, SymmetricDS can also provide an initial load of data before synchronization starts.

SymmetricDS will install or update its database triggers at start-up time and on a regular basis when a scheduled sync triggers job runs (by default, each night at midnight). The sync triggers job detects changes to your database structure or trigger configuration when deciding whether a trigger needs to be rebuilt. Optionally, the sync triggers job can be turned off and the database triggers DDL script can be generated and run by a DBA.

After changed data is inserted by the database trigger into the DATA table, it is batched and assigned to a node by the router job. Routing data refers to choosing the nodes in the SymmetricDS network to which the data should be sent. By default, data is routed to other nodes based on the node group. Optionally, characteristics of the data or of the target nodes can also be used for routing. A batch of data is a group of data changes that are transported and loaded together at the target node in a single database transaction. Batches are recorded in the OUTGOING_BATCH . Batches are node specific. DATA and OUTGOING_BATCH are linked by DATA_EVENT . The delivery status of a batch is maintained in OUTGOING_BATCH . After the data has been delivered to a remote node the batch status is changed to ‘OK.’

1.2.3. Change Data Delivery

Data is delivered to remote nodes over HTTP or HTTPS. It can be delivered in one of two ways depending on the type of transport link that is configured between node groups. A node group can be configured to push changes to other nodes in a group or pull changes from other nodes in a group. Pushing is initiated from the push job at the source node. If there are batches that are waiting to be transported, the pushing node will reserve a connection to each target node using an HTTP HEAD request. If the reservation request is accepted, then the source node will fully extract the data for the batch. Data is extracted to a memory buffer in CSV format until a configurable threshold is reached. If the threshold is reached, the data is flushed to a file and the extraction of data continues to that file. After the batch has been extracted, it is transported using an HTTP PUT to the target node. The next batch is then extracted and sent. This is repeated until the maximum number of batches have been sent for each channel or there are no more batches available to send. After all the batches have been sent for one push, the target returns a list of the batch statuses.

Pull requests are initiated by the pull job from at the target node. A pull request uses an HTTP GET. The same extraction process that happens for a "push" also happens during a "pull."

After data has been extracted and transported, the data is loaded at the target node. Similar to the extract process, while data is being received the data loader will cache the CSV in a memory buffer until a threshold is reached. If the threshold is reached the data is flushed to a file and the receiving of data continues. After all of the data in a batch is available locally, a database connection is retrieved from the connection pool and the events that had occurred at the source database are played back against the target database.

1.2.4. Data Channels

Data is always delivered to a remote node in the order it was recorded for a specific channel. A channel is a user defined grouping of tables that are dependent on each other. Data that is captured for tables belonging to a channel is always synchronized together. Each trigger must be assigned a channel id as part of the trigger definition process. The channel id is recorded on SYM_DATA and SYM_OUTGOING_BATCH. If a batch fails to load, then no more data is sent for that channel until the failure has been addressed. Data on other channels will continue to be synchronized, however.

If a remote node is offline, the data remains recorded at the source database until the node comes back online. Optionally, a timeout can be set where a node is removed from the network. Change data is purged from the data capture tables by SymmetricDS after it has been sent and a configurable purge retention period has been reached. Unsent change data for a disabled node is also purged.

The default behavior of SymmetricDS in the case of data integrity errors is to attempt to repair the data. If an insert statement is run and there is already a row that exists, SymmetricDS will fall back and try to update the existing row. Likewise, if an update that was successful on a source node is run and no rows are found to update on the destination, then SymmetricDS will fall back to an insert on the destination. If a delete is run and no rows were deleted, the condition is simply logged. This behavior can be modified by tweaking the settings for conflict detection and resolution.

SymmetricDS was designed to use standard web technologies so it can be scaled to many clients across different types of databases. It can synchronize data to and from as many client nodes as the deployed database and web infrastructure will support. When a two-tier database and web infrastructure is maxed out, a SymmetricDS network can be designed to use N-tiers to allow for even greater scalability. At this point we have covered what SymmetricDS is and how it does its job of replicating data to many databases using standard, well understood technologies.

1.3. 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. 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.

1.3.2. 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 3.3, Channels (p. 12) .

1.3.3. Change Notification

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.

1.3.4. 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.

1.3.5. 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` .
- In addition to synchronization, SymmetricDS is also capable of performing fairly complex transformations (see Section 3.8) 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.
- As data changes are loaded in the target database, data can be filtered, either by a simple bean shell load filter (see Section 3.9 data-load-filter) or by a class implementing IDatabaseWriterFilter. You can change the data in a column, route it somewhere else, trigger initial loads, or many other possibilities. 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.

1.3.6. 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.

1.3.7. 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.

1.3.8. File Synchronization

Quite a few users of SymmetricDS have found that they have a need to not only synchronize database tables to remote locations, but they also have a set of files that should be synchronized. As of version 3.5 SymmetricDS now support file synchronization.

Please see Section 3.5, File Triggers / File Synchronization (p. 16) for more information.

1.4. Why Database Triggers?

There are several industry recognized techniques to capture changing data for replication, synchronization and integration in a relational database.

- *Lazy data capture* queries changed data from a source system using some SQL condition (like a time stamp column).
- *Trigger-based data capture* installs database triggers to capture changes.
- *Log-based data capture* reads data changes from proprietary database recovery logs.

All three of these techniques have advantages and disadvantages, and all three are on the road map for SymmetricDS. At present time, SymmetricDS supports trigger-based data capture and partial lazy data capture. These two techniques were implemented first for a variety of reasons, not the least of which is that the majority of use cases that SymmetricDS targets can be solved using trigger-based and conditional replication in a way that allows for more database platforms to be supported using industry standard technologies. This fact allowed SymmetricDS developers' valuable time and energy to be invested in designing a product that is easy to install, configure and manage versus spending time reverse engineering proprietary and not well documented database log files.

Trigger-based data capture does introduce a measurable amount of overhead on database operations. The amount of overhead can vary greatly depending on the processing power and configuration of the database platform, and the usage of the database by applications. With nonstop advances in hardware and database technology, trigger-based data capture has become feasible for use cases that involve high data throughput or require scaling out.

Trigger-based data capture is easier to implement and support than log-based solutions. It uses well known database concepts and is very accessible to software and database developers and database administrators. It can usually be installed, configured, and managed by application development teams or database administrators and does not require deployment on the database server itself.

Chapter 2. Setup

2.1. Engine Files

Each node requires properties that allow it to connect to a database and register with a parent node. Properties are configured in a file named `xxxxx.properties` that is placed in the engines directory of the SymmetricDS install. The file is usually named according to the engine.name, but it is not a requirement.

To give a node its identity, the following properties are required. Any other properties found in `conf/symmetric.properties` can be overridden for a specific engine in an engine's properties file. If the properties are changed in `conf/symmetric.properties` they will take effect across all engines deployed to the server. Note that you can use the variable `$(hostName)` to represent the host name of the machine when defining these properties (for example, external.id=$(hostName)).

engine.name
This is an arbitrary name that is used to access a specific engine using an HTTP URL. Each node configured in the engines directory must have a unique engine name. The engine name is also used for the domain name of registered JMX beans.

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. The sync url is of the format: `http://{hostname}:{port}/{webcontext}/sync/{engine.name}` .

The {webcontext} is blank for a standalone deployment. It will typically be the name of the war file for an application server deployment.

The {engine.name} can be left blank if there is only one engine deployed in a SymmetricDS server.

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. This is typically equal to the

value of the sync.url of the registration server.

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` .

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.

See Section B.1, Startup Parameters (p. 121),)for additional parameters that can be specified in the engine properties file.

Chapter 3. Configuration

3.1. Groups

Groups 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');
```

3.2. Group Links

Group links are defined in the NODE_GROUP_LINK table. Links define how a node that belongs to a group will communicate with nodes in other groups. The following are the communication mechanisms that can be configured.

Push (P)
Indicates that the source node will initiate communication over an HTTP PUT.

Wait for Pull (W)
Indicates that the source node will *wait* for a target node to connect via an HTTP GET to pull data.

Route-only (R)
Route-only indicates that the data isn't going to be transported via SymmetricDS. This action type might be useful when using an XML publishing router or an audit table changes router.

The link also defines if configuration data will be synchronized on the link. For example, you might not want remote nodes to be able to change configuration and effect other nodes in the network. In this case you would set `sync_config_enabled` to 0 on the appropriate link.

A link can be configured to use the same node group as the source and the target. This configuration allows a node group to sync with every other node in its group.

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');
```

3.3. Channels

By categorizing data into channels and assigning them to TRIGGER s, 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, rocessing_order, max_batch_size, max_batch_to_s
                  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, descri
          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.

If a channel contains *only* tables that will be synchronized in one direction and and data is routed to all the nodes in the target node groups, then batching on the channel can be optimized to share batches across nodes. This is an important feature when data needs to be routed to thousands of nodes. When this mode is detected, you will see batches created in OUTGOING_BATCH with the `common_flag` set to 1.

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
Specifies 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).

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.

3.4. Table Triggers

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.

The `source_table_name` may contain the asterisk ('*') wildcard character so that one TRIGGER table entry can define synchronization for many tables. System tables and any tables that start with the SymmetricDS table prefix will be excluded. A list of wildcard tokens can also be supplied. If there are multiple tokens, they should be delimited with a comma. A wildcard token can also start with a bang ('!') to indicate an exclusive match. Tokens are always evalulated from left to right. When a table match is made, the table is either added to or removed from the list of tables. If another trigger already exists for a table, then that table is not included in the wildcard match (the explictly defined trigger entry take precendence).

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, DB2, SQL Server 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);
```

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.

3.4.1. Linking Triggers

The TRIGGER_ROUTER table is used to define which specific combinations of triggers and routers are needed for your configuration. The relationship between triggers and routers is many-to-many, so this table serves as the join table to define which combinations are valid, as well as to define settings available at the trigger-router level of granularity.

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

3.4.1.1. Enable / disable trigger router

Each individual trigger-router combination can be disabled or enabled if needed. By default, a trigger router is enabled, but if you have a reason you wish to define a trigger router combination prior to it being active, you can set the `enabled` flag to 0. This will cause the trigger-router mapping to be sent to all nodes, but the trigger-router mapping will not be considered active or enabled for the purposes of capturing data changes or routing.

3.4.1.2. Enabling "Ping Back"

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.

3.4.2. Large Objects

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.

3.4.3. External Select

Occasionally, you may find that you need to capture and save away a piece of data present in another table when a trigger is firing. This data is typically needed for the purposes of determining where to 'route' the data to once routing takes place. Each trigger definition contains an optional `external_select` field which can be used to specify the data to be captured. Once captured, this data is available during routing in DATA 's `external_data` field. For these cases, place a SQL select statement which returns the data item you need for routing in `external_select` . An example of the use of external select can be found in Section 3.6.7, Utilizing External Select when Routing (p. 27) .

3.4.4. 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 transactions 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
                            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);
```

3.4.5. 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 changes 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 4.3.5, Sync Triggers Job (p. 50).

3.5. File Triggers / File Synchronization

3.5.1. Overview

SymmetricDS not only supports the synchronization of database tables, but it also supports the synchronization of files and folders from one node to another.

File synchronization features include:

- Monitoring one or more file system directory locations for file and folder changes
- Support synchronizing a different target directory than the source directory
- Use of wild card expressions to “include” or “exclude” files
- Choice of whether to recurse into subfolders of monitored directories
- Use of existing SymmetricDS routers to subset target nodes based on file and directory metadata
- Ability to specify if files will be synchronized on creation, or deletion, and/or modification
- Ability to specify the frequency with which file systems are monitored for changes
- Ability to extend file synchronization through scripts that run before or after a file is copied to its source location
- Support for bidirectional file synchronization

Like database synchronization, file synchronization is configured in a series of database tables. The configuration was designed to be similar to database synchronization in order to maintain consistency and to give database synchronization users a sense of familiarity.

For database synchronization, SymmetricDS uses TRIGGER to configure which tables will capture data for synchronization and ROUTER to designate which nodes will be the source of data changes and which nodes will receive the data changes. TRIGGER_ROUTER links triggers to routers.

Likewise, for file synchronization, SymmetricDS uses FILE_TRIGGER to designate which base directories will be monitored. Each entry in FILE_TRIGGER designates one base directory to monitor for changes on the source system. The columns on FILE_TRIGGER provide additional settings for choosing specific files in the base directory that will be monitored, and whether to recurse into subdirectories, etc. File triggers are linked to routers by FILE_TRIGGER_ROUTER. The file trigger router not only links the source and the target node groups, but it also optionally provides the ability to override the base directory name at the target. FILE_TRIGGER_ROUTER also provides a flag that indicates if the target node should be seeded with the files from the source node during SymmetricDS's initial load process.

3.5.2. Operation

Not only is file synchronization configured similar to database synchronization, but it also operates in a very similar way. The file system is monitored for changes via a background job that tracks the file system changes (this parallels the use of triggers to monitor for changes when synchronizing database changes). When a change is detected it is written to the FILE_SNAPSHOT table. The file snapshot table represents the most recent known state of the monitored files. The file snapshot table has a SymmetricDS database trigger automatically installed on it so that when it is updated the changes are captured by SymmetricDS on an internal channel named `filesync`.

The changes to FILE_SNAPSHOT are then routed and batched by a file-synchronization-specific router that delegates to the configured router based on the FILE_TRIGGER_ROUTER configuration. The file sync router can make routing decisions based on the column data of the snapshot table, columns which contain attributes of the file like the name, path, size, and last modified time. Both old and new file snapshot data are also available. The router can, for example, parse the path or name of the file and use it as the node id to route to.

Batches of file snapshot changes are stored on the `filesync` channel in OUTGOING_BATCH. The existing SymmetricDS pull and push jobs ignore the `filesync` channel. Instead, they are processed by file-synchronization-specific push and pull jobs.

When transferring data, the file sync push and pull jobs build a zip file dynamically based on the batched snapshot data. The zip file contains a directory per batch. The directory name is the `batch_id`. A `sync.bsh` Bean Shell script is generated and placed in the root of each batch directory. The Bean Shell script contains the commands to copy or delete files at their file destination from an extracted zip in the staging directory on the target node. The zip file is downloaded in the case of a pull, or, in the case of a push, is uploaded as an HTTP multi-part attachment. Outgoing zip files are written and transferred from the outgoing staging directory. Incoming zip files are staged in the `filesync_incoming` staging directory by source node id. The `filesync_incoming/{node_id}` staging directory is cleared out before each subsequent delivery of files.

The acknowledgement of a batch happens the same way it is acknowledged in database synchronization. The client responds with an acknowledgement as part of the response during a file push or pull.

3.5.3. File Sync Bean Shell Scripts

There are two types of Bean Shell scripts that can be leveraged to customize file synchronization behavior: `before_copy_script` and `after_copy_script`.

Each of these scripts have access to local variables that can be read or set to affect the behavior of copying files.

targetBaseDir
The preset base directory as configured in FILE_TRIGGER or overwritten in FILE_TRIGGER_ROUTER. This variable can be set by the `before_copy_script` to set a different target directory.

targetFileName
The name of the file that is being synchronized. This variable can be overwritten by the `before_copy_script` to rename a file at the target.

targetRelativeDir
The name of a directory relative to the target base directory to which the target file will be copied. The default value of this variable is the relative directory of the source. For example, if the source base directory is `/src` and the target base directory is `/tgt` and the file `/src/subfolder/1.txt` is changed, then the default targetRelativeDir will be `subfolder`. This variable can be overwritten by the `before_copy_script` to change the relative directory at the target. In the above example, if the variable is set to blank using the following script, then the target file will be copied to `/tgt/1.txt`.

```
targetRelativeDir = "";
```

processFile
This is a variable that is set to true by default. A custom `before_copy_script` may process the file itself and set this variable to false to indicate that the file should NOT be copied to its target location.

sourceFileName
This is the name of the file.

sourceFilePath
This is the path where the file can be found relative to the batch directory.

batchDir
This is the staging directory where the batch has been extracted. The batchDir + sourceFilePath + sourceFileName can be used to locate the extracted file.

engine
This is the bound instance of the ISymmetricEngine that is processing a file. It gives access to all of the APIs available in SymmetricDS.

sourceNodeId
This is a bound variable that represents the nodeId that is the source of the file.

log
This is the bound instance of an `org.slf4j.Logger` that can be used to log to the SymmetricDS log file.

3.5.4. File Sync Examples

3.5.4.1. Sync Text Files From Server To Client

The following example is for a configuration with client and server node groups. Creation, modification, and deletion of files with the extension of `txt` will be captured recursively in the `/filesync/server/all` directory. A before copy script will set the targetBaseDir to `/filesync/clients/{externalId}`.

```
INSERT INTO sym_file_trigger
  (trigger_id,base_dir,recurse,includes_files,excludes_files,sync_on_create,
   sync_on_modified,sync_on_delete,sync_on_ctl_file,delete_after_sync,before_copy_script,aft
   create_time,last_update_by,last_update_time)
VALUES ('sync_directory','/filesync/server/all',1,'*.txt',null,1,1,1,0,0,
  'targetBaseDir = "/filesync/clients/" +
  engine.getParameterService().getExternalId();',null,current_timestamp,'example',
  current_timestamp);

INSERT INTO sym_file_trigger_router
 (trigger_id,router_id,enabled,initial_load_enabled,target_base_dir,
  conflict_strategy,create_time,last_update_by,last_update_time)
VALUES
  ('sync_directory','server_2_client',1,1,'','SOURCE_WINS',current_timestamp,
  'example',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
  ('server_2_client',null,null,null,'server','client','default',null,1,1,1,
   current_timestamp,'example',current_timestamp);
```

3.5.4.2. Route changes to a specific node based on a directory name

The following example is also for a configuration with client and server node groups. This example monitors the `/filesync/server/nodes` directory. It expects the directory to contain subdirectories that are named by the node_ids in the client group. Any files put directly into a folder with the name of the node will be routed to that node.

Note that the router is a Section 3.6.2, Column Match Router (p. 21) that is matching the client node_id with the value of the RELATIVE_DIR column in FILE_SNAPSHOT. Because the router is looking for an exact match any files in subdirectories would result in a path of node_id/subdir which would not match.

```


INSERT INTO sym_file_trigger
  (trigger_id,base_dir,recurse,includes_files,excludes_files,sync_on_create,
  sync_on_modified,sync_on_delete,sync_on_ctl_file,delete_after_sync,before_copy_script,afte
  last_update_by,last_update_time)
VALUES
  ('node_specific','/filesync/server/nodes',1,null,null,1,1,1,0,0,'',null,
  current_timestamp,'example',current_timestamp);

INSERT INTO sym_file_trigger_router
  (trigger_id,router_id,enabled,initial_load_enabled,target_base_dir,
  conflict_strategy,create_time,last_update_by,last_update_time)
VALUES
  ('node_specific','router_files_to_node',1,1,'/filesync/clients','SOURCE_WINS',
  current_timestamp,'example',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
  ('router_files_to_node',null,null,null,'server','client','column',
  'RELATIVE_DIR = :NODE_ID ',1,1,1,current_timestamp,'example', current_timestamp);
```

3.6. Routers

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 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.
- Audit Table Router - a router that inserts into an automatically created audit table. It records captured changes to tables that it is linked to.

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.

3.6.1. 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);
```

3.6.2. 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 'READY TO SEND.' 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=READY TO SEND',
               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

Captured EXTERNAL_DATA is also available for routing as a virtual column.

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);
```

3.6.3. 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);
```

3.6.4. Subselect Router

Sometimes routing decisions need to be made based on data that is not in the current row being synchronized. A 'subselect' router can be used in these cases. 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 data being relied on to determine the node id has been deleted before routing takes place, then no results would be returned and routing would not happen.

The `router_expression` you specify 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 ...
```

As you can see, you have access to information about the node currently under consideration for routing through the 'c' alias, for example `c.external_id` . There are two node-related tokens you can use in your expression:

- :NODE_GROUP_ID
- :EXTERNAL_DATA

Column names representing data for the row in question are prefixed with a colon as well, for example: `:EMPLOYEE_ID` , or `:OLD_EMPLOYEE_ID` . Here, the OLD_ prefix indicates the value before the change in cases where the old data has been captured.

For an example, consider the case where an Order table and an 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 solve this in SymmetricDS. One is to configure a 'subselect' router_type on the ROUTER table, shown below (The other possible approach is to use an `external_select` to capture the data via a trigger for use in a column match router, demonstrated in Section 3.6.7, Utilizing External Select when Routing (p. 27)).

Our solution utilizing subselect compares the external id of the current node with the store id from the Order table where the order id matches the order id of the current row 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','corp', 'store', 'subselect', 'c.external_id in (select
               STORE_ID from order where order_id=:ORDER_ID)', current_timestamp,
               current_timestamp);
```

As a final note, please note in this example that the parent row in Order must still exist at the moment of routing for the child rows (OrderLineItem) to route, since the select statement is run when routing is occurring, not when the change data is first captured.

3.6.5. 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 `org.jumpmind.symmetric.model.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_'.

If you need access to any of the SymmetricDS services, then the instance of `org.jumpmind.symmetric.ISymmetricEngine` is accessible via the bound `engine` variable.

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);
```

The next example shows a script that iterates over each eligible node and checks to see if the trimmed value of the column named STATION equals the external_id.

```
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-trimmed-station','corp', 'store', 'bsh', 'for
               (org.jumpmind.symmetric.model.Node node : nodes) { if (STATION != null
               && node.getExternalId().equals(STATION.trim())) {
               targetNodes.add(node.getNodeId()); } }', current_timestamp,
               current_timestamp);
```

3.6.6. Audit Table Router

This router audits captured data by recording the change in an audit table that the router creates and keeps up to date (as long as `auto.config.database` is set to true.) The router creates a table named the same as the table for which data was captured with the suffix of _AUDIT. It will contain all of the same columns as the original table with the same data types only each column is nullable with no default values.

Three extra "AUDIT" columns are added to the table:

- AUDIT_ID - the primary key of the table.
- AUDIT_TIME - the time at which the change occurred.
- AUDIT_EVENT - the DML type that happened to the row.

The following is an example of an audit router

```
insert into SYM_ROUTER (router_id,
               source_node_group_id, target_node_group_id, router_type, create_time,
               last_update_time) values ('audit_at_corp','corp', 'local', 'audit',
               current_timestamp, current_timestamp);
```

The audit router captures data for a group link. For the audit router to work it must be associated with a node_group_link with an action of type 'R'. The 'R' stands for 'only routes to'. In the above example, we refer to a 'corp to local' group link. Here, local is a new node_group created for the audit router. No nodes belong to the 'local' node_group. If a trigger linked to an audit router fires on the corp node, a new audit table will be created at the corp node with the new data inserted.

3.6.7. Utilizing External Select when Routing

There may be times when you wish to route based on a piece of data that exists in a table other than the one being routed. The approach, first discussed in Section 3.6.4, Subselect Router (p. 24) , is to utilize an `external_select` to save away data in `external_data` , which can then be referenced during routing.

Reconsider subselect's Order / OrderLineItem example (found in Section 3.6.4, Subselect Router (p. 24)), where routing for the line item is accomplished by linking to the "header" Order row. As an alternate way of solving the problem, we will now use External Select combined with a column match router.

In this version of the solution, the STORE_ID is captured from the Order table in the EXTERNAL_DATA column when the trigger fires. The router is configured to route based on the captured EXTERNAL_DATA to all nodes whose external id matches the captured external data.

```
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);
```

The following variables can be used with the external select:

$(curTriggerValue)
Variable to be replaced with the NEW or OLD column alias provided by the trigger context, which is

platform specific. For insert and update triggers, the NEW alias is used; for delete triggers, the OLD alias is used. For example, "$(curTriggerValue).COLUMN" becomes ":new.COLUMN" for an insert trigger on Oracle.

$(curColumnPrefix)
Variable to be replaced with the NEW_ or OLD_ column prefix for platforms that don't support column aliases. This is currently only used by the H2 database. All other platforms will replace the variable with an empty string. For example "$(curColumnPrefix)COLUMN" becomes "NEW_COLUMN" on H2 and "COLUMN" on Oracle.

The advantage of this approach over the 'subselect' approach is that it guards against the (somewhat unlikely) possibility that the master Order table row might have been deleted before routing has taken place. This external select solution also is a bit more efficient than the 'subselect' approach, although the triggers produced do run the extra external_select SQL inline with application database updates.

3.7. Conflicts

3.7.1. Conflict Detection and Resolution

Conflict detection and resolution is new as of SymmetricDS 3.0. Conflict detection is the act of determining if an insert, update or delete is in "conflict" due to the target data row not being consistent with the data at the source prior to the insert/update/delete. Conflict resolution is the act of figuring out what to do when a conflict is detected.

Conflict detection and resolution strategies are configured in the CONFLICT table. They are configured at minimum for a specific NODE_GROUP_LINK . The configuration can also be specific to a CHANNEL and/or table.

Conflict detection is configured in the `detect_type` and `detect_expression` columns of CONFLICT . The value for `detect_expression` depends on the `detect_type` . Conflicts are detected while data is being loaded into a target system.

USE_PK_DATA
Indicates that only the primary key is used to detect a conflict. If a row exists with the same primary key, then no conflict is detected during an update or a delete. Updates and deletes rows are resolved using only the primary key columns. If a row already exists during an insert then a conflict has been detected.

USE_OLD_DATA
Indicates that all of the old data values are used to detect a conflict. Old data is the data values of the row on the source system prior to the change. If a row exists with the same old values on the target system as they were on the source system, then no conflict is detected during an update or a delete. If a row already exists during an insert then a conflict has been detected.

Note that some platforms do not support comparisons of binary columns. Conflicts in binary column values will not be detected on the following platforms: DB2, DERBY, ORACLE, and SQLSERVER.

USE_CHANGED_DATA
Indicates that the primary key plus any data that has changed on the source system will be used to detect a conflict. If a row exists with the same old values on the target system as they were on the source system for the columns that have changed on the source system, then no conflict is detected during an update or a delete. If a row already exists during an insert then a conflict has been detected.

Note that some platforms do not support comparisons of binary columns. Conflicts in binary column values will not be detected on the following platforms: DB2, DERBY, ORACLE, and SQLSERVER.

The detect_expression can be used to exclude certain column names from being used. In order to exclude column1 and column2, the expression would be: `excluded_column_names=column1,column2`

USE_TIMESTAMP
Indicates that the primary key plus a timestamp column (as configured in `detect_expression`) will indicate whether a conflict has occurred. If the target timestamp column is not equal to the old source timestamp column, then a conflict has been detected. If a row already exists during an insert then a conflict has been detected.

USE_VERSION
Indicates that the primary key plus a version column (as configured in `detect_expression`) will indicate whether a conflict has occurred. If the target version column is not equal to the old source version column, then a conflict has been detected. If a row already exists during an insert then a conflict has been detected.

Important

Be aware that conflict detection will *not* detect changes to binary columns in the case where `use_stream_lobs` is true in the trigger for the table. In addition, some databases do not allow comparisons of binary columns whether `use_stream_lobs` is true or not.

The choice of how to resolve a detected conflict is configured via the `resolve_type` column. Depending on the setting, two additional boolean settings may also be configured, namely `resolve_row_only` and `resolve_changes_only`, as discussed in the resolution settings below.

FALLBACK
Indicates that when a conflict is detected the system should automatically apply the changes anyways. If the source operation was an insert, then an update will be attempted. If the source operation was an update and the row does not exist, then an insert will be attempted. If the source operation was a delete and the row does not exist, then the delete will be ignored. The `resolve_changes_only` flag controls whether all columns will be updated or only columns that have changed will be updated during a fallback operation.

IGNORE
Indicates that when a conflict is detected the system should automatically ignore the incoming change. The `resolve_row_only` column controls whether the entire batch should be ignore or just the row in conflict.

MANUAL

Indicates that when a conflict is detected the batch will remain in error until manual intervention occurs. A row in error is inserted into the INCOMING_ERROR table. The conflict detection id that detected the conflict is recorded (i.e., the `conflict_id` value from CONFLICT), along with the old data, new data, and the "current data" (by current data, we mean the unexpected data at the target which doesn't match the old data as expected) in columns `old_data`, `new_data`, and `cur_data`. In order to resolve, the `resolve_data` column can be manually filled out which will be used on the next load attempt instead of the original source data. The `resolve_ignore` flag can also be used to indicate that the row should be ignored on the next load attempt.

NEWER_WINS
Indicates that when a conflict is detected by USE_TIMESTAMP or USE_VERSION that the either the source or the target will win based on the which side has the newer timestamp or higher version number. The `resolve_row_only` column controls whether the entire batch should be ignore or just the row in conflict.

For each configured conflict, you also have the ability to control if and how much "resolved" data is sent back to the node who's data change is in conflict. This "ping back" behavior is specified by the setting of the `ping_back` column and can be one of the following values:

OFF
No data is sent back to the originating node, even if the resolved data doesn't match the data the node sent.

SINGLE_ROW
The resolved data of the single row in the batch that caused the conflict is sent back to the originating node.

REMAINING_ROWS.
The resolved data of the single row in the batch in conflict, along with the entire remainder of the batch, is sent back to the originating node.

3.8. Transforms

New as of 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 synchronization 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.

Please note that the transformation 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.

3.8.1. 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.
- column_policy: Indicates whether unspecified columns are passed thru or if all columns must be explicitly defined. The options include:
  - SPECIFIED - Indicates that only the transform columns that are defined will be the ones that end up as part of the transformation.
  - IMPLIED - Indicates that if not specified, then columns from the source are passed through to the target. This is useful if you just want to map a table from one name to anther or from one schema to another. It is also useful if you want to transform a table, but also want to pass it through. You would define an implied transform from the source to the target and would not have to configure each column.
- 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.

3.8.2. 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.

- **Remove Column Transform ('remove')**: This transformation type removes the source column. This transform type is only valid for a table transformation type of 'IMPLIED' where all the columns from the source are automatically copied to the target.

- **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 dynamic 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, `system_timestamp` is the current system date and time, `source_node_id` is the node id of the source, `target_node_id` is the node id of the target, `null` is a null value, and `old_column_value` is the column's old value prior to the DML operation, `source_table_name` is the name of the source table as captured in the trigger hist table, `source_catalog_name` is the name of the source catalog as captured in the trigger hist table, `source_schema_name` is the name of the source schema as captured in the trigger hist table.

- **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 the change to the existing value in the target column. That is, target = target + multiplier (source_new - source_old), where multiplier is a constant found in the transform_expression (default is 1 if not specified). For example, if the source column changed from a 2 to a 4, the target column is currently 10, and the multiplier is 3, the effect of the transform will be to change the target column to a value of 16 (10+3*(4-2) => 16). Note that, in the case of deletes, the new column value is considered 0 for the purposes of the calculation.

- **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.

- **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.

- **Lookup Transform ('lookup')**: This transformation determines the target column value by using a query, contained in transform_expression to lookup the value in another table. The query must return a single row, and the first column of the query is used as the value. Your query references source column names by prefixing with a colon (e.g., :MY_COLUMN).

- **BeanShell 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; `sqlTemplate` is a `org.jumpmind.db.sql.ISqlTemplate` object for querying or updating the database; `channelId` is a reference to the channel on which the transformation is happening; `sourceNode` is a `org.jumpmind.symmetric.model.Node` object that represents the node from where the data came; `targetNode` is a `org.jumpmind.symmetric.model.Node` object that represents the node where the data is being loaded.

- **Java Transform ('java')**: Use Java code in the transform expression that is included in the transform method of a class that extends JavaColumnTransform. The class is compiled whenever the transform expression changes and kept in memory for runtime. The code must return a String for the new value of the column being mapped. The following variables are available: `platform` is the IDatabasePlatform that contains objects for the database platform, such as DatabaseInfo, IDdlReader, IDdlBuilder, and ISqlTemplate. `context` is the DataContext that contains information about current row and the data loader session, such as Batch, Table, and CsvData. `column` is the TransformColumn that contains information from the TRANSFORM_COLUMN configuration. `data` is the TransformedData that contains information about the source and target values being transformed, including the TransformTable. `sourceValues` is a Map<String, String> contain all source column values for the row. `newValue` is a String for the new value of the column. `oldValue` is a String for the old value of the column if the event is an update or delete.

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

- **Mathematical Transform ('math')**: This transformation allows you to perform mathematical equations in the transform expression. 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.

- **Copy If Changed Transform ('copyIfChanged')**: This transformation will copy the value to the target column if the source value has changed. More specifically, the copy will occur if the the old value of the source does not equal the new value. If the old and new are, in fact, equal, then either the column will be ignored or the row will be ignored, based on the setting of the transform expression. If the transform expression is euqal to the string 'IgnoreColumn', the column will be ignored; otherwise, the row will be ignored.

- **Value Map Transform ('valueMap')**: This transformation allows for simple value substitutions through use of the transform expression. The transform expresion should consist of a space separated list of value pairs of the format sourceValue=TargetValue. The column value is used to locate the correct sourceValue, and the transform will change the value into the corresponding targetValue. A sourceValue of * can be used to represent a default target value in the event that the sourceValue is not found. Otherwise, if no default value is found, the result will be null. For example, consider the following transform expression: s1=t1 s2=t2 s3=t3 *=t4. A source value of s1 will be transformed to t1, s2 to t2, s3 to t3, s4 to t4, s5 to t4, null to t4, etc.

- **Clarion Date Time ('clarionDateTime')**: Convert a Clarion date with optional time into a timestamp. Clarion dates are stored as the number of days since December 28, 1800, while Clarion times are stored as hundredths of a second since midnight, plus one. Use a source column of the Clarion date and a target column of the timestamp. Optionally, in the transform expression, enter the name of the Clarion time column.
- **Columns To Rows ('columnsToRowsKey' and 'columnsToRowsValue')**: Convert column values from a single source row into a row per column value at the target. Two column mappings are needed to complete the work: use "columnsToRowsKey" to map which source column is used, and use "columnsToRowsValue" to map the value. The "columnsToRowsKey" mapping requires an expression in the format of "column1=key1,column2=key2" to list the source column names and which key value is stored in the target column. The "columnsToRowsValue" mapping sets the column's value at the target and allows an optional expression: "changesOnly=true" to convert only rows when the old and new values have changed; "ignoreNulls=true" to convert only rows that are not null. For example, column "fieldid" mapped as "columnsToRowsKey" with expression of "user1=1,user2=2" and column "color" mapped as "columnsToRowsValue" would convert a row with columns named "user1" and "user2" containing values "red" and "blue" into two rows with columns "fieldid" and "color" containing a row of "1" and "red" and a row of "2" and "blue".

3.9. Load Filters

New as of SymmetricDS 3.1, SymmetricDS is now capable of taking actions upon the load of certain data via configurable load filters. This new configurable option is in additon to the already existing option of writing a class that implements IDatabaseWriterFilter . A configurable load filter watches for specific data that is being loaded and then takes action based on the load of that data.

Specifying which data to action is done by specifying a souce and target node group (data extracted from this node group, and loaded into that node group), and a target catalog, schema and table name. You can decide to take action on rows that are inserted, updated and/or deleted, and can also further delineate which rows of the target table to take action on by specifying additional criteria in the bean shell script that is executed in response to the loaded data. As an example, old and new values for the row of data being loaded are available in the bean shell script, so you can action rows with a certain column value in old or new data.

The action taken is based on a bean shell script that you can provide as part of the configuration. Actions can be taken at different points in the load process including before write, after write, at batch complete, at batch commit and/or at batch rollback.

3.9.1. Load Filter Configuration Table

SymmetricDS stores its load filter configuration in a single table called LOAD_FILTER . The load filter table allows you to specify the following:

- Load Filter Type ('load_filter_type'): The type of load filter. Today only Bean Shell is supported ('BSH'), but SQL scripts may be added in a future release.
- Source Node Group ('source_node_group_id'): The source node group for which you would like to watch for changes.
- Target Node Group ('target_node_group_id'): The target node group for which you would like to watch for changes. The source and target not groups are used together to identify the node group link for which you would like to watch for changes (i.e. When the Server node group sends data to a Client node group).
- Target Catalog ('target_catalog_name'): The name of the target catalog for which you would like to watch for changes.
- Target Schema ('target_schema_name'): The name of the target schema for which you would like to watch for changes.
- Target Table ('target_table_name'): The name of the target table for which you would like to watch for changes. The target catalog, target schema and target table name are used together to fully qualify the table for which you would like to watch for changes.
- Filter on Update ('filter_on_update'): Determines whether the load filter takes action (executes) on a database update statement.
- Filter on Insert ('filter_on_insert'): Determines whether the load filter takes action (executes) on a database insert statement.
- Filter on Delete ('filter_on_delete'): Determines whether the load filter takes action (executes) on a database delete statement.
- Before Write Script ('before_write_script'): The script to execute before the database write occurs.
- After Write Script ('after_write_script'): The script to execute after the database write occurs.
- Batch Complete Script ('batch_complete_script'): The script to execute after the entire batch completes.
- Batch Commit Script ('batch_commit_script'): The script to execute after the entire batch is committed.
- Batch Rollback Script ('batch_rollback_script'): The script to execute if the batch rolls back.
- Handle Error Script ('handle_error_script'): A script to execute if data cannot be processed.
- Load Filter Order ('load_filter_order'): The order in which load filters should execute if there are multiple scripts pertaining to the same source and target data.

3.9.2. Variables available to Data Load Filters

As part of the bean shell load filters, SymmetricDS provides certain variables for use in the bean shell script. Those variables include:

- Symmetric Engine ('ENGINE'): The Symmetric engine object.
- Source Values ('<COLUMN_NAME>'): The source values for the row being inserted, updated or deleted.
- Old Values ('OLD_<COLUMN_NAME>'): The old values for the row being inserted, updated or deleted.
- Data Context ('CONTEXT'): The data context object for the data being inserted, updated or deleted. .
- Table Data ('TABLE'): The table object for the table being inserted, updated or deleted.

3.9.3. Data Load Filter Example

The following is an example of a load filter that watches a table named TABLE_TO_WATCH being loaded from the Server Node Group to the Client Node Group for inserts or updates, and performs an initial load on a table named "TABLE_TO_RELOAD" for KEY_FIELD on the reload table equal to a column named KEY_FIELD on the TABLE_TO_WATCH table.

```
insert into sym_load_filter
               (LOAD_FILTER_ID, LOAD_FILTER_TYPE, SOURCE_NODE_GROUP_ID,
               TARGET_NODE_GROUP_ID, TARGET_CATALOG_NAME, TARGET_SCHEMA_NAME,
               TARGET_TABLE_NAME, FILTER_ON_UPDATE, FILTER_ON_INSERT, FILTER_ON_DELETE,
               BEFORE_WRITE_SCRIPT, AFTER_WRITE_SCRIPT, BATCH_COMPLETE_SCRIPT,
               BATCH_COMMIT_SCRIPT, BATCH_ROLLBACK_SCRIPT, HANDLE_ERROR_SCRIPT,
               CREATE_TIME, LAST_UPDATE_BY, LAST_UPDATE_TIME, LOAD_FILTER_ORDER,
               FAIL_ON_ERROR) values
               ('TABLE_TO_RELOAD','BSH','Client','Server',NULL,NULL,
               'TABLE_TO_WATCH',1,1,0,null,
               'engine.getDataService().reloadTable(context.getBatch().getSourceNodeId(),
               table.getCatalog(), table.getSchema(), "TABLE_TO_RELOAD","KEY_FIELD='''"
               + KEY_FIELD + "'''");'
               ,null,null,null,null,sysdate,'userid',sysdate,1,1);
```

3.10. Grouplets

As you probably know by now, SymmetricDS stores its single configuration centrally and distributes it to all nodes. By default, a trigger-router is in effect for all nodes in the source node group or target node group. Triggers will be established on each node that is a member of the source node, and changes will be routed to all relevant nodes that are members of the target node group. If, for example, the router routes to "all" nodes, "all" means every node that is in the target node group. This is the default behavior of SymmetricDS.

Once in production, however, you will likely find you need or want to make configuration changes to triggers and routers as new features are rolled out to your network of SymmetricDS nodes. You may, for example, wish to "pilot" a new configuration, containing new synchronizations, only on specific nodes initially, and then increase the size of the pilot over time. SymmetricDS' does provide the ability to specify that only particular trigger-router combinations are applicable to particular nodes for this purpose. It does this by allowing you to define an arbitray collection of nodes, called a "grouplet", and then choosing which trigger-routers apply to the normal set of nodes (the default behavior) and which apply just to nodes in one or more "grouplets". This allows you, essentially, to filter the list of nodes that would otherwise be included as source nodes and/or target nodes. Through the use of grouplets, you can, for example, specify a subset of nodes on which a given trigger would be created. It also allows you to specify a subset of the normal set of nodes a change would be routed to. This behaviour is in addition to, and occurs before, any subsetting or filtering the router might otherwise do.

In its simplest form, a grouplet is just an arbitrary collection of nodes. To define a grouplet, you start by creating a grouplet with a unique id, a description, and a link policy, as defined in GROUPLET. To defined which nodes are members of (or are not members of) a grouplet, you provide a list of external ids of the nodes in GROUPLET_LINK. How those external ids are used varies based on the grouplet link policy. The `grouplet_link_policy` can be either I or E, representing an "inclusive" list of nodes or an "exclusive" list of nodes, respectively. In the case of "inclusive", you'll be listing each external id to be included in the grouplet. In the case of exclusive, all nodes will be included in the grouplet *except* ones which have an external id in the list of external ids.

Once you have defined your grouplet and which nodes are members of a grouplet, you can tie a grouplet to a given trigger-router through the use of TRIGGER_ROUTER_GROUPLET. If a particular trigger-router does not appear in this table, SymmetricDS behaves as normal. If, however, an entry for a particular trigger-router appears in this table, the default behavior is overridden based on the `grouplet_id` and `applies_when` settings. The grouplet id provides the node list, and the `applies_when` indicates whether the grouplet nodes are to be used to filter the source node list, the target node list, or both (settings are "S", "T", and "B", respectively). Nodes that survive the filtering process on as a source will have a trigger defined, and nodes that survive the filtering process as a target are eligible nodes that can be routed to.

3.10.1. Grouplet Example

At this point, an example would probably be useful. Picture the case where you have 100 retail stores (each containing one database, and each a member of the "store" node group) and a central office database (external id of corp, and a member of the "corp" node group). You wish to pilot two new trigger and routers for a new feature on your point-of-sale software (one which moves data from corp to store, and one which moves data from store to corp), but you only want the triggers to be installed on 10 specific stores that represent your "pilot" stores. In this case, the simplest approach would be to define a grouplet with, say, a grouplet id of "pilot". We'd use a grouplet link policy of "inclusive", and list each of the 10 external ids in the GROUPLET_LINK table.

For the trigger-router meant to send data from corp to store, we'd create an entry in TRIGGER_ROUTER_GROUPLET for our grouplet id of "pilot", and we'd specify "T" (target) as the applies-when setting. In this way, the source node list is not filtered, but the target node list used during routing will filter the potential target nodes to just our pilot stores. For the trigger-router meant to send data from a pilot store back to corp, we would have the grouplet apply when the node is in the source node list (i.e., `applies_when` will be "S"). This will cause the trigger to only be created for stores in the pilot list and not other stores.

An important thing to mention in this example: Since your grouplet only included the store nodes, you can't simply specify "both" for the applies when setting. For the corp-to-store trigger, for example, if you had said "both", no trigger would have been installed in corp since the grouplet nodes represent all possible source nodes as well as target nodes, and "corp" is not in the list! The same is true for the store to corp trigger-router as well. You could, however, use "both" as the applies when if you had included the "corp" external id in with the list of the 10 pilot store external ids.

3.11. Parameters

Parameters can be used to help tune and configure your SymmetricDS configuration. Parameters can be set for an individual node or for all nodes in your network.

See Appendix B, *Parameters* (p. 121), for a complete list of parameters.

3.12. Export

3.13. Import

3.14. Uninstall

Chapter 4. Manage

4.1. 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.

4.2. Creating Nodes

Nodes are 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.

4.2.1. 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.

4.2.2. Initial Loads

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 pare down the data that is selected out of the database.

An initial load cannot 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. You can configure SymmetricDS to automatically perform an initial load when a node registers by setting the parameter `auto.reload` to true. Regardless of how the initial load is initiated, the next time the source node routes data, 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 occurring 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 . If the `initial_load_order` column contains a negative value the associated table will *NOT* be loaded. If the `initial_load_order` column contains the same value for multiple tables, SymmetricDS will attempt to order the tables according to foreign key constraints. If there are cyclical constraints, then foreign keys might need to be turned off or the initial load will need to be manually configured based on knowledge of how the data is structured.

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.

4.2.2.1. Target table prep for initial load

There are several parameters that can be used to specify what, if anything, should be done to the table on the target database just prior to loading the data. Note that the parameters below specify the desired behavior for all tables in the initial load, not just one.

- `initial.load.delete.first / initial.load.delete.first.sql`

  By default, an initial load will not delete existing rows from a target table before loading the data. If a delete is desired, the parameter `initial.load.delete.first` can be set to true. If true, the command found in `initial.load.delete.first.sql` will be run on each table prior to loading the data. The default value for `initial.load.delete.first.sql` is `delete from %s` , but could be changed if needed. Note that additional reload batches are created, in the correct order, to achieve the delete.

- `initial.load.create.first`

  By default, an initial load will not create the table on the target if it doesn't alleady exist. If the desired behavior is to create the table on the target if it is not present, set the parameter `intial.load.create.first` to true. SymmetricDS will attempt to create the table and indexes on the target database before doing the initial load. (Additional batches are created to represent the table schema).

4.2.2.2. Loading subsets of data

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. The `$(externalId)` token can be used for subsetting the data in the where clause.

In cases where routing is done using a feature like Section 3.6.4, Subselect Router (p. 24) , an `initial_load_select` clause matching the subselect's criteria would be a more efficient approach. Some routers will check to see if the `initial_load_select` clause is provided, and they will *not* execute assuming that the more optimal path is using the `initial_load_select` statement.

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.

4.2.2.3. Splitting an Initial Load for a Table Across Multiple Batches

By default, all data for a given table will be initial loaded in a single batch, regardless of the max batch size parameter on the reload channel. That is, for a table with one million rows, all rows for that table will be initial loaded and sent to the destination node in a single batch. For large tables, this can result in a batch that can take a long time to extract and load.

Initial loads for a table can be broken into multiple batches by specifying `initial.load.use.extract.job.enabled` to true. This parameter allows SymmetricDS to pre-extract initial load batches versus having them extracted when the batch is pulled or pushed. When using this parameter, there are two ways to tell SymmetricDS the number of batches to create for a given table. The first is to specify a positive integer in the initial_load_batch_count column on TRIGGER_ROUTER. This number will dictate the number of batches created for the initial load of the given table. The second way is to specify 0 for initial_load_batch_count on TRIGGER_ROUTER and specify a max_batch_size on the reload channel in CHANNEL. When 0 is specified for initial_load_batch_count, SymmetricDS will execute a count(*) query on the table during the extract process and create N batches based on the total number of records found in the table divided by the max_batch_size on the reload channel.

4.2.2.4. Reverse Initial Loads

The default behavior for initial loads is to load data from the registration server or parent node, to a client node. Occasionally, there may be need to do a one-time intial load of data in the opposite or "reverse" direction, namely from a client node to the registration node. To achieve this, set the parameter `auto.reload.reverse` to be true, *but only for the specific node group representing the client nodes* . This will cause a onetime reverse load of data, for tables configured with non-negative initial load orders, to be batched at the point when registration of the client node is occurring. These batches are then sent to the parent or registration node. This capability might be needed, for example, if there is data already present in the client that doesn't exist in the parent but needs to.

4.2.3. Data Reloads

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

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 transaction 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. 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).

4.3. Jobs

Work done by SymmetricDS is initiated by jobs. Jobs are tasks that are started and scheduled by a job manager. Jobs are enabled by the `start.{name}.job` property. Most jobs are enabled by default. The frequency at which a job runs in controlled by one of two properties: `job.{name}.period.time.ms` or `job.{name}.cron` . If a valid cron property exists in the configuration, then it will be used to schedule the job. Otherwise, the job manager will attempt to use the period.time.ms property.

The frequency of jobs can be configured in either the engines properties file or in PARAMETER . When managed in PARAMETER the frequency properties can be changed in the registration server and when the updated settings sync to the nodes in the system the job manager will restart the jobs at the new frequency settings.

SymmetricDS utilizes Spring's CRON support, which includes seconds as the first parameter. This differs from the typical Unix-based implementation, where the first parameter is usually minutes. For example, `*/15 * * * * *` means every 15 seconds, not every 15 minutes. See Spring's documentation for more details.

Some jobs cannot be run in parallel against a single node. When running on a cluster these jobs use the LOCK table to get an exclusive semaphore to run the job. In order to use this table the `cluster.lock.enabled` must be set to true.

The three main jobs in SymmetricDS are the route, push and pull jobs. The route job decides what captured data changes should be sent to which nodes. It also decides what captured data changes should be transported and loaded together in a batch. The push and pull jobs are responsible for initiating HTTP communication with linked nodes to push or pull data changes that have been routed.

4.3.1. Route Job

After data is captured in the DATA table, it is routed to specific nodes in batches by the *Route Job* . It is a single background task that inserts into DATA_EVENT and OUTGOING_BATCH .

The job processes each enabled channel, one at a time, collecting a list of data ids from DATA which have not been routed (see Section 4.3.1.1, Data Gaps (p. 47) for much more detail about this step), 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. Note that, for the `default` and `transactional` 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".

The route job will respect the `max_batch_size` on OUTGOING_BATCH . If the max batch size is reached before the end of a database tranaction and the batch algorithm is set to something other than `nontransactional` the batch may exceed the specified max size.

The route job delegates to a router defined by the `router_type` and configured by the `router_expression` in the ROUTER table. Each router that has a `source_node_group_id` that matches the current node's source node group id and is linked to the TRIGGER that captured the data gets an opportunity to choose a list of nodes the data should be sent to. Data can only be routed to nodes that belong to the router's `target_node_group_id` .

4.3.1.1. 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.

4.3.2. Push and Pull Jobs for Database changes

After database-change 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.

The maximum number of batches to extract per synchronization is controlled by `max_batch_to_send` on the CHANNEL table. There is also a setting that controls the max number of bytes to send in one synchronization. If SymmetricDS has extracted the more than the number of bytes configured by the `transport.max.bytes.to.sync` parameter, then it will finish extracting the current batch and finish synchronization so the client has a chance to process and acknowlege the "big" batch. This may happen before the configured max number of batches has been reached.

Both the push and pull jobs can be configured to push and pull multiple nodes in parallel. In order to take advantage of this the `pull.thread.per.server.count` or `push.thread.per.server.count` should be adjusted (from their default value of 10) to the number to the number of concurrent push/pulls you want to occur per period on each SymmetricDS instance. Push and pull activity is recorded in the NODE_COMMUNICATION table. This table is also used to lock push and pull activity across multiple servers in a cluster.

SymmetricDS also provides the ability to configure windows of time when synchronization is allowed. This is done using the NODE_GROUP_CHANNEL_WND 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.

Node communication over HTTP is represented in the following figure.

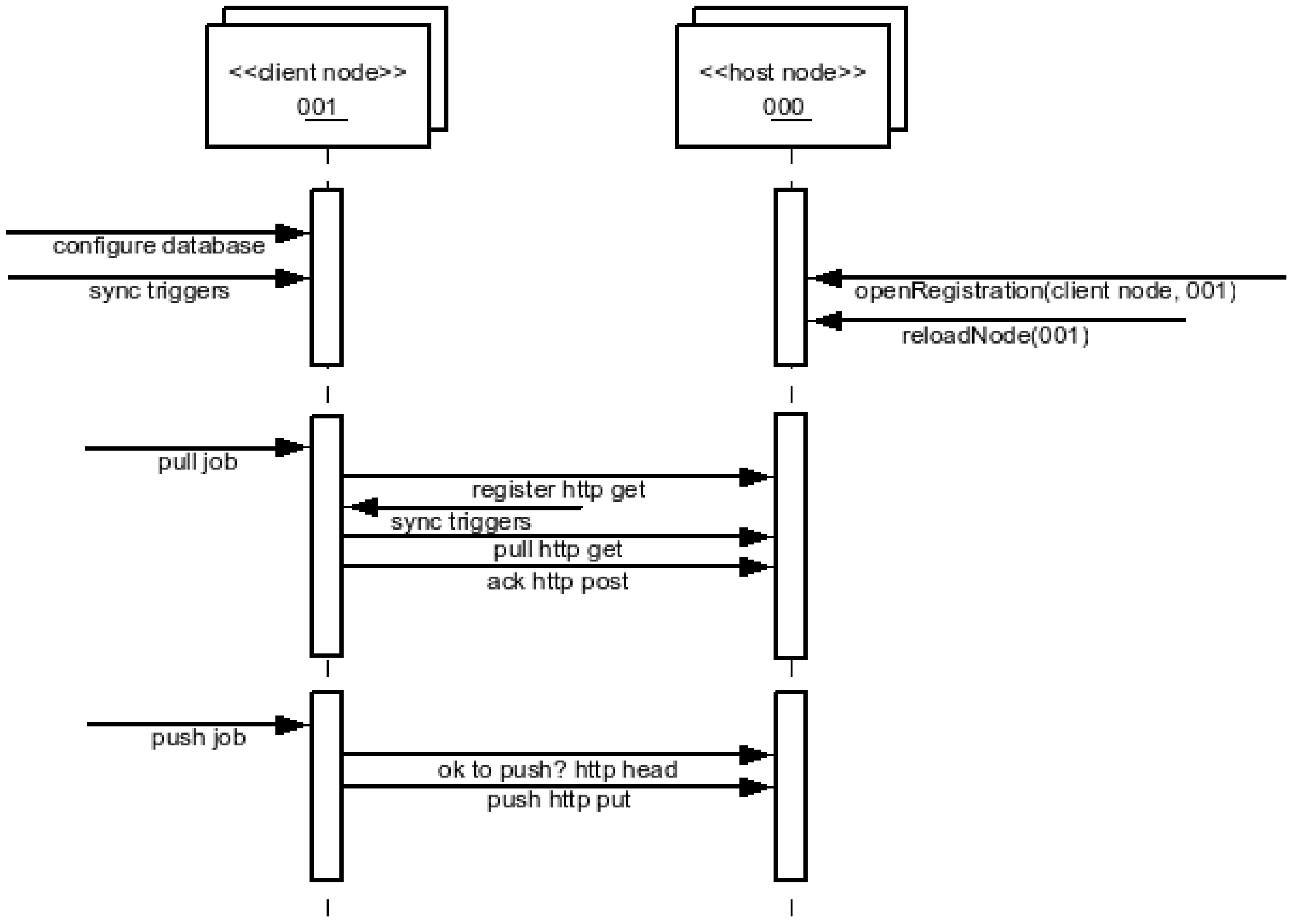


Figure 4.1. Node Communication

4.3.3. File Sync Push and Pull Jobs

The File Sync Push and Pull jobs (introduced in version 3.5) are responsible for synchronizing file changes. These jobs work with batches on the `filesync` channel and create ZIP files of changed files to be sent and applied on other nodes. The parameters `job.file.sync.push.period.time.ms` and `job.file.sync.pull.period.time.ms` control how often the jobs runs, which default to every 60 seconds. See also Section 4.3, Jobs (p. 46) and Section 3.5.2, Operation (p. 17) .

4.3.4. File System Tracker Job

The File System Tracker job (introduced in version 3.5) is responsible for monitoring and recording the events of files being created, modified, or deleted. It records the current state of files to the FILE_SNAPSHOT table. The parameter `job.file.sync.tracker.cron` controls how often the job runs, which defaults to every 5 minutes. See also Section 4.3, Jobs (p. 46) and Section 3.5, File Triggers / File Synchronization (p. 16) .

4.3.5. 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.

4.3.6. Purge Jobs

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 timing of the three purge jobs (incoming, outgoing, and data gaps) is controlled by a cron expression as specified by the following properties: `job.purge.outgoing.cron` , `job.purge.incoming.cron` , and `job.purge.datagaps.cron` . The default is `0 0 0 * * *` , or once per day at midnight.

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.

4.4. Outgoing Batches

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).

Analyzing and resolving issues can take place on the outgoing or incoming side. The techniques for analysis are slightly different in the two cases, however, due to the fact that the node with outgoing batch data also has the data and data events associated with the batch in the database. On the incoming node, however, all that is available is the incoming batch header and data present in an incoming error table.

4.4.1. 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 OUTGOING_BATCH 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 4.4.1, Analyzing the Issue (p. 51) 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'
                and node_id='YYYYY');
```

where XXXXXX is the batch id and YYYYY is the node id of 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.

4.4.2. 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.

4.5. Incoming Batches

4.5.1. Analyzing the Issue

Analysis using an incoming batch is different than that of outgoing batches. For incoming batches, you will rely on two tables, INCOMING_BATCH and INCOMING_ERROR. The first step in analyzing the cause of an incoming failed batch is to locate information about the batch, starting with INCOMING_BATCH To locate batches in error, use:

```
select * from sym_incoming_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`. Note that this is the batch number of the outgoing batch on the outgoing node.
- The node the batch is being sent from, available in column `NODE_ID`.
- The channel to which the batch belongs, available in column `CHANNEL_ID`. All subsequent batches on this channel from this node will be held until the error condition is resolved.
- The data_id that was being processed when the batch failed, 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.

For incoming batches, we do not have data and data event entries in the database we can query. We do, however, have a table, INCOMING_ERROR, which provides some information about the batch.

```
select * from sym_incoming_error
                where batch_id='XXXXXX' and node_id='YYYYY';
```

where XXXXXX is the batch id and YYYYY is the node 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 `TARGET_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 column names of the table, available in column `COLUMN_NAMES`,
- The primary key column names of the table, available in column `PK_COLUMN_NAMES`,

4.5.2. Resolving the Issue

For batches in error, from the incoming side you'll also have to decide the best course of action to fix the issue. Incoming batch errors *that are in conflict* can by fixed by taking advantage of two columns in INCOMING_ERROR which are examined each time batches are processed. The first column, `resolve_data` if filled in will be used in place of `row_data`. The second column, `resolve_ignore` if set will cause this particular data item to be ignored and batch processing to continue. This is the same two columns used when a manual conflict resolution strategy is chosen, as discussed in Section 3.7.1, Conflict Detection and Resolution (p. 28) .

4.6. Staging Area

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.

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
```

4.7. Logging

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.

There is a command line option to turn on preconfigured debugging levels. When the `--debug` option is used the `conf/debug-log4j.xml` is used instead of log4j.xml.

SymmetricDS proxies all of its logging through SLF4J. When deploying to an application server or if Log4J is not being leveraged, then the general rules for for SLF4J logging apply.

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.

Conflict resolution strategies can be configured for specific links and/or sets of tables.

5.1.2. Multi-Tiered Synchronization

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 be 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 can only pull and push data to other nodes that are represented in the node's NODE table and in cases where that node's `sync_enabled` column is set to 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 sends an HTTP redirect to the `sync_url` of the node that matches the `registration_node_id`.

Important

Please see Section 4.2.2, Initial Loads (p. 42) for important details around initial loads and registration when using registration redirect.

5.2. 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

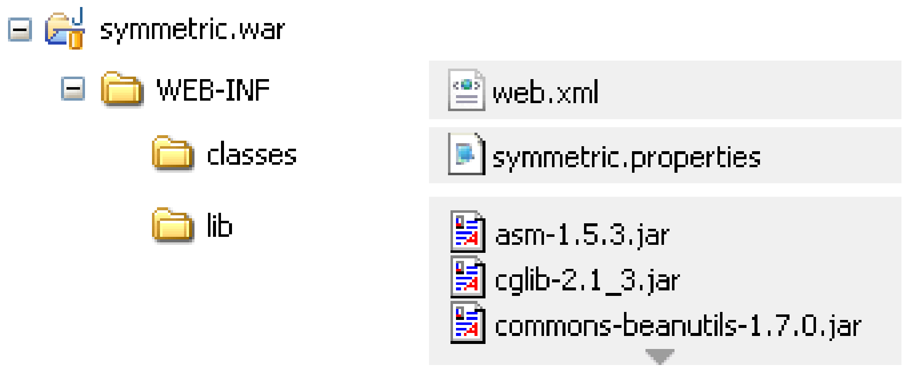
  This option means you must write a wrapper Java program that runs SymmetricDS. You would probably use Jetty web server, which is also embeddable. You could bring up an embedded database like Derby or H2. You could configure the web server, database, or SymmetricDS to do whatever you needed, but it's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

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.

5.2.1. Web Archive (WAR)

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 `symadmin` utility and the `create-war` subcommand. 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/symadmin -p my-symmetric-ds.properties create-war /some/path/to/symmetric-ds.war

5.2.2. Embedded

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

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

public class StartSymmetricEngine {

    public static void main(String[] args) throws Exception {

        SymmetricWebServer node = new SymmetricWebServer(
                                   "classpath://my-application.properties", "conf/web_dir");

        // 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. The configuration properties file, `my-application.properties`, is packaged in the application to provide properties that override the SymmetricDS default values. The second parameter to the constructor points to the web directory. The default location is `../web`. In this example the web directory is located at `conf/web_dir`. The web.xml is expected to be found at `conf/web_dir/WEB-INF/web.xml`.

5.2.3. Standalone

The `sym` command line utility starts a standalone web server with SymmetricDS pre-deployed. The standalone server uses an embedded instance of the Jetty application server to handle web requests. The web server can be configured using command line options or the web server can be configured by changing properties in the `conf/symmetric-server.properties` file.

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

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

Even though the port and properties settings can be passed in on the command line, the preferred configuration approach is to put each hosted node's properties file in the `engines` directory and to modify port settings and enable secure mode using the `conf/symmetric-server.properties`.

It is also suggested that SymmetricDS be configured to run as a service according to the instructions for your platform as documented in the following section.

5.3. Running SymmetricDS as a Service

SymmetricDS can be configured to start automatically when the system boots, running as a Windows service or Linux/Unix daemon. A wrapper process starts SymmetricDS and monitors it, so it can be restarted if it runs out of memory or exits unexpectedly. The wrapper writes standard output and standard error to the `logs/wrapper.log` file.

5.3.1. Running as a Windows Service

To install the service, run the following command as Administrator:

```
bin\sym_service.bat install
```

Most configuration changes do not require the service to be re-installed. To un-install the service, run the following command as Administrator:

```
bin\sym_service.bat uninstall
```

To start and stop the service manually, run the following commands as Administrator:

```
bin\sym_service.bat start
bin\sym_service.bat stop
```

5.3.2. Running as a Linux/Unix daemon

An init script is written to the system `/etc/init.d` directory. Symbolic links are created for starting on run levels 2, 3, and 5 and stopping on run levels 0, 1, and 6. To install the script, running the following command as root:

```
bin/sym_service install
```

Most configuration changes do not require the service to be re-installed. To un-install the service, run the following command as root:

```
bin/sym_service uninstall
```

To start and stop the service manually, run the following commands:

```
bin/sym_service start
bin/sym_service stop
```

5.4. 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 2.1, Engine Files (p. 9)) 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.

5.5. 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:

```
symadmin -e {engine name} encrypt-text text-to-encrypt
```

or

```
symadmin -p {properties file} encrypt-text text-to-encrypt
```

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
```

```
symadmin -providerClass 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.security.SecurityService
```

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

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

5.6. 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.

5.6.1. 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

5.6.2. 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" />
```

5.6.3. 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.

5.6.4. 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.

5.7. 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 Web Server also supports Basic Authentication. It can be enabled by passing the following arguments to the startup program

--http-basic-auth-user
username for basic authentication [Default:]

--http-basic-auth-password
password for basic authentication [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.

5.8. Data Loaders

SymmetricDS supports the concept of pluggable data loaders. A data loader defines how data is loaded into a target datasource. The default data loader for SymmetricDS loads data to the relational database that is represented by the SymmetricDS node. Data loaders do not always have to load into the target relational database. They can write to a file, a web service, or any other type of non-relational data source. Data loaders can also use other techniques to increase performance of data loads into the target relation database. Data loaders are pluggable at the CHANNEL level. They are configured by setting the `data_loader_type` on the channel table.

5.8.1. Bulk Data Loaders

To use the preconfigured bulk data loaders, you set the `data_loader_type` on a channel to one of the following:

- mysql_bulk
- mssql_bulk
- postgres_bulk
- oracle_bulk

Tables that should be data loaded should be configured to use this channel. Many times, a reload channel will be set to bulk load to increase the performance of an initial load.

5.8.2. MongoDB

The MongoDB data loader maps relational database rows to MongoDB documents in collections. To use the preconfigured MongoDB data loader, you set the `data_loader_type` to *MongoDB* on a channel. Tables that should be synchronized to MongoDB should be configured to use this channel. In order to point it to a MongoDB instance set the following properties in the engines properties file.

```
mongodb.username=xxxx
mongodb.password=xxxx
mongodb.host=xxxx
mongodb.port=xxxx
mongodb.default.databasename=default
```

By default, the catalog or schema passed by SymmetricDS will be used for the MongoDB database name. The table passed by SymmetricDS will be used as the MongoDB collection name. If the catalog or schema are not set, the default database name property is used as the database name.

The _id of the MongoDB document will be the primary key of the database record. If the table has a composite primary key, then the _id will be an embedded document that has name value pairs of the composite key. The body of the document will be name value pairs of the table column name and table row value.

SymmetricDS uses the MongoDB Java Driver to upsert documents.

SymmetricDS transforms can be used to transform the data. If a complex mapping is required that is not supported by transforms, then the `IDBObjectMapper` can be implemented and a new `MongoDataLoaderFactory` can be wired up as an extension point.

5.9. 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/
```

In order to use jconsole, you must enable JMX remote management in the JVM. You can edit the startup scripts to set the following system parameters.

```

          -Dcom.sun.management.jmxremote.port=31417
          -Dcom.sun.management.jmxremote.authenticate=false
          -Dcom.sun.management.jmxremote.ssl=false

```

More details about enabling JMX for JConsole can be found here.

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

The management interfaces under SymmetricDS are organized as follows:

- Node - administrative operations
- Parameters - access to properties set through the parameter service

5.10. 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 XmlPublisherDatabaseWriterFilter is a IDatabaseWriterFilter that may be configured to publish specific tables as an XML message to a JMS provider. See Section 6.1, Extension Points (p. 71) 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.XmlPublisherDatabaseWriterFilter">
        <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 XmlPublisherDatabaseWriterFilter 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 similar 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.

Chapter 6. Developer

This chapter focuses on a variety of ways for developers to build upon and extend some of the existing features found within SymmetricDS.

6.1. Extension Points

SymmetricDS has a pluggable architecture that can be extended. A Java class that implements the appropriate extension point interface, can implement custom logic and change the behavior of SymmetricDS to suit special needs. All supported extension points extend the `IExtensionPoint` interface. The available extension points are documented in the following sections.

When SymmetricDS starts up, the `ExtensionPointManager` searches a Spring Framework context for classes that implement the `IExtensionPoint` interface, then creates and registers the class with the appropriate SymmetricDS component.

Extensions should be configured in the `conf/symmetric-extensions.xml` file as Spring beans. The jar file that contains the extension should be placed in the web/WEB-INF/lib directory.

If an extension point needs access to SymmetricDS services or needs to connect to the database it may implement the `ISymmetricEngineAware` interface in order to get a handle to the `ISymmetricEngine`.

The `INodeGroupExtensionPoint` interface may be optionally implemented to indicate that a registered extension point should only be registered with specific node groups.

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

6.1.1. 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;
    }

}
```

6.1.2. IDatabaseWriterFilter

Data can be filtered or manipulated before it is loaded into the target database. A filter can change the data in a column, save it somewhere else or do something else with the data entirely. 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, for example, might be to route credit card data to a secure database and blank it out as it loads into a less-restricted reporting database.

A `DataContext` is passed to each of the callback methods. A new context is created for each synchronization. The context provides a mechanism to share data during the load of a batch between different rows of data that are committed in a single database transaction.

The filter also provides callback methods for the batch lifecycle. The `DatabaseWriterFilterAdapter` may be used if not all methods are required.

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

```
public class MyFilter extends DatabaseWriterFilterAdapter {

    @Override
    public boolean beforeWrite(DataContext context, Table table, CsvData data) {
        if (table.getName().equalsIgnoreCase("CREDIT_CARD_TENDER")
                && data.getDataEventType().equals(DataEventType.INSERT)) {
            String[] parsedData = data.getParsedData(CsvData.ROW_DATA);
            // blank out credit card number
            parsedData[table.getColumnIndex("CREDIT_CARD_NUMBER")] = null;
        }
        return true;
    }
}
```

The filter class should be specified in `conf/symmetric-extensions.xml` as follows.

```
<?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>
```

6.1.3. IDatabaseWriterErrorHandler

Implement this extension point to override how errors are handled. You can use this extension point to ignore rows that produce foreign key errors.

6.1.4. IDataLoaderFactory

Implement this extension point to provide a different implementation of the `org.jumpmind.symmetric.io.data.IDataWriter` that is used by the SymmetricDS data loader. Data loaders are configured for a channel. After this extension point is registered it can be activated for a CHANNEL by indicating the data loader name in the `data_loader_type` column.

SymmetricDS has two out of the box extensions of IDataLoaderFactory already implemented in its PostgresBulkDataLoaderFactory and OracleBulkDataLoaderFactory classes. These extension points implement bulk data loading capabilities for Oracle, Postgres and Greenplum dialects. See Appendix C. Database Notes for details.

Another possible use of this extension point is to route data to a NOSQL data sink.

6.1.5. 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.

6.1.6. 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.

6.1.7. 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.

6.1.8. IColumnTransform

This extension point allows custom column transformations to be created. There are a handful of out-of-the-box implementations. If any of these do not meet the column transformation needs of the application, then a custom transform can be created and registered. It can be activated by referencing the column transform's name `transform_type` column of TRANSFORM_COLUMN

6.1.9. INodeIdCreator

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 creator per SymmetricDS instance (Please note that the node creator extension has replaced the node generator extension).

6.1.10. ITriggerCreationListener

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

6.1.11. 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.

6.1.12. 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 programmatically decide which nodes data should be routed to.

6.1.13. IHeartbeatListener

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

6.1.14. IOfflineClientListener

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

6.1.15. IOfflineServerListener

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

6.1.16. INodePasswordFilter

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

6.2. Embedding in Android

SymmetricDS now has its web-enabled, fault-tolerant, database synchronization software available on the Android mobile computing platform. The Android client follows all of the same concepts and brings to Android all of the same core SymmetricDS features as the full-featured, Java-based SymmetricDS client. The Android client is a little bit different in that it is not a stand-alone application, but is designed to be referenced as a library to run in-process with an Android application requiring synchronization for its SQLite database.

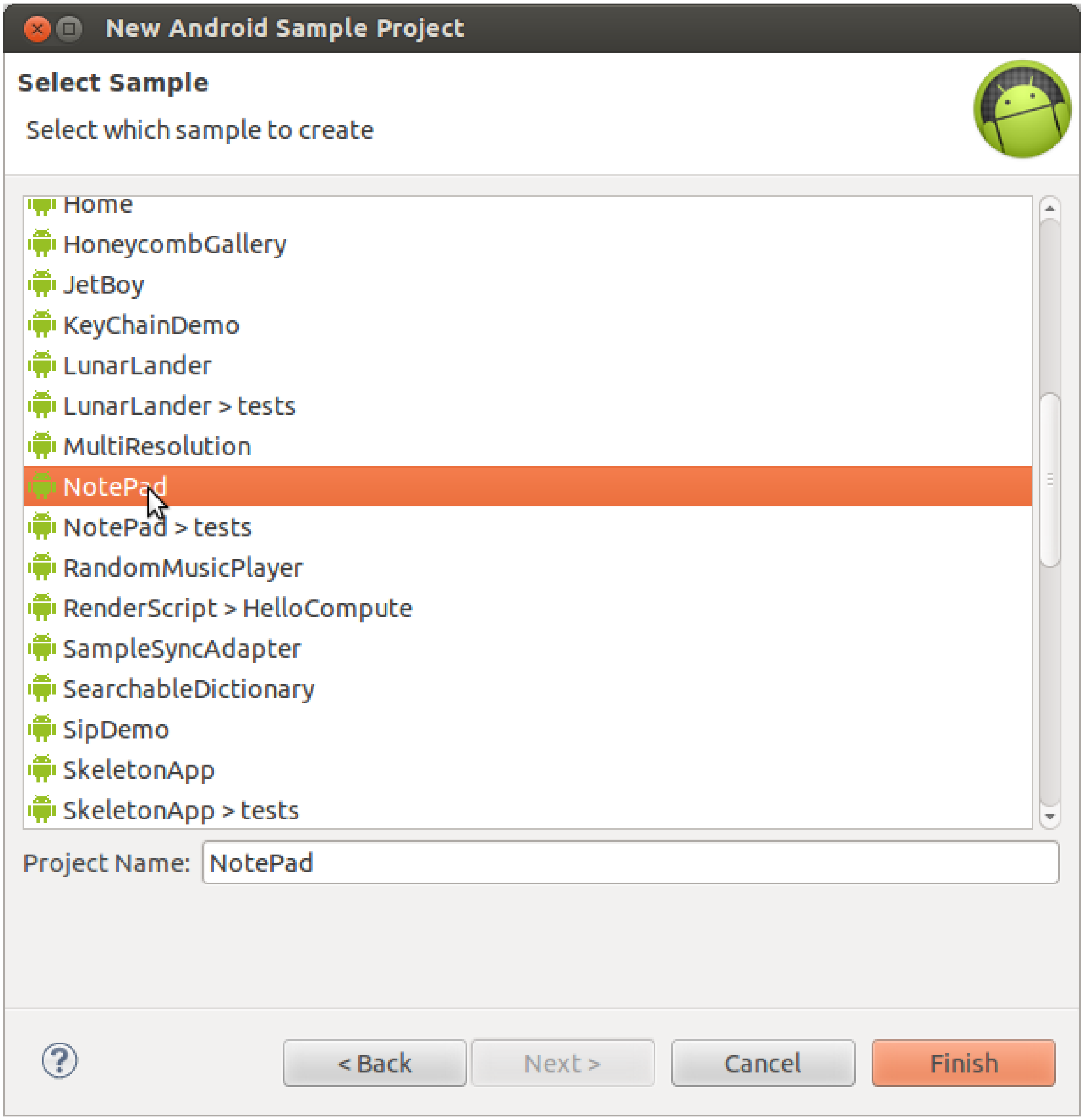
By using SymmetricDS, mobile application development is simplified, in that the mobile application developer can now focus solely on interacting with their local SQLite database. SymmetricDS takes care of capturing and moving data changes to and from a centralized database when the network is available

The same core libraries that are used for the SymmetricDS server are also used for Android. SymmetricDS's overall footprint is reduced by eliminating a number of external dependencies in order to fit better on an Android device. The database access layer is abstracted so that the Android specific database access layer could be used. This allows SymmetricDS to be efficient in accessing the SQLite database on the Android device.

In order to convey how to use the SymmetricDS Android libraries, the example below will show how to integrate SymmetricDS into the NotePad sample application that comes with the Android ADK.

The NotePad sample application is a very simple task list application that persists *notes* to a SQLite database table called Notes. Eclipse 3.7.2 and Android ADK 20.0.3 were used for this example.

Create the NotePad project. You do this by adding a new Android Sample Project. Select the NotePad project.



Project Name: NotePad

Figure 6.1. New Sample NotePad Project

SymmetricDS for Android comes as a zip file of Java archives (jar files) that are required by the SymmetricDS client at runtime. This zip file ()symmetric-ds-3.4.7-android.zip) can be downloaded from the SymmetricDS.org website. The first step to using SymmetricDS in an Android application is to unzip the jar files into a location where the project will recognize them. The latest Android SDK and the Eclipse ADK requires that these jar files be put into a *libs* directory under the Android application project.

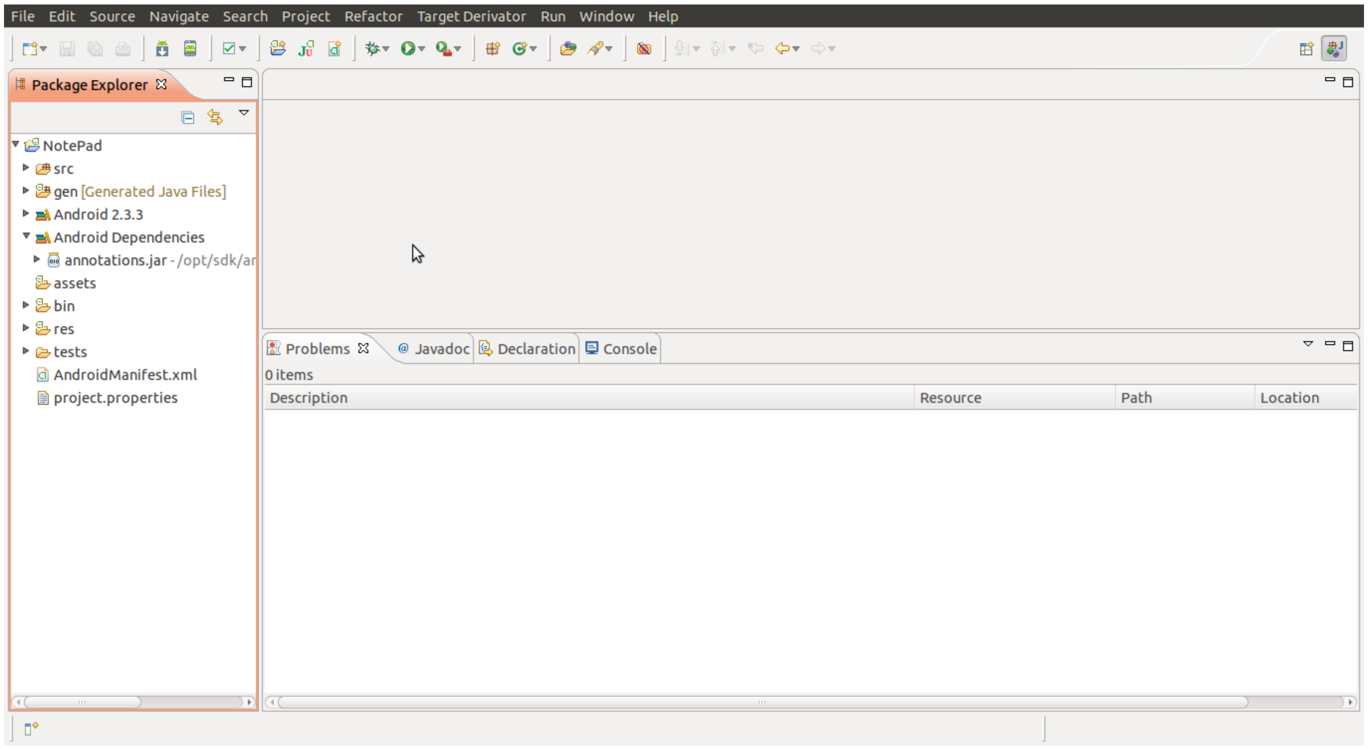


Figure 6.2. New Sample NotePad Project

Unzip the symmetric-ds-x.x.x-android.zip file to the NotePad project directory. Refresh the NotePad project in Eclipse. You should end up with a libs directory that is automatically added to the Android Dependencies.

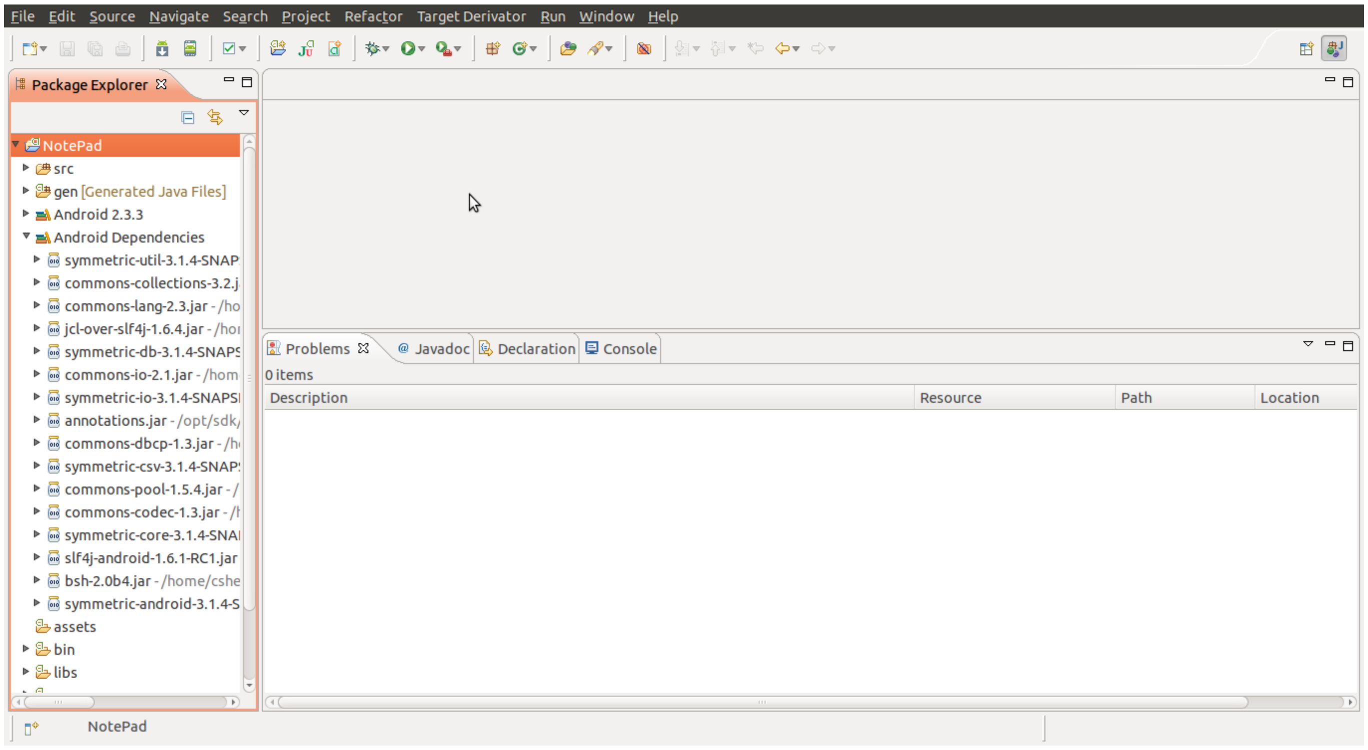

File Edit Source Navigate Search Project Refactor Target Derivator Run Window Help
Package Explorer
NotePad
src
gen [Generated Java Files]
Android 2.3.3
Android Dependencies
symmetric-util-3.1.4-SNAP
commons-collections-3.2.j
commons-lang-2.3.jar - /ho
jcl-over-slf4j-1.6.4.jar - /hoi
symmetric-db-3.1.4-SNAPS
commons-io-2.1.jar - /hom
symmetric-io-3.1.4-SNAPSI
annotations.jar - /opt/sdk/
commons-dbcp-1.3.jar - /h
symmetric-csv-3.1.4-SNAP
commons-pool-1.5.4.jar - /
commons-codec-1.3.jar - /
symmetric-core-3.1.4-SNAI
slf4j-android-1.6.1-RC1.jar
bsh-2.0b4.jar - /home/cshe
symmetric-android-3.1.4-S
assets
bin
libs
Problems
Javadoc
Declaration
Console
0 items
Description
Resource
Path
Location
NotePad

Figure 6.3. Jar Files Added to Libs

The Android version of the SymmetricDS engine is a Java class that can be instantiated directly or wired into an application via a provided Android service. Whether you are using the service or the engine directly you need to provide a few required startup parameters to the engine:

- **SQLiteOpenHelper** It is best (but not required) if the SQLiteOpenHelper is shared with the application that will be sharing the SQLite database. This core Android Java class provides software synchronization around the access to the database and minimizes locking errors.
- **registrationUrl** This is the URL of where the centralized SymmetricDS instance is hosted.
- **externalId** This is the identifier that can be used by the centralized SymmetricDS server to identify whether this instance should get data changes that happen on the server. It could be the serial number of the device, an account username, or some other business concept like store number.
- **nodeGroupId** This is the group id for the mobile device in the synchronization configuration. For example, if the nodeGroupId is 'handheld', then the SymmetricDS configuration might have a group called 'handheld' and a group called 'corp' where 'handheld' is configured to push and pull data from 'corp.'
- **properties** Optionally tweak the settings for SymmetricDS.

In order to integrate SymmetricDS into the NotePad application, the Android-specific SymmetricService will be used, and we need to tell the Android application this by adding the service to the AndroidManifest.xml file. Add the following snipped to the Manifest as the last entry under the <application> tag.

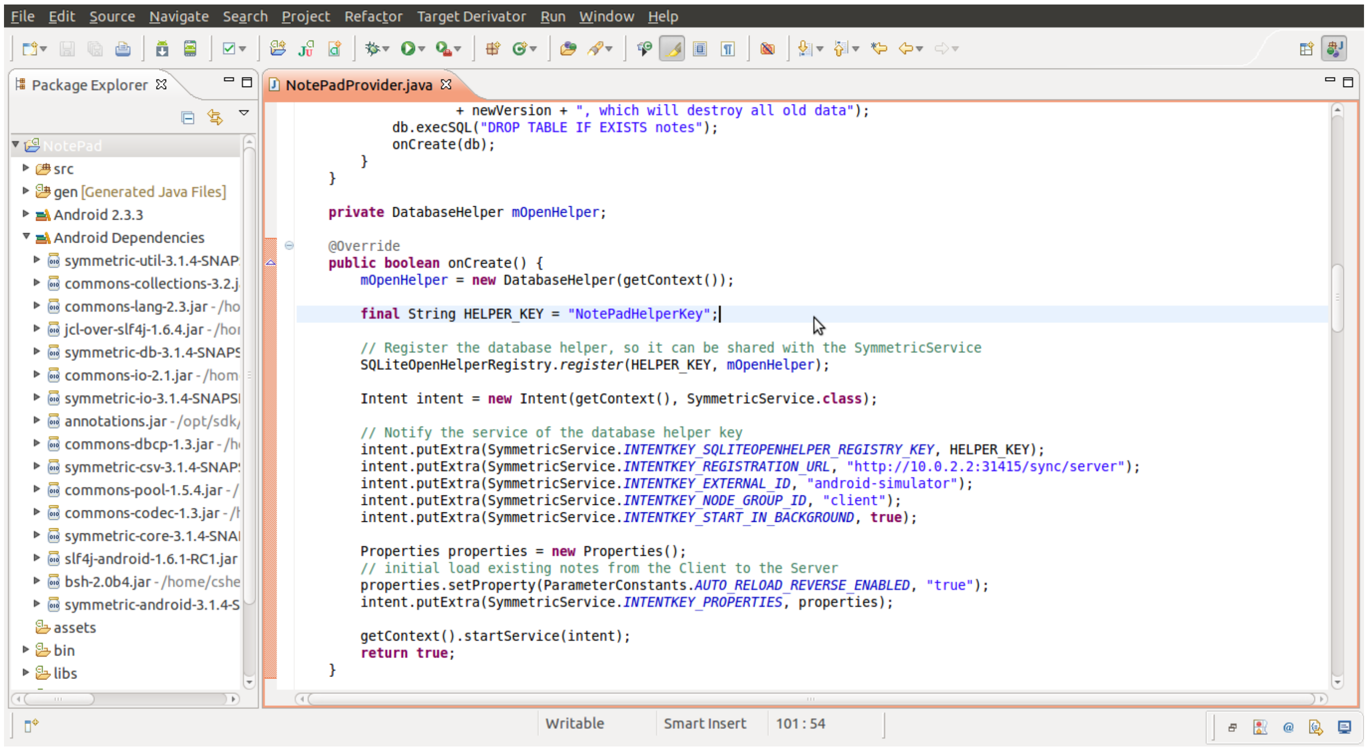
```
<service android:name="org.jumpmind.symmetric.android.SymmetricService"
android:enabled="true" >
    <intent-filter>
                  <action android:name="org.jumpmind.symmetric.android.
                  SymmetricService" />
          </intent-filter>
</service>
```

The other change required in the Manifest is to give the application permission to use the Internet. Add this as the first entry in the AndroidManifest.xml right before the <application> tag.

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

The only additional change needed is the call to start the service in the application. The service needs to be started manually because we need to give the application a chance to provide configuration information to the service.

In NotePadProvider.java add the following code snippet in the onCreate method.


File Edit Source Navigate Search Project Refactor Target Derivator Run Window Help
Package Explorer
NotePad
src
gen [Generated Java Files]
Android 2.3.3
Android Dependencies
symmetric-util-3.1.4-SNAP
commons-collections-3.2.j
commons-lang-2.3.jar - /ho
jcl-over-slf4j-1.6.4.jar - /ho
symmetric-db-3.1.4-SNAPS
commons-io-2.1.jar - /hom
symmetric-io-3.1.4-SNAPS
annotations.jar - /opt/sdk/
commons-dbcp-1.3.jar - /h
symmetric-csv-3.1.4-SNAP
commons-pool-1.5.4.jar - /
commons-codec-1.3.jar - /
symmetric-core-3.1.4-SNA
slf4j-android-1.6.1-RC1.jar
bsh-2.0b4.jar - /home/cshe
symmetric-android-3.1.4-S
assets
bin
libs
NotePadProvider.java
Writable
Smart Insert
101 : 54

```
                + newVersion + ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS notes");
            onCreate(db);
        }
    }

    private DatabaseHelper mOpenHelper;

    @Override
    public boolean onCreate() {
        mOpenHelper = new DatabaseHelper(getContext());

        final String HELPER_KEY = "NotePadHelperKey";

        // Register the database helper, so it can be shared with the SymmetricService
        SQLiteOpenHelperRegistry.register(HELPER_KEY, mOpenHelper);

        Intent intent = new Intent(getContext(), SymmetricService.class);

        // Notify the service of the database helper key
        intent.putExtra(SymmetricService.INTENTKEY_SQLITEOPENHELPER_REGISTRY_KEY, HELPER_KEY);
        intent.putExtra(SymmetricService.INTENTKEY_REGISTRATION_URL, "http://10.0.2.2:31415/sync/server");
        intent.putExtra(SymmetricService.INTENTKEY_EXTERNAL_ID, "android-simulator");
        intent.putExtra(SymmetricService.INTENTKEY_NODE_GROUP_ID, "client");
        intent.putExtra(SymmetricService.INTENTKEY_START_IN_BACKGROUND, true);

        Properties properties = new Properties();
        // initial load existing notes from the Client to the Server
        properties.setProperty(ParameterConstants.AUTO_RELOAD_REVERSE_ENABLED, "true");
        intent.putExtra(SymmetricService.INTENTKEY_PROPERTIES, properties);

        getContext().startService(intent);
        return true;
    }
```

Figure 6.4. NotePadProvider.java

```
final String HELPER_KEY = "NotePadHelperKey";

// Register the database helper, so it can be shared with the SymmetricService
SQLiteOpenHelperRegistry.register(HELPER_KEY, mOpenHelper);

Intent intent = new Intent(getContext(), SymmetricService.class);

// Notify the service of the database helper key
intent.putExtra(SymmetricService.INTENTKEY_SQLITEOPENHELPER_REGISTRY_KEY,
HELPER_KEY);
intent.putExtra(SymmetricService.INTENTKEY_REGISTRATION_URL,
"http://10.0.2.2:31415/sync/server");
intent.putExtra(SymmetricService.INTENTKEY_EXTERNAL_ID,
"android-simulator");
intent.putExtra(SymmetricService.INTENTKEY_NODE_GROUP_ID, "client");
intent.putExtra(SymmetricService.INTENTKEY_START_IN_BACKGROUND,
true);

Properties properties = new Properties();
// initial load existing notes from the Client to the Server
properties.setProperty(ParameterConstants.AUTO_RELOAD_REVERSE_ENABLED,
"true");
intent.putExtra(SymmetricService.INTENTKEY_PROPERTIES, properties);
```

```
getContext().startService(intent);
```

This code snippet shows how the SQLiteOpenHelper is shared. The application's SQLiteOpenHelper is registered in a static registry provided by the SymmetricDS Android library. When the service is started, the key used to store the helper is passed to the service so that the service may pull the helper back out of the registry.

The various parameters needed by SymmetricDS are being set in the Intent which will be used by the SymmetricService to start the engine.

Most of the parameters will be familiar to SymmetricDS users. In this case a property is being set which will force an initial load of the existing Notes from the client to the server. This allows the user of the application to enter Notes for the first time offline or while the SymmetricDS engine is unregistered and still have them arrive at the centralized server once the SymmetricDS engine does get registered.

Next, set up an Android Emulator. This can be done by opening the Android Virtual Device Manager. Click New and follow the steps. The higher the Emulator's API, the better.

Run your NotePad project by pressing Run on NotePadProvider.java in Eclipse. When prompted, select the emulator you just created. Monitor the Console in Eclipse. Let the NotePad.apk install on the emulator. Now watch the LogCat and wait as it attempts to register with your SymmetricDS Master Node.

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.

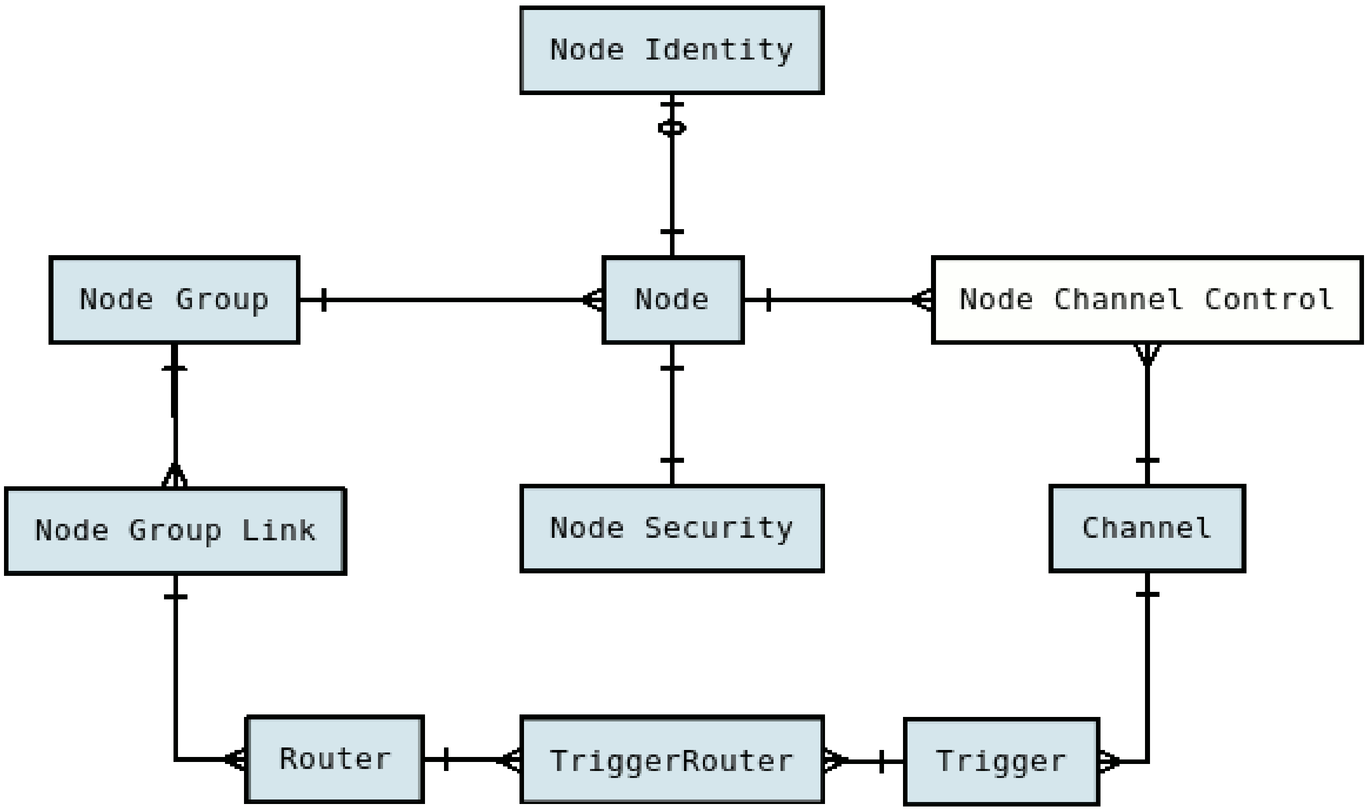


Figure A.1. Configuration Data Model

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.

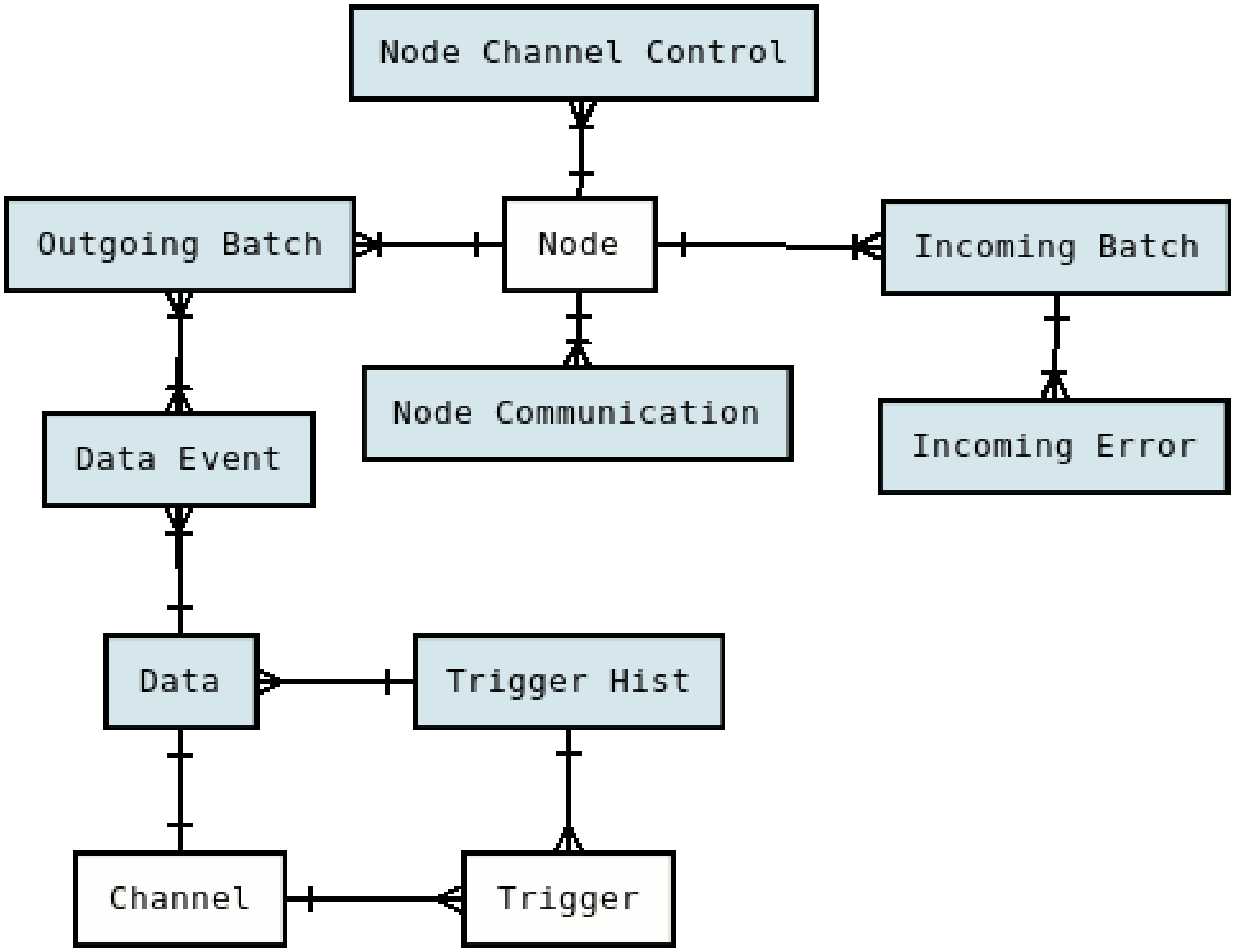


Figure A.2. Runtime Data Model

A.1. 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.1. CHANNEL

Name	Type / Size	Default	PK FK	not null	Description
CHANNEL_ID	VARCHAR (128)		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

Name	Type / Size	Default	PK FK	not null	Description
					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.
RELOAD_FLAG	INTEGER (1)	0		X	Indicates that this channel is used for reloads.
FILE_SYNC_FLAG	INTEGER (1)	0		X	Indicates that this channel is used for file sync.
CONTAINS_BIG_LOB	INTEGER (1)	0		X	Provides SymmetricDS a hint on how to treat captured data. Currently only supported by Oracle. If set to '0', then selects for routing and data extraction will be more efficient and lobs will be truncated at 4k in the trigger text. When it is set to '0' there is a 4k limit on the total size of a row and on the size of a LOB column. Note, when switching this value back and forth triggers need to be forced to regenerate.
BATCH_ALGORITHM	VARCHAR (50)	default		X	The algorithm to use when batching data on this channel. Possible values are: 'default', 'transactional', and 'nontransactional'
DATA_LOADER_TYPE	VARCHAR (50)	default		X	Identify the type of data loader this channel should use. Allows for the default dataloader to be swapped out via configuration for more efficient platform specific data loaders.
DESCRIPTION	VARCHAR (255)				Description on the type of data carried in this channel.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.2. CONFLICT

Defines how conflicts in row data should be handled during the load process.

Table A.2. CONFLICT

Name	Type / Size	Default	PK FK	not null	Description
CONFLICT_ID	VARCHAR (50)		PK	X	Unique identifier for a specific conflict detection setting.
source_node_group_id	VARCHAR (50)		FK	X	The source node group for which this setting will be applied to. References a node group link.
target_node_group_id	VARCHAR (50)		FK	X	The target node group for which this setting will be applied to. References a node group link.
TARGET_CHANNEL_ID	VARCHAR (128)				Optional channel that this setting will be applied to.
TARGET_CATALOG_NAME	VARCHAR (255)				Optional database catalog that the target table belongs to. Only use this if the target table is not in the default catalog.
TARGET_SCHEMA_NAME	VARCHAR (255)				Optional database schema that the target table belongs to. Only use this if the target table is not in the default schema.
TARGET_TABLE_NAME	VARCHAR (255)				Optional database table that this setting will apply to. If left blank, the setting will be for any table in the channel (if set) and in the specified node group link.
DETECT_TYPE	VARCHAR (128)			X	Indicates the strategy to use for detecting conflicts during a dml action. The possible values are: use_pk_data (manual, fallback, ignore), use_changed_data (manual, fallback, ignore), use_old_data (manual, fallback, ignore), use_timestamp (newer_wins), use_version (newer_wins)
DETECT_EXPRESSION	LONGVARCHAR				An expression that provides additional information about the detection mechanism. If the detection mechanism is use_timestamp or use_version then this expression will be the name of the timestamp or version column.
RESOLVE_TYPE	VARCHAR (128)			X	Indicates the strategy for resolving update conflicts. The possible values differ based on the detect_type that is specified.
PING_BACK	VARCHAR (128)			X	Indicates the strategy for sending resolved conflicts back to the source system. Possible values are: OFF, SINGLE_ROW, and REMAINING_ROWS.

Name	Type / Size	Default	PK FK	not null	Description
RESOLVE_CHANGES_ONLY	INTEGER (1)	0			Indicates that when applying changes during an update that only data that has changed should be applied. Otherwise, all the columns will be updated. This really only applies to updates.
RESOLVE_ROW_ONLY	INTEGER (1)	0			Indicates that an action should take place for the entire batch if possible. This applies to a resolve type of 'ignore'. If a row is in conflict and the resolve type is 'ignore', then the entire batch will be ignored.
CREATE_TIME	TIMESTAMP			X	The date and time when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	The date and time when a user last updated this entry.

A.3. DATA

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

Table A.3. DATA

Name	Type / Size	Default	PK FK	not null	Description
DATA_ID	BIGINT		PK	X	Unique identifier for a data.
TABLE_NAME	VARCHAR (255)			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.

Name	Type / Size	Default	PK FK	not null	Description
					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 (128)				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
NODE_LIST	VARCHAR (255)				A field that can be populated with a comma separated subset of node ids which will be the only nodes available to the router
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.4. DATA_EVENT

Each row represents the mapping between a data change that was captured and the batch that contains it. Entries in data_event are created as part of the routing process.

Table A.4. DATA_EVENT

Name	Type / Size	Default	PK FK	not null	Description
DATA_ID	BIGINT		PK	X	Id of the data to be routed.
BATCH_ID	BIGINT		PK	X	Id of the batch containing the data.
ROUTER_ID	VARCHAR (50)		PK	X	Id of the router that routed this data_event.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.5. 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.5. DATA_GAP

Name	Type / Size	Default	PK FK	not null	Description
START_ID	BIGINT		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	BIGINT		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.6. EXTRACT_REQUEST

This table is used internally to request the extract of initial loads asynchronously when the initial load extract job is enabled.

Table A.6. EXTRACT_REQUEST

Name	Type / Size	Default	PK FK	not null	Description
REQUEST_ID	BIGINT		PK	X	Unique identifier for a request.
NODE_ID	VARCHAR (50)			X	The node_id of the batch being loaded.
STATUS	CHAR (2)				NE, OK
START_BATCH_ID	BIGINT			X	A load can be split across multiple batches. This is the first of N batches the load will be split across.
END_BATCH_ID	BIGINT			X	This is the last of N batches the load will be split across.
TRIGGER_ID	VARCHAR (128)			X	Unique identifier for a trigger associated with the extract request.
ROUTER_ID	VARCHAR			X	Unique description of the router associated with

Name	Type / Size	Default	PK FK	not null	Description
	(50)				the extract request.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a process last updated this entry.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.

A.7. FILE_INCOMING

As files are loaded from another node the file and source node are captured here for file sync to use to prevent file ping backs in bidirectional file synchronization.

Table A.7. FILE_INCOMING

Name	Type / Size	Default	PK FK	not null	Description
RELATIVE_DIR	VARCHAR (255)		PK	X	The path to the file starting at the base_dir and excluding the file name itself.
FILE_NAME	VARCHAR (128)		PK	X	The name of the file that has been loaded.
LAST_EVENT_TYPE	CHAR (1)			X	The type of event that caused the file to be loaded from another node. 'C' is for create, 'M' is for modified, and 'D' is for deleted.
NODE_ID	VARCHAR (50)			X	The node_id of the source of the batch being loaded.
FILE_MODIFIED_TIME	BIGINT				The last modified time of the file at the time the file was loaded.

A.8. FILE_SNAPSHOT

Table used to capture file changes. Updates to the table are captured and routed according to the configured file trigger routers.

Table A.8. FILE_SNAPSHOT

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_ID	VARCHAR (128)		PK	X	The id of the trigger that caused this snapshot to be taken.
ROUTER_ID	VARCHAR (50)		PK	X	The id of the router that caused this snapshot to be taken.
RELATIVE_DIR	VARCHAR		PK	X	The path to the file starting at the base_dir

Name	Type / Size	Default	PK FK	not null	Description
	(255)				
FILE_NAME	VARCHAR (128)		PK	X	The name of the file that changed.
CHANNEL_ID	VARCHAR (128)	filesync		X	The channel_id of the channel that data changes will flow through.
RELOAD_CHANNEL_ID	VARCHAR (128)	filesync_reload		X	The channel_id of the channel that data changes will flow through.
LAST_EVENT_TYPE	CHAR (1)			X	The type of event captured by this entry. 'C' is for create, 'M' is for modified, and 'D' is for deleted.
CRC32_CHECKSUM	BIGINT				File checksum. Can be used to determine if file content has changed.
FILE_SIZE	BIGINT				The size in bytes of the file at the time this change was detected.
FILE_MODIFIED_TIME	BIGINT				The last modified time of the file at the time this change was detected.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
CREATE_TIME	TIMESTAMP			X	Timestamp when this entry was created.

A.9. FILE_TRIGGER

This table defines files or sets of files for which changes will be captured for file synchronization

Table A.9. FILE_TRIGGER

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_ID	VARCHAR (128)		PK	X	Unique identifier for a trigger.
CHANNEL_ID	VARCHAR (128)	filesync		X	The channel_id of the channel that data changes will flow through.
RELOAD_CHANNEL_ID	VARCHAR (128)	filesync_reload		X	The channel_id of the channel that will be used for reloads.
BASE_DIR	VARCHAR (255)			X	The base directory on the client that will be synchronized.
RECURSE	INTEGER (1)	1		X	Whether to synchronize child directories.
INCLUDES_FILES	VARCHAR (255)				Wildcard-enabled, comma-separated list of file to include in synchronization.

Name	Type / Size	Default	PK FK	not null	Description
EXCLUDES_FILES	VARCHAR (255)				Wildcard-enabled, comma-separated list of file to exclude from synchronization.
SYNC_ON_CREATE	INTEGER (1)	1		X	Whether to capture and send files when they are created.
SYNC_ON_MODIFIED	INTEGER (1)	1		X	Whether to capture and send files when they are modified.
SYNC_ON_DELETE	INTEGER (1)	1		X	Whether to capture and remove files when they are deleted.
SYNC_ON_CTL_FILE	INTEGER (1)	0		X	Combined with sync_on_create, determines whether to capture and send files when a matching control file exists. The control file is a file of the same name with a '.ctl' extension appended to the end.
DELETE_AFTER_SYNC	INTEGER (1)	0		X	Determines whether to delete the file after it has synced successfully.
BEFORE_COPY_SCRIPT	LONGVARCHAR				A bsh script that is run right before the file copy.
AFTER_COPY_SCRIPT	LONGVARCHAR				A bsh script that is run right after the file copy.
CREATE_TIME	TIMESTAMP			X	Timestamp of when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp of when a user last updated this entry.

A.10. FILE_TRIGGER_ROUTER

Maps a file trigger to a router.

Table A.10. FILE_TRIGGER_ROUTER

Name	Type / Size	Default	PK FK	not null	Description
trigger_id	VARCHAR (128)		PK FK	X	The id of a file trigger.
router_id	VARCHAR (50)		PK FK	X	The id of a router.
ENABLED	INTEGER (1)	1		X	Indicates whether this file trigger router is enabled or not.
INITIAL_LOAD_ENABLED	INTEGER (1)	1		X	Indicates whether this file trigger should be initial loaded.
TARGET_BASE_DIR	VARCHAR				The base directory on the destination that files

Name	Type / Size	Default	PK FK	not null	Description
	(255)				will be synchronized to.
CONFLICT_STRATEGY	VARCHAR (128)	source_wins		X	The strategy to employ when a file has been modified at both the client and the server. Possible values are: source_wins, target_wins, manual
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.11. GROUPLET

This tables defines named groups to which nodes can belong to based on their external id. Grouplets are used to designate that synchronization should only affect an explicit subset of nodes in a node group.

Table A.11. GROUPLET

Name	Type / Size	Default	PK FK	not null	Description
GROUPLET_ID	VARCHAR (50)		PK	X	Unique identifier for the grouplet.
GROUPLET_LINK_POLICY	CHAR (1)	I		X	Specified whether the external ids in the grouplet_link are included in the group or excluded from the grouplet. In the case of excluded, the grouplet starts with all external ids and removes the excluded ones listed. Use 'I' for inclusive and 'E' for exclusive.
DESCRIPTION	VARCHAR (255)				A description of this grouplet.
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.12. GROUPLET_LINK

This tables defines nodes belong to a grouplet based on their external.id

Table A.12. GROUPLET_LINK

Name	Type / Size	Default	PK FK	not null	Description
grouplet_id	VARCHAR (50)		PK FK	X	Unique identifier for the grouplet.
EXTERNAL_ID	VARCHAR (50)		PK	X	Provides a means to select the nodes that belong to a grouplet.
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.13. 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.13. INCOMING_BATCH

Name	Type / Size	Default	PK FK	not null	Description
BATCH_ID	BIGINT (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 (128)				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	This numbered data event that failed as read from the CSV.
FAILED_LINE_NUMBER	BIGINT	0		X	The current line number in the CSV for this batch that failed.
BYTE_COUNT	BIGINT	0		X	The number of bytes that were sent as part of

Name	Type / Size	Default	PK FK	not null	Description
					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.
IGNORE_COUNT	BIGINT	0		X	The number of times a row was ignored.
MISSING_DELETE_COUNT	BIGINT	0		X	The number of times a delete did not affect 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 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.14. INCOMING_ERROR

The captured data change that is in error for a batch. The user can tell the system what to do by updating the resolve columns. Entries in data_error are created when an incoming batch encounters an error.

Table A.14. INCOMING_ERROR

Name	Type / Size	Default	PK FK	not null	Description
BATCH_ID	BIGINT (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. A node_id of -1 means that the batch was 'unrouted'.

Name	Type / Size	Default	PK FK	not null	Description
FAILED_ROW_NUMBER	BIGINT		PK	X	The row number in the batch that encountered an error when loading.
FAILED_LINE_NUMBER	BIGINT	0		X	The current line number in the CSV for this batch that failed.
TARGET_CATALOG_NAME	VARCHAR (255)				The catalog name for the table being loaded.
TARGET_SCHEMA_NAME	VARCHAR (255)				The schema name for the table being loaded.
TARGET_TABLE_NAME	VARCHAR (255)			X	The table name for the table being loaded.
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.
BINARY_ENCODING	VARCHAR (10)	HEX		X	The type of encoding the source system used for encoding binary data.
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.
ROW_DATA	LONGVARCHAR				The row data from the batch as captured from the source. The column values are stored in comma-separated values (CSV) format.
OLD_DATA	LONGVARCHAR				The old row data prior to update from the batch as captured from the source. The column values are stored in CSV format.
CUR_DATA	LONGVARCHAR				The current row data that caused the error to occur. The column values are stored in CSV format.
RESOLVE_DATA	LONGVARCHAR				The capture data change from the user that is used instead of row_data. This is useful when resolving a conflict manually by specifying the data that should load.
RESOLVE_IGNORE	INTEGER (1)	0			Indication from the user that the row_data should be ignored and the batch can continue loading with the next row.
CONFLICT_ID	VARCHAR (50)				Unique identifier for the conflict detection setting that caused the error
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR				The user who last updated this entry.

Name	Type / Size	Default	PK FK	not null	Description
	(50)				
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.15. LOAD_FILTER

A table that allows you to dynamically define filters using bsh.

Table A.15. LOAD_FILTER

Name	Type / Size	Default	PK FK	not null	Description
LOAD_FILTER_ID	VARCHAR (50)		PK	X	The id of the load filter.
LOAD_FILTER_TYPE	VARCHAR (10)			X	The type of load filter. Currently 'bsh'. May add 'sql' in the future.
SOURCE_NODE_GROUP_ID	VARCHAR (50)			X	The source node group for the filter.
TARGET_NODE_GROUP_ID	VARCHAR (50)			X	The destination node group for the filter.
TARGET_CATALOG_NAME	VARCHAR (255)				Optional name for the catalog the configured table is in.
TARGET_SCHEMA_NAME	VARCHAR (255)				Optional name for the schema a configured table is in.
TARGET_TABLE_NAME	VARCHAR (255)				The name of the target table that will trigger the bsh filter.
FILTER_ON_UPDATE	INTEGER (1)	1		X	Whether or not the filter should apply on an update.
FILTER_ON_INSERT	INTEGER (1)	1		X	Whether or not the filter should apply on an insert.
FILTER_ON_DELETE	INTEGER (1)	1		X	Whether or not the filter should apply on a delete.
BEFORE_WRITE_SCRIPT	LONGVARCHAR				The script to apply before the write is completed.
AFTER_WRITE_SCRIPT	LONGVARCHAR				The script to apply after the write is completed.
BATCH_COMPLETE_SCRIPT	LONGVARCHAR				The script to apply on batch complete.
BATCH_COMMIT_SCRIPT	LONGVARCHAR				The script to apply on batch commit.
BATCH_ROLLBACK_SCRIPT	LONGVARCHAR				The script to apply on batch rollback.
HANDLE_ERROR_SCRIPT	LONGVARCHAR				The script to apply when data cannot be processed.

Name	Type / Size	Default	PK FK	not null	Description
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.
LOAD_FILTER_ORDER	INTEGER	1		X	Specifies the order in which to apply load filters if more than one target operation occurs.
FAIL_ON_ERROR	INTEGER (1)	0		X	Whether we should fail the batch if the filter fails.

A.16. 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.16. LOCK

Name	Type / Size	Default	PK FK	not null	Description
LOCK_ACTION	VARCHAR (50)		PK	X	The process that needs a lock.
LOCK_TYPE	VARCHAR (50)			X	Type of lock that indicates differently locking behavior. Types include cluster, exclusive, and shared. Cluster lock is used to allow one server to run at a time, but any process from the same server can overtake the lock, which avoids stalled processing. Exclusive lock is owned by one process, regardless of which server it is on, but another process can acquire the lock after lock_time is older than exclusive.lock.timeout.ms. Shared lock allows multiple processes to use the same lock, incrementing the shared_count, but requires no exclusive lock exists and prevents an exclusive 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.
SHARED_COUNT	INTEGER	0		X	For a lock_type of SHARED, this is the number of processes sharing the same lock. After the shared_count drops to zero, a shared lock is

Name	Type / Size	Default	PK FK	not null	Description
					removed.
SHARED_ENABLE	INTEGER	0		X	For a lock_type of SHARED, this flag set to 1 indicates that more processes can share the lock. If an exclusive lock is needed, the flag is set to 0 to prevent further shared locks from accumulating.
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.17. 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.17. 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 (50)				The version of SymmetricDS running at this 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.

Name	Type / Size	Default	PK FK	not null	Description
HEARTBEAT_TIME	TIMESTAMP				Deprecated. Use node_host.heartbeat_time instead.
TIMEZONE_OFFSET	VARCHAR (6)				Deprecated. Use node_host.timezone_offset instead.
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 if the heartbeat.update.node.with.batch.status property is set to true.
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 if the heartbeat.update.node.with.batch.status property is set to true.
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.18. NODE_COMMUNICATION

This table is used to coordinate communication with other nodes.

Table A.18. NODE_COMMUNICATION

Name	Type / Size	Default	PK FK	not null	Description
NODE_ID	VARCHAR (50)		PK	X	Unique identifier for a node.
COMMUNICATION_TYPE	VARCHAR (10)		PK	X	The type of communication that is taking place with this node. Valid values are: PULL, PUSH
LOCK_TIME	TIMESTAMP				The timestamp when this node was locked
LOCKING_SERVER_ID	VARCHAR (255)				The name of the server that currently has a pull lock for the node. This is typically a host name, but it can be overridden using the -Druntime.symmetric.cluster.server.id=name System property.
LAST_LOCK_TIME	TIMESTAMP				The timestamp when this node was last locked
LAST_LOCK_MILLIS	BIGINT	0			The amount of time the last communication took.

Name	Type / Size	Default	PK FK	not null	Description
SUCCESS_COUNT	BIGINT	0			The number of successive successful communication attempts.
FAIL_COUNT	BIGINT	0			The number of successive failed communication attempts.
TOTAL_SUCCESS_COUNT	BIGINT	0			The total number of successful communication attempts with the node.
TOTAL_FAIL_COUNT	BIGINT	0			The total number of failed communication attempts with the node.
TOTAL_SUCCESS_MILLIS	BIGINT	0			The total amount of time spent during successful communication attempts with the node.
TOTAL_FAIL_MILLIS	BIGINT	0			The total amount of time spent during failed communication attempts with the node.

A.19. 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.19. 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 (128)		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.20. 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.20. 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.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.21. NODE_GROUP_CHANNEL_WND

An optional window of time for which a node group and channel will extract and send data.

Table A.21. NODE_GROUP_CHANNEL_WND

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 (128)		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.22. NODE_GROUP_LINK

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

Table A.22. NODE_GROUP_LINK

Name	Type / Size	Default	PK FK	not null	Description
source_node_group_id	VARCHAR (50)		PK FK	X	The node group where data changes should be captured.
target_node_group_id	VARCHAR (50)		PK FK	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, R = Route-Only)
SYNC_CONFIG_ENABLED	INTEGER (1)	1		X	Indicates whether configuration that has changed should be synchronized to target nodes on this link.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.23. 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.23. 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.

Name	Type / Size	Default	PK FK	not null	Description
FREE_MEMORY_BYTES	BIGINT	0			The amount of free memory available to the JVM.
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.
JDBC_VERSION	VARCHAR (255)				The verision of the JDBC driver that is being used.
SYMMETRIC_VERSION	VARCHAR (50)				The version of SymmetricDS running at this node.
TIMEZONE_OFFSET	VARCHAR (6)				The time zone 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.24. NODE_HOST_CHANNEL_STATS

Table A.24. 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 (128)		PK	X	The channel_id of the channel that data changes will flow through.
START_TIME	TIMESTAMP		PK	X	The start time for the period which this row represents.

Name	Type / Size	Default	PK FK	not null	Description
END_TIME	TIMESTAMP		PK	X	The end time for the period which this row represents.
DATA_ROUTED	BIGINT	0			Indicate the number of data rows that have been routed during this period.
DATA_UNROUTED	BIGINT	0			The amount of data that has not yet been routed at the time this stats row was recorded.
DATA_EVENT_INSERTED	BIGINT	0			Indicate the number of data rows that have been routed during this period.
DATA_EXTRACTED	BIGINT	0			The number of data rows that were extracted during this time period.
DATA_BYTES_EXTRACTED	BIGINT	0			The number of bytes that were extracted during this time period.
DATA_EXTRACTED_ERRORS	BIGINT	0			The number of errors that occurred during extraction during this time period.
DATA_BYTES_SENT	BIGINT	0			The number of bytes that were sent during this time period.
DATA_SENT	BIGINT	0			The number of rows that were sent during this time period.
DATA_SENT_ERRORS	BIGINT	0			The number of errors that occurred while sending during this time period.
DATA_LOADED	BIGINT	0			The number of rows that were loaded during this time period.
DATA_BYTES_LOADED	BIGINT	0			The number of bytes that were loaded during this time period.
DATA_LOADED_ERRORS	BIGINT	0			The number of errors that occurred while loading during this time period.

A.25. NODE_HOST_JOB_STATS

Table A.25. 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

Name	Type / Size	Default	PK FK	not null	Description
JOB_NAME	VARCHAR (50)		PK	X	The name of the job.
START_TIME	TIMESTAMP		PK	X	The start time for the period which this row represents.
END_TIME	TIMESTAMP		PK	X	The end time for the period which this row represents.
PROCESSED_COUNT	BIGINT	0			The number of items that were processed during the job run.

A.26. NODE_HOST_STATS

Table A.26. 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	The end time for the period which this row represents.
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			
PURGED_DATA_ROWS	BIGINT	0			
PURGED_DATA_EVENT_ROWS	BIGINT	0			
PURGED_BATCH_OUTGOING_ROWS	BIGINT	0			

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

A.27. 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.27. NODE_IDENTITY

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

A.28. NODE_SECURITY

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

Table A.28. NODE_SECURITY

Name	Type / Size	Default	PK FK	not null	Description
node_id	VARCHAR (50)		PK FK	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.
INITIAL_LOAD_ENABLED	INTEGER (1)	0			Indicates whether an initial load will be sent to this node.
INITIAL_LOAD_TIME	TIMESTAMP				The timestamp when an initial load was started for this node.

Name	Type / Size	Default	PK FK	not null	Description
INITIAL_LOAD_ID	BIGINT				A reference to the load_id in outgoing_batch for the last load that occurred.
INITIAL_LOAD_CREATE_BY	VARCHAR (255)				The user that created the initial load. A null value means that the system created the batch.
REV_INITIAL_LOAD_ENABLED	INTEGER (1)	0			Indicates that this node should send a reverse initial load.
REV_INITIAL_LOAD_TIME	TIMESTAMP				The timestamp when this node last sent an initial load.
REV_INITIAL_LOAD_ID	BIGINT				A reference to the load_id in outgoing_batch for the last reverse load that occurred.
REV_INITIAL_LOAD_CREATE_BY	VARCHAR (255)				The user that created the reverse initial load. A null value means that the system created the batch.
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.

A.29. 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.29. OUTGOING_BATCH

Name	Type / Size	Default	PK FK	not null	Description
BATCH_ID	BIGINT		PK	X	A unique id for the batch.
NODE_ID	VARCHAR (50)		PK	X	The node that this batch is targeted at.
CHANNEL_ID	VARCHAR (128)				The channel that this batch is part of.
STATUS	CHAR (2)				The current status of a batch can be routing (RT), requested to be extracted in the background (RQ), newly created and ready for replication (NE), being queried from the database (QY), sent to a Node (SE), ready to be loaded (LD) and acknowledged as successful (OK), ignored (IG) or in error (ER).
LOAD_ID	BIGINT				An id that ties multiple batches together to

Name	Type / Size	Default	PK FK	not null	Description
					identify them as being part of an initial load.
EXTRACT_JOB_FLAG	INTEGER (1)	0			A flag that indicates that this batch is going to be extracted by another job.
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.
COMMON_FLAG	INTEGER (1)	0			A flag that indicates that the data in this batch is shared by other nodes (they will have the same batch_id). Shared batches will be extracted to a common location.
IGNORE_COUNT	BIGINT	0		X	The number of times a batch was ignored.
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 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.

Name	Type / Size	Default	PK FK	not null	Description
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.
FAILED_LINE_NUMBER	BIGINT	0		X	The current line number in the CSV for this batch that 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.
CREATE_BY	VARCHAR (255)				The user that created the batch. A null value means that the system created the batch.

A.30. PARAMETER

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

Table A.30. 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.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.31. 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.31. 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.32. REGISTRATION_REQUEST

Audits when a node registers or attempts to register.

Table A.32. REGISTRATION_REQUEST

Name	Type / Size	Default	PK FK	not null	Description
NODE_GROUP_ID	VARCHAR (50)		PK	X	The node group that this node belongs to, such as 'store'.
EXTERNAL_ID	VARCHAR (50)		PK	X	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 (50)			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.
ERROR_MESSAGE	LONGVARCHAR				Record any errors or warnings that occurred when attempting to register.

Name	Type / Size	Default	PK FK	not null	Description
CREATE_TIME	TIMESTAMP		PK	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.33. ROUTER

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

Table A.33. 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 (255)				Optional name for the catalog a target table is in. Only use this if the target table is not in the default catalog. If this field is left blank, then the source_catalog_name for the trigger will be used as the target name. If the target name should be left blank and the source name is set, then the token of $(none) may be used to force the target name to be blanked out.
TARGET_SCHEMA_NAME	VARCHAR (255)				Optional name of the schema a target table is in. On use this if the target table is not in the default schema. If this field is left blank, then the source_schema_name for the trigger will be used as the target name. If the target name should be left blank and the source name is set, then the token of $(none) may be used to force the target name to be blanked out.
TARGET_TABLE_NAME	VARCHAR (255)				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)		FK	X	Routers with this node_group_id will install triggers that are mapped to this router.
target_node_group_id	VARCHAR (50)		FK	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 (50)				The name of a specific type of router. Out of the box routers are 'default','column','bsh', 'subselect' and 'audit.' Custom routers can be configured as extension points.

Name	Type / Size	Default	PK FK	not null	Description
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.
USE_SOURCE_CATALOG_SCHEMA	INTEGER (1)	1		X	Whether or not to assume that the target catalog/schema name should be the same as the source catalog/schema name. The target catalog or schema name will still override if not blank.
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.34. SEQUENCE

A table that supports application level sequence numbering.

Table A.34. SEQUENCE

Name	Type / Size	Default	PK FK	not null	Description
SEQUENCE_NAME	VARCHAR (50)		PK	X	Unique identifier of a specific sequence.
CURRENT_VALUE	BIGINT	0		X	The current value of the sequence.
INCREMENT_BY	INTEGER	1		X	Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0.
MIN_VALUE	BIGINT	1		X	Specify the minimum value of the sequence.
MAX_VALUE	BIGINT	9999999999		X	Specify the maximum value the sequence can generate.
CYCLE	INTEGER (1)	0			Indicate whether the sequence should automatically cycle once a boundary is hit.
CREATE_TIME	TIMESTAMP				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.35. TABLE_RELOAD_REQUEST

This table acts as a means to queue up a reload of a specific table. Either the target or the source node may insert into this table to queue up a load. If the target node inserts into the table, then the row will be synchronized to the source node and the reload events will be queued up during routing.

Table A.35. TABLE_RELOAD_REQUEST

Name	Type / Size	Default	PK FK	not null	Description
TARGET_NODE_ID	VARCHAR (50)		PK	X	Unique identifier for the node to receive the table reload.
SOURCE_NODE_ID	VARCHAR (50)		PK	X	Unique identifier for the node that will be the source of the table reload.
TRIGGER_ID	VARCHAR (128)		PK	X	Unique identifier for a trigger associated with the table reload. Note the trigger must be linked to the router.
ROUTER_ID	VARCHAR (50)		PK	X	Unique description of the router associated with the table reload. Note the router must be linked to the trigger.
RELOAD_SELECT	LONGVARCHAR				Overrides the initial load select.
RELOAD_DELETE_STMT	LONGVARCHAR				Overrides the initial load delete statement.
RELOAD_ENABLED	INTEGER (1)	0			Indicates that a reload should be queued up.
RELOAD_TIME	TIMESTAMP				The timestamp when the reload was started for this node.
CREATE_TIME	TIMESTAMP				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.36. TRANSFORM_TABLE

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

Table A.36. 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 FK	X	The node group where data changes are captured.
target_node_group_id	VARCHAR (50)		PK FK	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 (255)				Optional name for the catalog the configured table is in.
SOURCE_SCHEMA_NAME	VARCHAR (255)				Optional name for the schema a configured table is in.
SOURCE_TABLE_NAME	VARCHAR (255)			X	The name of the source table that will be transformed.
TARGET_CATALOG_NAME	VARCHAR (255)				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 (255)				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 (255)				The name of the target table.
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. Possible values are: DEL_ROW, UPDATE_COL, or NONE.
TRANSFORM_ORDER	INTEGER	1		X	Specifies the order in which to apply transforms if more than one target operation occurs.
COLUMN_POLICY	VARCHAR (10)	SPECIFIED		X	Specifies whether all columns need to be specified or whether they are implied. Possible values are SPECIFIED or IMPLIED.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.37. TRANSFORM_COLUMN

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

Table A.37. 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 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.
CREATE_TIME	TIMESTAMP				Timestamp when this entry was created.
LAST_UPDATE_BY	VARCHAR (50)				The user who last updated this entry.
LAST_UPDATE_TIME	TIMESTAMP				Timestamp when a user last updated this entry.

A.38. 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.38. TRIGGER

Name	Type / Size	Default	PK FK	not null	Description
TRIGGER_ID	VARCHAR (128)		PK	X	Unique identifier for a trigger.
SOURCE_CATALOG_NAME	VARCHAR (255)				Optional name for the catalog the configured table is in. If the name includes * then a wildcard match on the table name will be attempted. Wildcard names can include a list of names that are comma separated. The ! symbol may be used to indicate a NOT match condition.
SOURCE_SCHEMA_NAME	VARCHAR (255)				Optional name for the schema a configured table is in. If the name includes * then a wildcard match on the table name will be attempted. Wildcard names can include a list of names that are comma separated. The ! symbol may be used to indicate a NOT match condition.
SOURCE_TABLE_NAME	VARCHAR (255)			X	The name of the source table that will have a trigger installed to watch for data changes. If the name includes * then a wildcard match on the table name will be attempted. Wildcard names can include a list of names that are comma separated. The ! symbol may be used to indicate a NOT match condition.
channel_id	VARCHAR (128)		FK	X	The channel_id of the channel that data changes will flow through.
reload_channel_id	VARCHAR (128)	reload	FK	X	The channel_id of the channel that will be used for reloads.
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 (255)				Override the default generated name for the update trigger.
NAME_FOR_INSERT_TRIGGER	VARCHAR (255)				Override the default generated name for the insert trigger.
NAME_FOR_DELETE_TRIGGER	VARCHAR (255)				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.

Name	Type / Size	Default	PK FK	not null	Description
CUSTOM_ON_UPDATE_TEXT	LONGVARCHAR				Specify update trigger text to execute after the SymmetricDS trigger text runs. This field is not applicable for H2, HSQLDB 1.x or Apachy Derby.
CUSTOM_ON_INSERT_TEXT	LONGVARCHAR				Specify insert trigger text to execute after the SymmetricDS trigger text runs. This field is not applicable for H2, HSQLDB 1.x or Apachy Derby.
CUSTOM_ON_DELETE_TEXT	LONGVARCHAR				Specify delete trigger text to execute after the SymmetricDS trigger text runs. This field is not applicable for H2, HSQLDB 1.x or Apachy Derby.
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.
CHANNEL_EXPRESSION	LONGVARCHAR				An expression that will be used to capture the channel id in the trigger. This expression will only be used if the channel_id is set to 'dynamic.'
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.
SYNC_KEY_NAMES	LONGVARCHAR				Specify a comma-delimited list of columns that should be used as the key for synchronization operations. By default, if not specified, then the primary key of the table will be used.
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. On Oracle, this may need to be set to 1 to get around 4k concatenation errors during data capture and during initial load.
USE_CAPTURE_OLD_DATA	INTEGER (1)	1		X	Indicates whether this trigger should capture and send the old data (previous state of the row before the change).

Name	Type / Size	Default	PK FK	not null	Description
USE_HANDLE_KEY_UPDATES	INTEGER (1)	0		X	Allows handling of primary key updates (SQLServer dialect only)
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.39. 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.39. 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 (128)			X	Unique identifier for a trigger
SOURCE_TABLE_NAME	VARCHAR (255)			X	The name of the source table that will have a trigger installed to watch for data changes.
SOURCE_CATALOG_NAME	VARCHAR (255)				The catalog name where the source table resides.
SOURCE_SCHEMA_NAME	VARCHAR (255)				The schema name where the source table resides.
NAME_FOR_UPDATE_TRIGGER	VARCHAR (255)				The name used when the insert trigger was created.
NAME_FOR_INSERT_TRIGGER	VARCHAR (255)				The name used when the update trigger was created.
NAME_FOR_DELETE_TRIGGER	VARCHAR (255)				The name used when the delete trigger was created.
TABLE_HASH	BIGINT	0		X	A hash of the table definition, used to detect changes in the definition.
TRIGGER_ROW_HASH	BIGINT	0		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.
TRIGGER_TEMPLATE_HASH	BIGINT	0		X	A hash of the trigger text. If changes are

Name	Type / Size	Default	PK FK	not null	Description
					detected to the values that affect a trigger text 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), Trigger template changed (E), Forced rebuild (F).
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.40. TRIGGER_ROUTER

Map a trigger to a router.

Table A.40. TRIGGER_ROUTER

Name	Type / Size	Default	PK FK	not null	Description
trigger_id	VARCHAR (128)		PK FK	X	The id of a trigger.
router_id	VARCHAR (50)		PK FK	X	The id of a router.
ENABLED	INTEGER (1)	1		X	Indicates whether this trigger router is enabled or not.
INITIAL_LOAD_ORDER	INTEGER	1		X	Order sequence of this table when an initial load is sent to a node. If this value is the same for multiple tables, then SymmetricDS will attempt to order the tables according to FK constraints. If this value is set to a negative number, then the table will be excluded from an initial load.
INITIAL_LOAD_SELECT	LONGVARCHAR				Optional expression that can be used to pare down the data selected from a table during the initial load process.

Name	Type / Size	Default	PK FK	not null	Description
INITIAL_LOAD_DELETE_STMT	LONGVARCHAR				The expression that is used to delete data when an initial load occurs. If this field is empty, no delete will occur before the initial load. If this field is not empty, the text will be used as a sql statement and executed for the initial load delete.
INITIAL_LOAD_BATCH_COUNT	INTEGER	1			Only applicable if the initial load extract job is enabled. The number of batches to split an initial load of a table across. If 0 then a select count(*) will be used to dynamically determine the number of batches based on the max_batch_size of the reload channel.
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.
LAST_UPDATE_TIME	TIMESTAMP			X	Timestamp when a user last updated this entry.

A.41. TRIGGER_ROUTER_GROUPLET

This tables defines what grouplets are associated with what trigger routers. The existence of the grouplet for a trigger_router enables nodes associated with the grouplet and at the same time it disables the trigger router for all other nodes.

Table A.41. TRIGGER_ROUTER_GROUPLET

Name	Type / Size	Default	PK FK	not null	Description
grouplet_id	VARCHAR (50)		PK FK	X	Unique identifier for the grouplet.
trigger_id	VARCHAR (128)		PK FK	X	The id of a trigger.
router_id	VARCHAR (50)		PK FK	X	The id of a router.
APPLIES_WHEN	CHAR (1)		PK	X	Indicates the side that a grouplet should be applied to. Use 'T' for target and 'S' for source and 'B' for both source and target.
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.

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-core jar file. This file has all the default settings along with descriptions.
conf/symmetric.properties	N	Changes to this file in the conf directory of a standalone install apply to all engines in the JVM.
symmetric-override.properties	N	Changes to this file, provided by the end user in the JVM's classpath, apply to all engines in the JVM.
engines/.properties*	N	Properties for a specific engine or node that is hosted in a standalone install.
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:

#auto.insert.registration.svr.if.not.found
If this is true, then node, group, security and identity rows will be inserted if the registration.url is blank and there is no configured node identity. [Default: false]

#db.driver
null [Default: com.sybase.jdbc4.jdbc.SybDriver]

#db.url
null [Default: jdbc:sybase:Tds:localhost:5000/client]

auto.config.database
If this is true, when symmetric starts up it will try to create the necessary tables. [Default: true]

auto.config.registration.svr.sql.script
Provide the path to a SQL script that can be run to do initial setup of a registration server. This script will only be run on a registration server if the node_identity cannot be found. [Default:]

auto.sync.configuration
Capture and send SymmetricDS configuration changes to client nodes. [Default: true]

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

cache.table.time.ms
This is the amount of time table meta data will be cached before re-reading it from the database [Default: 3600000]

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:]

db.connection.properties
These are settings that will be passed to the JDBC driver as connection properties. Suggested settings by database are as follows: Oracle
db.connection.properties=oracle.net.CONNECT_TIMEOUT=300000;oracle.net.READ_TIMEOUT=300000;Set
[Default:]

db.delimited.identifier.mode
Determines whether delimited identifiers are used or normal SQL92 identifiers (which may only contain alphanumerical characters and the underscore, must start with a letter and cannot be a reserved keyword). [Default: true]

db.driver
Specify your database driver [Default: org.h2.Driver]

db.init.sql
Specify a SQL statement that will be run when a database connection is created [Default:]

db.jdbc.execute.batch.size
This is the default number of rows that will be sent to the database as a batch when SymmetricDS uses the JDBC batch API. Currently, only routing uses JDBC batch. The data loader does not. [Default: 100]

db.jdbc.streaming.results.fetch.size
This is the default fetch size for streaming result sets. [Default: 100]

db.jndi.name
Name of a JNDI data source to use instead of using SymmetricDS's connection pool. When this is set

the db.url is ignored. Using a JNDI data source is relevant when deploying to an application server. [Default:]

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

db.native.extractor
Name of class that can extract native JDBC objects and interact directly with the driver. Spring uses this to perform operations specific to database, like handling LOBs on Oracle. [Default: org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor]

db.password
Specify your database password [Default:]

db.pool.initial.size
The initial size of the connection pool [Default: 5]

db.pool.max.active
The maximum number of connections that will be allocated in the pool The http.concurrent.workers.max value should be half or less than half of this value. [Default: 40]

db.pool.max.idle
The maximum number of connections that can remain idle in the pool, without extra ones being released [Default: 20]

db.pool.max.wait.millis
This is how long a request for a connection from the datasource will wait before giving up. [Default: 30000]

db.pool.min.evictable.idle.millis
This is how long a connection can be idle before it will be evicted. [Default: 120000]

db.pool.min.idle
The minimum number of connections that can remain idle in the pool, without extra ones being created [Default: 5]

db.read.strings.as.bytes
If set to true forces database columns that contain character data to be read as bytes (bypassing JDBC driver character encoding) so the raw values be encoded using the system default character set (usually UTF8). This property was added to bypass MySQL character encoding so the raw data can be converted to utf8 directly. [Default: false]

db.sql.query.timeout.seconds
Most symmetric queries have a timeout associated with them. This is the default. [Default: 300]

db.url
Specify your database URL [Default: jdbc:h2:mem:setme;AUTO_SERVER=TRUE]

db.user
Specify your database user [Default: please set me]

db.validation.query
This is the query to validate the database connection in Connection Pool. It is database specific. The following are example statements for different databases. MySQL db.validation.query=select 1 Oracle db.validation.query=select 1 from dual DB2 db.validation.query=select max(1) from syscat.datatypes [Default:]

db2.zseries.version
Use to map the version string a zseries jdbc driver returns to the 'zseries' dialect [Default: DSN08015]

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. Please do not use underscores in this name. [Default: SymmetricDS]

external.id
The external id for this SymmetricDS node. The external id is usually used as all or part of the node id. [Default: please set me]

file.sync.lock.wait.ms
How long file sync should wait in millis for the exclusive lock used by file tracker or the shared lock used by file sync push/pull jobs. [Default: 300000]

group.id
The node group id that this node belongs to [Default: please set me]

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]

http.concurrent.reservation.timeout.ms
This is the amount of time the host will keep a concurrent connection reservation after it has been attained by a client node while waiting for the subsequent reconnect to push. [Default: 20000]

https.verified.server.names
During SSL handshaking, if the URL's hostname and the server's identification hostname mismatch, the verification mechanism will check this comma separated list of server names to see if the cert should be accepted (see javax.net.ssl.HostnameVerifier.) Set this value equal to 'all' if all server names should be accepted. Set this value to blank if a valid SSL cert is required. [Default:]

jmx.line.feed
Specify the type of line feed to use in JMX console methods. Possible values are: text or html. [Default: text]

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]

mysql.bulk.load.local
Whether or not files are local to client only, so we must send the file to MySQL to load. If client is

running on same server as MySQL, then this can be set to false to have MySQL read file directly. [Default: true]

mysql.bulk.load.max.bytes.before.flush
Maximum number of bytes to write to file before running with 'LOAD DATA INFILE' to MySQL [Default: 1000000000]

mysql.bulk.load.max.rows.before.flush
Maximum number of rows to write to file before running with 'LOAD DATA INFILE' to MySQL [Default: 100000]

oracle.template.precision
This is the precision that is used in the number template for oracle triggers [Default: 30,10]

oracle.transaction.view.clock.sync.threshold.ms
Requires access to gv$transaction. This is the threshold by which clock can be off in an oracle rac environment. It is only applicable when oracle.use.transaction.view is set to true. [Default: 60000]

oracle.use.transaction.view
Requires access to gv$transaction [Default: false]

registration.url
This is the URL this node will use to register and pull it's configuration. If this is the root server, then this may remain blank and the configuration should be inserted directly into the database [Default: please set me]

security.service.class.name
The class name for the Security Service to use for encrypting and decrypting database passwords [Default: org.jumpmind.security.SecurityService]

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_host table for the current node. [Default: true]

start.initial.load.extract.job
Whether the background initial load extractor job is started. [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.push.job
Whether the push job is enabled for this node. [Default: true]

start.refresh.cache.job
Whether the refresh cache job is enabled for this node. [Default: false]

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

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

start.stat.flush.job
Whether the statistic flush job is enabled for this node. [Default: true]

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

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

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

sync.url
The url that can be used to access this SymmetricDS node. The default setting of http://$(hostName):31415/sync should be valid of the standalone launcher is used with the default settings The tokens of $(hostName) and $(ipAddress) are supported for this property. [Default: http://$(hostName):31415/sync/$(engineName)]

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

web.batch.servlet.enable
Indicate whether the batch servlet (which allows specific batches to be requested) is enabled. [Default: true]

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.reload.reverse
If this is true, a reload is automatically sent from a source node to all target nodes after the source node has registered. [Default: false]

auto.sync.configuration.on.incoming
Whether triggers should fire when changes sync into the node that this property is configured for. [Default: true]

auto.sync.triggers
If this is true, when symmetric starts up it will make sure the triggers in the database are up to date. [Default: true]

auto.sync.triggers.after.config.change
If this is true, when a configuration change is detected, symmetric will make sure all triggers in the database are up to date. [Default: true]

auto.sync.triggers.at.startup
If this is true, then run the sync triggers process at startup [Default: true]

bsh.load.filter.handles.missing.tables
This parameter can be used to indicate that bean shell load filters will handle missing tables. Useful for the case where you want to make, for example, global catalog or schema changes at the destination in the case where the catalog, schema, or table doesn't exist but the BSH will handle it. [Default: false]

bsh.transform.global.script
BeanShell script to include at the beginning of all scripts used in transforms [Default:]

cache.channel.time.ms
This is the amount of time channel entries will be cached before re-reading them from the database. [Default: 60000]

cache.conflict.time.ms
This is the amount of time conflict setting entries will be cached before re-reading them from the database. [Default: 600000]

cache.grouplets.time.ms
This is the amount of time grouplet entries will be cached before re-reading them from the database. [Default: 600000]

cache.load.filter.time.ms
This is the amount of time load filter entries will be cached before re-reading them from the database. [Default: 600000]

cache.node.group.link.time.ms
This is the amount of time node group links entries will be cached before re-reading them from the database. [Default: 600000]

cache.node.security.time.ms
This is the amount of time node security entries will be cached before re-reading them from the database. [Default: 0]

cache.transform.time.ms
This is the amount of time transform entries will be cached before re-reading them from the database. [Default: 600000]

cache.trigger.router.time.ms
This is the amount of time trigger entries will be cached before re-reading them from the database. [

Default: 600000]

cluster.lock.enabled
Enables clustering of jobs. [Default: false]

cluster.lock.timeout.ms
Indicate that this node is being run on a farm or cluster of servers and it needs to use the database to 'lock' out other activity when actions are taken. [Default: 1800000]

compression.level
Set the compression level this node will use when compressing synchronization payloads. @see java.util.zip.Deflater 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. @see java.util.zip.Deflater FILTERED = 1 HUFFMAN_ONLY = 2 DEFAULT_STRATEGY = 0 [Default: 0]

data.id.increment.by
This is the expected increment value for the data_id in the data table. This is useful if you use auto_increment_increment and auto_increment_offset in MySQL. Note that these settings require innodb_autoinc_lock_mode=0, otherwise the increment and offset are not guaranteed. [Default: 1]

dataextractor.enable
Disable the extraction of all channels with the exception of the config channel [Default: true]

dataloader.create.table.alter.to.match.db.case
Whether to alter the case of the database tables that are created by the SymmetricDS data loader to match the default case of the target database. [Default: true]

dataloader.enable
Disable the loading of all channel with the exception of the config channel. This property can be set to allow all changes to be extracted without introducing other changes in order to allow maintenance operations. [Default: true]

dataloader.error.save.curval
Indicates that the current value of the row should be recorded in the incoming_error table [Default: false]

dataloader.fit.to.column
Indicate that the data loader should truncate data that is bigger than the target columns can handle. This applies to text-based columns only. [Default: false]

dataloader.ignore.missing.tables
Tables that are missing at the target database will be ignored. This should be set to true if you expect that in some clients a table might not exist. If set to false, the batch will fail. [Default: false]

dataloader.max.rows.before.commit
This is the maximum number of rows that will be supported in a single transaction. If the database transaction row count reaches a size that is greater than this number then the transaction will be auto

committed. The default value of -1 indicates that there is no size limit. [Default: 10000]

dataloader.sleep.time.after.early.commit
Amount of time to sleep before continuing data load after dataloader.max.rows.before.commit rows have been loaded. This is useful to give other application threads a chance to do work before continuing to load. [Default: 5]

datareload.batch.insert.transactional
Indicate whether the process of inserting data, data_events and outgoing_batches for a reload is transactional. The only reason this might be marked as false is to reduce possible contention while multiple nodes connect for reloads at the same time. [Default: true]

db.treat.date.time.as.varchar.enabled
This is a setting that instructs the data capture and data load to treat JDBC TIME, DATE, and TIMESTAMP columns as if they were VARCHAR columns. This means that the columns will be captured and loaded in the form that the database stores them. Setting this to true on MySQL will allow datetime columns with the value of '0000-00-00 00:00:00' to be synchronized. [Default: false]

extensions.xml
Spring xml configuration for extension points. This property enables maintaining Spring extension point configuration in the database. After changing this property a server restart is required. [Default:
<?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"
xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd"> </beans>]

external.id.is.unique.enabled
null [Default: true]

file.pull.lock.timeout.ms
null [Default: 7200000]

file.pull.period.minimum.ms
null [Default: 0]

file.pull.thread.per.server.count
null [Default: 1]

file.push.lock.timeout.ms
null [Default: 7200000]

file.push.period.minimum.ms
null [Default: 0]

file.push.thread.per.server.count
null [Default: 1]

file.sync.enable
Enables File Synchronization capabilities [Default: true]

heartbeat.sync.on.push.enabled
Specify whether to push node_host records to configured push clients. If this is true the node for this instance and the node_host rows for all children instances will be pushed to all nodes that this node is configured to push to. [Default: true]

heartbeat.sync.on.push.period.sec
This is the number of seconds between when the sym_node_host table's heartbeat_time column is updated. This property depends on the frequency of the heartbeat job. If the heartbeat job is set to run every 10 minutes and this property is set to 10 seconds, then the heartbeat will only update every 10 minutes. [Default: 0]

heartbeat.sync.on.startup
When this property is set to true the heartbeat process will run at server startup. Prior to 3.4 the heartbeat always happened at startup. [Default: false]

heartbeat.update.node.with.batch.status
When this is set to true, SymmetricDS will update fields in the sym_node table that indicate the number of outstanding errors and/or batches it has pending [Default: false]

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. @see web.compression.disabled to enable/disable the filter [Default: true]

http.concurrent.workers.max
This is the number of HTTP concurrent push/pull requests SymmetricDS will accept. This is controlled by the NodeConcurrencyFilter. The number is per servlet the filter is applied to. The db.pool.max.active value should be twice this value. [Default: 20]

http.push.stream.output.enabled
The HTTP client connection, during a push, buffers the entire outgoing pay-load locally before sending it. Set this to true if you are getting heap space errors during a push. Note that basic auth may not work when this is turned on. [Default: false]

http.push.stream.output.size
When HTTP chunking is turned on, this is the size to use for each chunk. [Default: 30720]

http.timeout.ms
Sets both the connection and read timeout on the internal HttpUrlConnection [Default: 7200000]

incoming.batches.record.ok.enabled
Indicates whether batches that have loaded successfully should be recorded in the incoming_batch table. Note that if this is set to false, then duplicate batches will NOT be skipped because SymmetricDS will have no way of knowing that a batch has already loaded. This parameter can be set

to false to reduce contention on sym_incoming_batch for systems with many clients. [Default: true]

incoming.batches.skip.duplicates
This instructs symmetric to attempt to skip duplicate batches that are received. Symmetric might be more efficient when recovering from error conditions if this is set to true, but you run the risk of missing data if the batch ids get reset (on one node, but not another) somehow (which is unlikely in production, but fairly likely in lab or development setups). [Default: true]

initial.load.after.sql
This is SQL that will run on the client after an initial load finishes. [Default:]

initial.load.before.sql
This is SQL that will run on the client before an initial load starts. [Default:]

initial.load.concat.csv.in.sql.enabled
Indicates that the SQL used to extract data from a table for an initial load should concatenate the data using the same SQL expression that a trigger uses versus concatenating the data in code. [Default: false]

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

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

initial.load.delete.first.sql
This is the SQL statement that will be used for purging a table during an initial load. [Default: delete from %s]

initial.load.extract.thread.per.server.count
The number of threads available for concurrent extracts of initial load batches. [Default: 20]

initial.load.extract.timeout.ms
The number of milliseconds to wait until the lock will be broken on an initial load extract job. [Default: 7200000]

initial.load.reverse.after.sql
This is SQL that will run on the server after a reverse initial load finishes. [Default:]

initial.load.reverse.before.sql
This is SQL that will run on the server before a reverse initial load starts. [Default:]

initial.load.reverse.first
Indicate that if both the initial load and the reverse initial load are requested, then the reverse initial load should take place first. [Default: true]

initial.load.schema.dump.command
Specify a system command that writes the structure of the database to system.out to be captured and sent to the node that is being initial loaded. Used in conjunction with initial.load.schema.load.command. An example is: pg_dump --dbname=server --schema=my_schema --schema-only --clean [Default:]

initial.load.schema.load.command
Specify a system command that will take the content captured by initial.load.schema.dump.command and apply it to the database. The content is passed to the system command via system.in. An example is: psql --output=output.log --dbname=client [Default:]

initial.load.use.extract.job.enabled
Indicate that the extract job job should be used to extract reload batches [Default: false]

initial.load.use.reload.channel
Indicate that the initial load events should be put on the reload channel. If this is set to false each table will be put on it's assigned channel during the reload. [Default: true]

job.file.sync.pull.period.time.ms
null [Default: 60000]

job.file.sync.push.period.time.ms
null [Default: 60000]

job.file.sync.tracker.cron
null [Default: 0 0/5 * * * *]

job.heartbeat.period.time.ms
This is how often the heartbeat job runs. Note that this doesn't mean that a heartbeat is performed this often. See heartbeat.sync.on.push.period.sec to change how often the heartbeat is sync'd [Default: 900000]

job.initial.load.extract.period.time.ms
This is how often the initial load extract queue job will run in the background [Default: 10000]

job.pull.period.time.ms
This is how often the pull job will be run to schedule pulls of nodes. [Default: 60000]

job.purge.datagaps.cron
This is how often the data gaps purge job will be run. [Default: 0 0 0 * * *]

job.purge.incoming.cron
This is how often the incoming batch purge job will be run. [Default: 0 0 0 * * *]

job.purge.max.num.batches.to.delete.in.tx
This is the number of batches that will be purged in one database transaction. [Default: 5000]

job.purge.max.num.data.event.batches.to.delete.in.tx
This is the number of batches that will be purged from the data_event table in one database transaction. [Default: 5]

job.purge.max.num.data.to.delete.in.tx
This is the number of data ids that will be purged in one database transaction. [Default: 5000]

job.purge.outgoing.cron
This is how often the outgoing batch and data purge job will be run. [Default: 0 0 0 * * *]

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

job.refresh.cache.cron
This is when the refresh cache job will run. [Default: 0/30 * * * * *]

job.routing.period.time.ms
This is how often the router will run in the background [Default: 10000]

job.stage.management.period.time.ms
This is when the stage management job will run. [Default: 15000]

job.stat.flush.cron
This is how often accumulated statistics will be flushed out to the database from memory. [Default: 0 0/5 * * * *]

job.synctriggers.cron
This is when the sync triggers job will run. [Default: 0 0 0 * * *]

job.watchdog.period.time.ms
null [Default: 3600000]

jobs.synchronized.enable
If jobs need to be synchronized so that only one job can run at a time, set this parameter to true [Default: false]

lock.timeout.ms
The amount of time a thread can hold a shared or exclusive lock before another thread can break the lock. The timeout is a safeguard in case an unexpected exception causes a lock to be abandoned. Restarting the service will clear all locks. [Default: 1800000]

lock.wait.retry.ms
While waiting for a lock to be released, how often should we check the lock status in the sym_lock table in the database. [Default: 10000]

mssql.allow.only.row.level.locks.on.runtime.tables
Automatically update data, data_event and outgoing_batch tables to allow only row level locking. [Default: true]

mssql.bulk.load.fire.triggers
Whether or not triggers should be allowed to fire when bulk loading data. [Default: false]

mssql.bulk.load.max.rows.before.flush
Maximum number of rows to write to file before running with "BULK INSERT" to SQL-Server [Default: 100000]

mssql.bulk.load.unc.path
Specify a UNC network path to the tmp\bulkloaddir directory for SQL Server to access bulk load files. Use this property with bulk loader when SymmetricDS is on a separate server from SQL Server. [Default:]

mssql.trigger.execute.as
Specify the user the SymmetricDS triggers should execute as. Possible values are { CALLER | SELF | OWNER | 'user_name' } [Default: caller]

mssql.use.ntypes.for.sync
Use ntext for the data capture columns and cast to nvarchar(max) in the trigger text so that nvarchar, ntext and nchar double byte data isn't lost when the database collation for char types isn't compatible with n char types. [Default: false]

mysql.bulk.load.replace
Whether or not to replace rows that already exist, based on primary key or unique key. If set to false, duplicates will be skipped. [Default: true]

node.copy.mode.enabled
If the copy mode is enabled and the node starts up with an identity that does not match the configured external id, then the node will register with a special parameter that indicates the registration server should copy outgoing batch to the new node id. [Default: false]

node.id.creator.script
This is a bean shell script that will be used to generate the node id for a registering node [Default:]

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]

offline.node.detection.period.minutes
This is the number of minutes that a node has been offline before taking action A value of -1 (or any negative value) disables the feature. [Default: -1]

outgoing.batches.max.to.select
The maximum number of unprocessed outgoing batch rows for a node that will be read into memory for the next data extraction. [Default: 50000]

outgoing.batches.peek.ahead.batch.commit.size
This is the number of data events that will be batched and committed together while building a batch. Note that this only kicks in if the prospective batch size is bigger than the configured max batch size. [Default: 10]

parameter.reload.timeout.ms
The number of milliseconds parameters will be cached by the ParameterService before they are reread from the file system and database. [Default: 600000]

pull.lock.timeout.ms
The amount of time a single pull worker node_communication lock will timeout after. [Default: 7200000]

pull.period.minimum.ms
This is the minimum time that is allowed between pulls of a specific node. [Default: 0]

pull.thread.per.server.count

The number of threads created that will be used to pull nodes concurrently on one server in the cluster. [Default: 1]

purge.extract.request.retention.minutes
This is the retention time for how long a extract request will be retained [Default: 7200]

purge.log.summary.retention.minutes
This is the retention for how long log summary messages will be retained in memory. [Default: 60]

purge.registration.request.retention.minutes
This is the retention time for how long a registration request will be retained [Default: 7200]

purge.retention.minutes
This is the retention for how long synchronization data will be kept in the symmetric synchronization tables. Note that data will be purged only if the purge job is enabled. [Default: 1440]

purge.stats.retention.minutes
This is the retention for how long statistic data will be kept in the symmetric stats tables. Note that data will be purged only if the statistics flush job is enabled. [Default: 1440]

push.lock.timeout.ms
The amount of time a single push worker node_communication lock will timeout after. [Default: 7200000]

push.period.minimum.ms
This is the minimum time that is allowed between pushes to a specific node. [Default: 0]

push.thread.per.server.count
The number of threads created that will be used to push to nodes concurrently on one server in the cluster. [Default: 1]

redshift.bulk.load.max.bytes.before.flush
Maximum number of bytes to write to file before copying to S3 and running with COPY statement [Default: 1000000000]

redshift.bulk.load.max.rows.before.flush
Maximum number of rows to write to file before copying to S3 and running with COPY statement [Default: 100000]

redshift.bulk.load.s3.access.key
The AWS access key ID (aws_access_key_id) to use as credentials for uploading to S3 [Default:]

redshift.bulk.load.s3.bucket
The S3 bucket where bulk load files will be uploaded to before bulk loading into Redshift [Default:]

redshift.bulk.load.s3.secret.key
The AWS secret access key (aws_secret_access_key) to use as credentials for uploading to S3 [Default:]

registration.number.of.attempts
This is the number of times registration will be attempted before being aborted. The default value is -1

which means an endless number of attempts. This parameter is specific to the node that is trying to register, not the node that is providing registration. [Default: -1]

registration.reinitialize.enable
Indicates whether SymmetricDS should be re-initialized immediately before registration. [Default: false]

registration.reopen.use.same.password
Indicates that if registration is reopened if the same password should be used. If set to false then a new password will be generated. [Default: true]

rest.api.enable
Enables the REST API [Default: false]

rest.api.heartbeat.on.pull
Enables the REST API to update the heartbeat when pulling data [Default: false]

routing.data.reader.order.by.gap.id.enabled
Use the order by clause to order sym_data when selecting data for routing. Most databases order the data naturally and might even have better performance when the order by clause is left off. [Default: true]

routing.data.reader.threshold.gaps.to.use.greater.than.query
Select data to route from sym_data using a simple > start_gap_id query if the number of gaps in sym_data_gap are greater than the following number [Default: 100]

routing.data.reader.type.gap.retention.period.minutes
null [Default: 1440]

routing.delete.filled.in.gaps.immediately
When true, delete the gaps instead of marking them as OK or SK. [Default: true]

routing.flush.jdbc.batch.size
null [Default: 50000]

routing.largest.gap.size
This is the maximum number of data that will be routed during one run. It should be a number that well exceeds the number rows that will be in a transaction. [Default: 50000000]

routing.log.stats.on.batch.error
Enable to collect routing statistics for each batch and log the statistics when a batch goes into error. [Default: false]

routing.max.gaps.to.qualify.in.sql
This is the number of gaps that will be included in the SQL that is used to select data from sym_data. If there are more gaps than this number, then the last gap will in the SQL will use the end id of the last gap. [Default: 100]

routing.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: 2000]

routing.stale.dataid.gap.time.ms
This is the time that any gaps in data_ids will be considered stale and skipped. [Default: 7200000]

routing.wait.for.data.timeout.seconds
null [Default: 330]

schema.version
This is hook to give the user a mechanism to indicate the schema version that is being synchronized. [Default: ?]

server.log.file
The name of the active log file. This is used by the system to locate the log file for analysis and trouble shooting. It is set to the default log file location for the standalone server. If deployed as a war file, you should update the value. Note that this property does not change the actual location the log file will be written. It just tells SymmetricDS where to find the log file. [Default: ../logs/symmetric.log]

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]

stream.to.file.ttl.ms
If stream.to.file.enabled is true, then this is how long a file will be retained in the staging directory after it has been marked as done. [Default: 3600000]

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]

transport.max.bytes.to.sync
This is the number of maximum number of bytes to synchronize in one connect. [Default: 1048576]

trigger.create.before.initial.load.enabled
Disable this property to prevent table triggers from being created before initial load has completed. [Default: true]

trigger.update.capture.changed.data.only.enabled
Enable this property to force a compare of old and new data in triggers. If old=new, then don't record the change in the data capture table. This is currently supported by the following dialects: mysql, oracle, db2, postgres, sql server [Default: false]

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]

B.3. Server Configuration

Server configuration is read from `conf/symmetric-server.conf` for settings needed by the server before the parameter system has been initialized.

#host.bind.name
null [Default:]

http.enable
Enable synchronization over HTTP. [Default: true]

http.port
Port number for synchronization over HTTP. [Default: 31415]

https.allow.self.signed.certs
Use a trust manager that allows self-signed server SSL certificates. [Default: true]

https.enable
Enable synchronization over HTTPS (HTTP over SSL). [Default: false]

https.port
Port number for synchronization over HTTPS (HTTP over SSL). [Default: 31417]

https.verified.server.names
List host names that are allowed for server SSL certificates. [Default: all]

jmx.agent.enable
Enable Java Management Extensions (JMX) remote agent. [Default: true]

jmx.agent.port
Port number for the Java Management Extensions (JMX) remote agent. [Default: 31418]

jmx.http.enable
Enable Java Management Extensions (JMX) web console. [Default: true]

jmx.http.port
Port number for Java Management Extensions (JMX) web console. [Default: 31416]

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	Data Capture	Conditional Sync	Update Loop Prevention	BLOB Sync
Oracle	10g and above	Y	Y	Y	Y	Y
MySQL	5.0.2 and above	Y	Y	Y	Y	Y
MariaDB	5.1 and above	Y	Y	Y	Y	Y
PostgreSQL	8.2.5 and above	Y (8.3 and above only)	Y	Y	Y	Y
Greenplum	8.2.15 and above	N	N	N	Y	N
SQL Server	2005 and above	Y	Y	Y	Y	Y
SQL Server Azure	Tested on 11.00.2065	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
SQLite	3.x	N	Y	Y	Y	Y
Active Server Enterprise	12.5	Y	Y	Y	Y	Y
SQL Anywhere	9	Y	Y	Y	Y	Y
Redshift	1.0	N	N	N	Y	N

C.1. Oracle

SymmetricDS has bulk loading capability available for Oracle. SymmetricDS specifies data loader types on a channel by channel basis. To utilize Oracle Bulk loading versus straight JDBC insert, specify the Oracle Bulk Loader ("oracle_bulk") in the data_loader_type column of sym_channel.

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
);
```

```

```

Note also that, for Oracle, you can control the amount of precision used by the Oracle triggers with the parameter `oracle.template.precision`, which defaults to a precision of 30,10.

If the following Oracle error 'ORA-01489: result of string concatenation is too long' is encountered you might need to set `use_capture_lobs` to 1 on in the TRIGGER table and resync the triggers. The error can happen when the captured data in a row exceeds 4k and lob columns do not exist in the table. By enabling `use_capture_lobs` the concatanated varchar string is cast to a clob which allows a length of more than 4k.

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 cannot 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`.

If you are using UTF-8 encoding in the database, you might consider using the `characterEncoding` parameter in the JDBC URL.

```
jdbc:mysql://hostname/databasename?tinyInt1isBit=false&characterEncoding=utf8
```

C.3. MariaDB

See MySQL notes. In addition, you will need to use a MySQL driver for this dialect.

C.4. PostgreSQL

SymmetricDS has bulk loading capability available for Postgres. SymmetricDS specifies data loader types on a channel by channel basis. To utilize Postgres Bulk loading versus straight JDBC insert, specify the Postgres Bulk Loader ("postgres_bulk") in the data_loader_type column of sym_channel.

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;
```

If you want SymmetricDS to install into a schema other than public you should alter the database user to set the default schema.

```
        alter user {user name} set search_path to {schema name};
```

In addition, you will likely need the follow privelegdes as well:

```
        GRANT USAGE ON SCHEMA {schema name} TO {user name};
        GRANT CREATE ON SCHEMA {schema name} TO {user name};
```

C.5. Greenplum

Greenplum is a data warehouse based on PostgreSQL. It is supported as a target platform in SymmetricDS.

SymmetricDS has bulk loading capability available for Greenplum. SymmetricDS specifies data loader types on a channel by channel basis. To utilize Greenplum Bulk loading versus straight JDBC insert, specify the Postgres Bulk Loader ("postgres_bulk") in the data_loader_type column of sym_channel.

C.6. MS SQL Server

SQL Server was tested using the jTDS JDBC driver.

SQL Server allows the update of primary key fields via the SQL update statement. If your application allows updating of the primary key field(s) for a table, and you want those updates synchronized, you will need to set the "Handle Key Updates" field on the trigger record for that specific table. The default for Handle Key Updates is false.

C.7. 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.8. 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.9. 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.10. 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.11. 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.12. 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.13. 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.

C.14. SQLite

For SQLite, the implementation of sync-on-incoming back and the population of a source node if in the sym data rows relies on use of a context table (by default, called sym_context) to hold a boolean and node id in place of the more common methods of using temp tables (which are inaccessible from triggers) or functions (which are not available). The context table assumes there's a single thread updating the database at any onetime. If that is not the case in the future, the current implementation of sync on incoming batch will be unreliable.

Nodes using SQLite should have the `jobs.synchronized.enable` parameter set to `true`. This parameter causes the jobs and push/pull threads to all run in a synchronized fashion, which is needed in the case of SQLite.

The SQLite dialect has the following limitations:

- 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.
- Due to the single threaded access to SQLite, the following parameter should be set to true: `jobs.synchronized.enable`.

C.15. Sybase Active Server Enterprise

Active Server Enterprise (ASE) was tested using the jConnect JDBC driver. The jConnect JDBC driver should be placed in the "lib" folder.

Columns of type DATETIME are accurate to 1/300th of a second, which means that the last digit of the milliseconds portion will end with 0, 3, or 6. An incoming DATETIME synced from another database will also have its millisconds rounded to one of these digits (0 and 1 become 0; 2, 3, and 4 become 3; 5, 6, 7, and 8 become 6; 9 becomes 10). If DATETIME is used as the primary key or as one of the columns to detect a conflict, then conflict resolution could fail unless the milliseconds are rounded in the same fashion on the source system.

On ASE, each new trigger in a table for the same operation (insert, update, or delete) overwrites the previous one. No warning message displays before the overwrite occurs. When SymmetricDS is installed and configured to synchronize a table, it will install triggers that could overwrite already existing triggers on the database. New triggers created after SymmetricDS is installed will overwrite the SymmetricDS triggers. Custom trigger text can be added to the SymmetricDS triggers by modifying

CUSTOM_ON_INSERT_TEXT, CUSTOM_ON_UPDATE_TEXT, and CUSTOM_ON_DELETE_TEXT on the TRIGGER table.

C.16. Sybase SQL Anywhere

SQL Anywhere was tested using the jConnect JDBC driver. The jConnect JDBC driver should be placed in the "lib" folder.

C.17. Redshift

Redshift is a managed data warehouse in the cloud from Amazon. Version 1.0 of Redshift is based on PostgreSQL 8.0, with some features modified or removed. SymmetricDS supports Redshift as a target platform where data can be loaded, but it does not support data capture. However, the initial load and reload functions are implemented, so it is possible to query rows from Redshift tables and send them to another database.

While Redshift started with PostgreSQL 8.0, there are some important differences from PostgreSQL. Redshift does not support constraints, indexes, functions, triggers, or sequences. Primary keys, foreign keys, and unique indexes can be defined on tables, but they are informational metadata that are not enforced by the system. When using the default data loader with SymmetricDS, it will enforce primary keys, either defined in the database or with the sync keys features, by checking if a row exists before attempting an insert. However, the bulk loader does not perform this check. The data types supported are smallint, integer, bigint, decimal, real, double precision, boolean, char, varchar, date, and timestamp.

A data loader named "redshift_bulk" is a bulk loader that can be set for a channel to improve loading performance. Instead of sending individual SQL statements to the database, it creates a comma separated value (CSV) file, uploads the object to Amazon S3, and uses the COPY statement to load it. The COPY command appends the new data to any existing rows in the table. If the target table has any IDENTITY columns, the EXPLICIT_IDS option is enabled to override the auto-generated values and load the incoming values. The following parameters (see Appendix B) can be set for bulk loader:

- **redshift.bulk.load.max.rows.before.flush** - When the max rows is reached, the flat file is sent to S3 and loaded into the database. The default is 100,000 rows.
- **redshift.bulk.load.max.bytes.before.flush** - When the max bytes is reached, the flat file is sent to S3 and loaded into the database. The default is 1,000,000,000 bytes.
- **redshift.bulk.load.s3.bucket** - The S3 bucket name where files are uploaded. This bucket should be created from the AWS console ahead of time.
- **redshift.bulk.load.s3.access.key** - The AWS access key ID to use as credentials for uploading to S3 and loading from S3.
- **redshift.bulk.load.s3.secret.key** - The AWS secret key to use as credentials for uploading to S3 and loading from S3.

To clean and organize tables after bulk changes, it is recommended to run a "vacuum" against individual tables or the entire database so that consistent query performance is maintained. Deletes and updates mark rows for delete that are not automatically reclaimed. New rows are stored in a separate unsorted region, forcing queries to sort on demand. Consider running a "vacuum" periodically during a maintenance window when there is minimal query activity that will be affected. If large batches are continually loaded from SymmetricDS, the "vacuum" command can be run after committing a batch by using a load filter (see Section 3.9) for the "batch commit" event, like this:

```
for (String tablename : context.getParsedTables().keySet()) {
    engine.getSqlTemplate().update("vacuum " + tablename, new Object[] { } );
}
```

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. Upgrading from 2.x

Please test carefully when upgrading SymmetricDS 2 to SymmetricDS 3. Note that OUTGOING_BATCH table's primary key changed. The automatic upgrade backs up and copies the table. This might take some time if the table is large.

The following parameters are no longer supported:

- `db.spring.bean.name` - The connection pool is no longer wired in via the Spring Framework
- `db.tx.timeout.seconds` - Transactions are no longer managed by the Spring Framework
- `db.default.schema` - The default schema is always the schema associated with the database user
- `db.jndi.name` - JNDI data sources are no longer supported
- `auto.upgrade` - Database upgrade is controlled by `auto.config.database`
- `routing.data.reader.type` - As of this release, there is only one data reader type.
- `job.purge.max.num.data.events.to.delete.in.tx` - The name of this property changed to `job.purge.max.num.data.event.batches.to.delete.in.tx`
- `web.base.servlet.path` - No longer needed
- `dataloader.allow.missing.delete` - Controlled by conflict detection and resolution
- `dataloader.enable.fallback.insert` - Controlled by conflict detection and resolution
- `dataloader.enable.fallback.update` - Controlled by conflict detection and resolution
- `dataloader.enable.fallback.savepoint` - No longer needed
- `db.force.delimited.identifier.mode.on` - No longer needed
- `db.force.delimited.identifier.mode.off` - No longer needed

The way extension points work has changed. SymmetricDS services are no longer Spring injectable into extension points. Please use the `ISymmetricEngineAware` interface to get a handle to the engine which gives access to services.

The following extension points are no longer supported:

- `IDataLoaderFilter` - Replaced by IDatabaseWriterFilter
- `IBatchListener` - Replaced by IDatabaseWriterFilter
- `IExtractorFilter` - No longer supported. Rarely used.
- `IColumnFilter` - No longer needed. Please use the transformation feature.

Appendix F. 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.