*MSync Replication*

Table of Contents

[1 Purpose 4](#_Toc484004471)

[2 Key Terms 4](#_Toc484004472)

[3 How It Works 4](#_Toc484004473)

[4 Publisher 4](#_Toc484004474)

[4.1 Structure 4](#_Toc484004475)

[4.2 Initial Setup 5](#_Toc484004476)

[4.3 Table Components 6](#_Toc484004477)

[4.4 Requirements and Limitations 6](#_Toc484004478)

[5 Subscriber 7](#_Toc484004479)

[5.1 Structure 7](#_Toc484004480)

[5.2 To Add or Remove Tables 8](#_Toc484004481)

[5.3 Requirements and Limitations 8](#_Toc484004482)

[5.4 Considerations on Subscription Separation 9](#_Toc484004483)

[6 Distribution 10](#_Toc484004484)

[6.1 How It Works 10](#_Toc484004485)

# Purpose

MSync is an open-source tool to help synchronize data across SQL databases. It was built independently by Marcelo Silva and is distributed as-is, with no warranty.

# Key Terms

Key terms:

* Publisher Server / Database – the location where data comes from.
* Subscriber Server/ Database – the location where data goes to.
* Distributor – the server that transfers data, could be the publisher, the subscriber or a separate server.
* RowVersion - An incrementing number automatically updated by the database on every insert/update. For every published table a column name RV of type Rowversion is added. More details at: <https://docs.microsoft.com/en-us/sql/t-sql/data-types/rowversion-transact-sql>

# How It Works

The data sync process consists of 3 major steps: Extraction, Transfer and Merge. MSync uses highly efficient strategies on all these steps.

1. Extraction: For each published tabled we add a column named RV, of data type rowversion, to capture inserts and updates. We also add a trigger (for delete only) to capture the keys of deleted records. During the first load all records are pulled, on subsequent loads only the records that changed sync last sync are pulled. Stored procs, narrow indexes and seek operations are used to minimize impact at the publisher.
2. Transfer: A C# console application bulk copies tables in parallel into a staging area. Minimally logged operations and compression options are available.
3. Merge: Stored procedures update subscriber tables from staging tables using set operations.

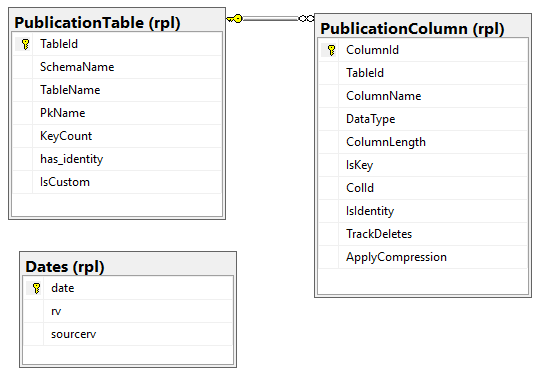
All components are created automatically by the framework. No manual development of routines or packages is needed. All operations are performed in batches, and multiple tables are processed in parallel.

Before utilization setup scripts must be applied at the publisher and subscriber. This configuration needs to be done only once, and it creates tables and stored procedures to assist managing what tables get published/subscribed.

# Publisher

## Structure

The Publisher is structured as follows:



These tables define what tables have been published and are available for subscriptions. Note that all components reside on the RPL schema.

To add or remove a table use:

exec rpl.spPublishTable 'schema','table'

exec rpl.spUnPublishTable 'schema','table'

These procs populate the tables above with details from the database schema. Differently from Microsoft Replication, the act of publishing a table does not actually publish it, but simply creates the means by which other databases may subscribe to them. When we publish a table automatically all columns are made available. Some may be removed manually for security purposes.

## Initial Setup

During initial setup we create the table rpl.Dates, along with the SQL Agent job 'MSync Checkpoint PublisherDb'. The job inserts a new date every minute, which serves 2 purposes:

* As a checkpoint – every subscription that gets data from this database will automatically pull this table, which will indicate how far behind a subscription is.
* As an approximation of when rows were modified. Rowversions are ever-increasing binary values that have no correlation with date. By looking up at this table we can know the minute a timestamp was generated, or the date a change occurred. This information is not used by replication itself, but is reserved for future use.

## Table Components

The components for each published table are:

1. **A column named RV, of type rowversion**. This is the heart of the change detection. All rows inserted and updated will be automatically marked by SQL Server. Users can read but cannot change those values.

This could cause issues when:

1. Inserts don't have a columns list.

Example:

insert into Tbl values (id, name) -- the values list does not match the column definition...

ii. Views that have select \* from multiple tables.

Example:

create view vTest as

select \* from a join b on a.id = b.id-- the column rv is duplicated...

1. **An index on RV column**: ix\_schema\_table\_rv. This allow for efficient index seeks during incremental extracts.
2. A stored proc **rpl.spGet\_schema\_table**. To extract the data to the subscribers. This proc will get only the last version for each record. If we change a record 10 times, delete, reinsert, etc., only the last version for the PK is retrieved.
3. To track deletes:
   1. **A table named rpl.del\_schema\_table** with same columns of the primary key of the published table. Other columns will indicate when the delete happened.
   2. **A trigger named trg\_rpl\_del\_table** on the published table for delete operation that stores the PK values of deleted records.

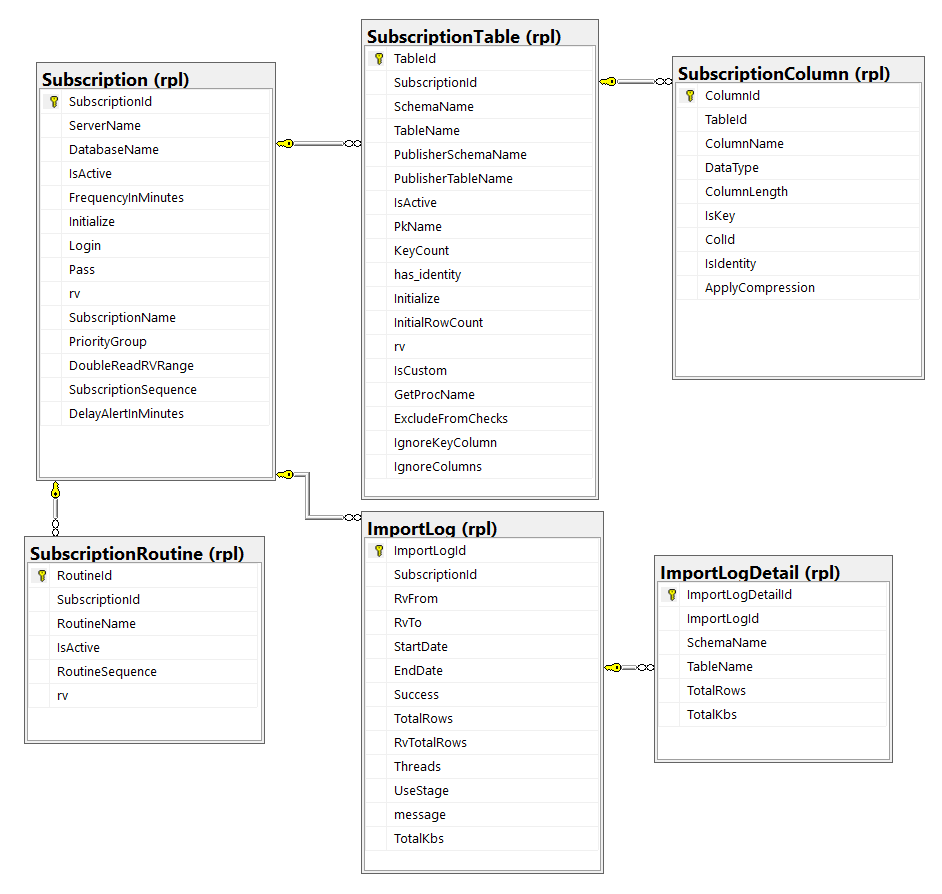
## Requirements and Limitations

1. In order for a table to be publishable, it:
   1. Must have a primary key or unique index. The index with lowest ID will be used as Key.
   2. Cannot already have a column named RV that is not of type rowversion.
   3. Cannot already have a column of type rowversion or timestamp.
2. Key columns are non-updatable for replication purposes. If you wish to modify key columns you must delete the old key values from the subscribers. Updates to primary keys will be propagated as new rows. Another approach is to mark subscriber tables to be re initialized.
3. There should be no limitation of data types. Blob types should supported including TEXT, VAR… (MAX) and IMAGE.
4. If you wish to truncate a published table, you need to also manually truncate the subscribers. MSync will not prevent truncates, and will not automatically truncate subscribers either.
5. Tracking Deleted Values – By default only the key columns are stored in the delete table (rpl.del\_schema\_table). You may store more columns there by turning on TrackDeletes in rpl.PublicationColumn. These values must be manually updated, then you need to publish the table again with rpl.spPublishTable.
6. Compressing Large Text values – By default compression will not be used. This feature requires SQL 2016 or later, and it may be useful for large texts such as product descriptions. You may enable ApplyCompression in rpl.PublicationColumn, then you need to re publish the table with rpl.spPublishTable. You should make sure subscribers have compression enabled in rpl.SubscriptionColumn.

# Subscriber

## Structure

The Subscriber is structured as follows:



These tables define both the scope of data subscribed and the import execution log. Note that all components reside on the RPL schema.

## To Create a Subscription

A subscription defines where the data comes from.

exec rpl.spCreateSubscription

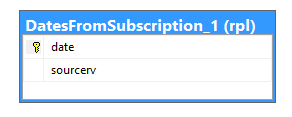
@Name = 'FriendlyName'

, @Server = 'PublisherServer'

, @Database = 'PublisherDatabase'

The first time a subscription runs, all tables will be initialized, meaning that all data will be deleted and reloaded. A table can be part of only one subscription.

With every subscription, a table is automatically added to represent how far behind the subscription is:



The number in the end of the table name represents the subscriptionid, and this table gets populated with rpl.Dates from the publisher database.

Column definition of table rpl.Subscription:

* SubscriptionName – this is just a friendly reference name
* PriorityGroup – this allows replication jobs to isolate subscriptions in channels. A priority group is just a separate lane, it does not mean the data will have priority by itself. For instance catalog subscriptions have 2 priority groups, number 1 is used for product data, and number 2 for everything else, so a large product update will not affect customer data from being propagated, and vice versa.
* ServerName – this is the name or IP of the publisher server
* DatabaseName - this is the name of the publisher database
* IsActive – only active subscriptions will be picked up by the console application, unless you pass the subscriptionid in the 4th parameter, in this case the subscription will be executed, whether active or not. This gives the ability take a subscription out of the main jobs and run it manually, for instance when you need to initialize certain tables.
* FrequencyInMinutes – this determines the minimum amount of minutes a subscription will be picked up since its last successful run. This allows creating a job with high frequency, but not running the same subscriptions all the time. This feature is not currently in use and all frequencies are set to zero.
* Initialize – setting this column to 1 will cause the subscriber tables to be truncated and fully reloaded. The bulk copy will skip staging tables and go directly into the subscribed table. Foreign key constraints will be dropped and recreated. Non clustered indexes will be disabled and rebuilt. If the table has other schema bound objects they must be manually dropped and recreated.
* Login – if the publisher server is in a different domain this is the login to be used while connecting to the publisher. This account must have read and execute rights on the rpl schema. If this column is blank then trusted authentication will be attempted.
* Pass – this is the password for the login above. If you pass this value on spCreateSubscription you must enter the unencrypted value, however it will be stored encrypted. If you update after the subscription is created you need to encrypt the value with the function dbo.fnEncrypt.
* DoubleReadRVRange – Each time a subscription runs it uses the same Start and End RV values as range for all Get procs. The End value is the current timestamp from the publisher. This is needed because as we read data new changes may be happening and new RVs get created. The Start RV is typically the last successful End RV value. However in tables with high activity or long lasting transactions some RV values may have been generated, but are still not available for read at the time the Get proc gets executed. By setting this column to 1 will cause replication to use the RV from 2 runs ago, which nearly causes every update to be propagated twice to stage. The actual update happens only once, because every merge proc updates only if the sourcerv has changed. This strategy is acceptable when network bandwidth is not an issue or we want to propagate changes as soon as possible. An alternative strategy, If you can tolerate a longer propogation delay, is to modify spGetCurrentTimestamp at the publisher to return the timestamp a few seconds (or minutes old), this way we make sure the recently generated RV values are available for reads.
* Rv – this is just an indicator of when the row was last modified. To convert it to a data you may use the function rpl.fnRvToDate, which gets created by the master publisher script.
* SubscriptionSequence – Defines the order subscriptions are executed within a PriorityGroup. If all values are the same then execution order follows SubscriptionId
* DelayAlertInMinutes – By default all subscriptions are alerted after 30 minutes of latency. This column will overwrite the default value.

## To Add or Remove Tables

To add/ remove tables we use:

exec rpl.spSubscribeTable SubscriptionId, 'Schema','Table'

exec rpl.spUnSubscribeTable SubscriptionId, 'Schema','Table'

Where the first parameter represents the SubscriptionId. The tables SubscriptionTables and SubscriptinColumns get populated with SQL Server system data.

For each table the system creates:

1. A column sourcerv of type varbinary(8), these do not have indexes and do not get automatically updated by SQL. They serve only for reference and it contains the same value as the RV column at the publisher.
2. A table rpl.stg\_schema\_table (operation  char(1) + same columns as target table). This table receives a bulk copy from the results of rpl.spGet\_schema\_table from the publisher database.
3. A stored proc rpl.spMerge\_schema\_table. This proc merges the staging table into the production table.

Column Definition for table rpl.SubscriptionTable:

* SchemaName / TableName – self explanatory
* PublisherSchemaName / PublisherTableName – if the table is different at the publisher we may update these values manually
* IsActive – a table may be temporarily deactivated, which is useful when we need to run large updates, when can manually update publisher and subscribers, leaving the table inactive, so that the rows don’t get picked up by the job. This mechanism needs however a method to keep track of updates to the publisher other than the manual one, for instance by creating a trigger or relying on DateLastChanged. When we re enable the table we can force the rows to be pushed with a dummy update.
* PKName – If you don’t specify one on the SubscribeTable proc the system will automatically pick the unique index with lowest ID.
* KeyCount – number of columns in PKName, for reference only
* Has\_identity
* Initialize – this provides a mean to initialize a single table, but the data will be copied to stage then merged to prod. This allows taking advantage of compression.
* InitialRowCount - the console app uses this information to determine the order in which to process the tables, starting with smaller ones.
* IsCustom – enabling this column will prevent the SubscribeProc from recreating stage tables and procs, which is useful if you customized the process.
* GetProcName – by default the process will look for a procedure names rpl.get\_schema\_table at the publisher, if you for whatever reason want to use a different proc you may use this column. This is used by the Product custom distribution, Large text columns updated infrequently go by one way, and small columns updated frequently go by another way.
* ExcludeFromChecks – part of the daily checks is to compare rowcounts between publisher and subscriber and alert when thresholds are surpassed.
* IgnoreKeyColumn – In some cases the subscriber may have a different key with more columns.
* IgnoreColumns – Some columns may be updated locally, and we don’t want values overwritten from the publisher.

We may also add/remove routines. This feature is useful when we need to perform post-import steps, the syntax is:

exec rpl.spSubscribeRoutine 1, 'schema.RoutineName'

The routine must be a store proc, and must receive the parameters: @subscriptionId int, @rvfrom varchar(20), @rvto varchar(20).

Routines can be deactivated and have their execution sequence updated in the table rpl.SubscriptionRoutine.

This feature is used by the product custom distribution, and the reverse replication that pulls Discounts and Promos applied from LocalCheckouts to DistDb, then to Sales, and finally to Repdb. This is also used by catalog distribution.

Product Rank and reviews are constantly updated in Purchasing and Shop. And our de-normalized design was causing all product descriptions to be sent, in multiple languages, every time a rank changed. Network bandwidth was a problem, so we changed the distribution to propagate the normalized changes, and re-generate the large records with replication routines at the catalogs.

## Requirements and Limitations

1. The subscription table must already exist. Unlike Microsoft Replication, MSync does not create tables automatically.
2. In order to be subscribed or published, a table:
   1. Must have a primary key or unique index. The index with lowest ID will be used as Key.
   2. Cannot already have a column named SOURCERV, that is not of type varbinary(8).
   3. Cannot already have a column named RV, that’s is not of type rowversion.

Notice a view may also be published, as long as it has a unique index, like an indexed view. The subscribed table name may be different than the published table name, however the key columns must be an exact match in number, names and types. Only non-key and non-identity columns get updated. The non-key columns in SubscriptionColumn may be a subset of PublishedColumn, but every column in SubscriptionColumn must exist in PublishedColumn.

1. Subscribed tables should be considered ReadOnly. Although the replication does not block updates, any changes to local tables may be overwritten with data from the publisher.
2. MSync detects all rows inserted/updated/deleted from publisher and pushes to subscriber, but if rows are changed in subscriber they will not be reverted. To do so, we need to reinitialize the subscriber.
3. A subscribed table may be published to other subscribers, however all changes should come from the parent publisher, and the subscribed table should work only as a broadcast hub.

## Optional Features

**Column Compression**This feature is useful for large text fields, such as product description. The process will apply compression on the Get proc, copy compressed data to Stage, and decompress during Merge, saving it decompressed to the subscription table. Notice this applies only for incremental loads, because when the whole subscription gets initialized the data is bulk copied directly into the subscription table. The steps are:

* 1. **At Publisher**
     1. Run spPublishTable.
     2. Manually update rpl.PublicationColumn set ApplyCompression=1
     3. Run spPublishTable a second time. This will modify the Get proc to apply compression during read.
  2. At subscriber
     1. Run spSubscribeTable.
     2. Manually update rpl.SubscriptionColumn set ApplyCompression=1
     3. Run spSubscribeTable a second time. This will modify the Stage table and Merge proc.

**Tracking Deletes**

This feature is useful when we need special handling of deleted data, such as using SubscriptionRoutines. By enabling TrackDeletes the process will add that column to the delete log table and the delete trigger. Notice all columns marked with IsKey=1 will automatically be tracked. The steps are:

* 1. Run spPublishTable.
  2. Manually update rpl.PublicationColumns set TrackDeletes=1
  3. Run spPublishTable a second time.

## Considerations on Subscription Separation

You may define several subscriptions, from multiple publishers or even from the subscriber database itself. You may also create multiple subscriptions from the same publisher, which allows table groups to be pulled at different schedules. Another advantage is greater independence between priority groups, however this may yield to inconsistencies on reports. For instance, let’s assume we want to have a subscription for products and coupons, and for some reason and copy to products is failing, then the rv pointer never moves forward for both tables. If we separate the tables into two subscriptions then the rv pointer for coupons will move forward independently, and may we may end up with coupons for products that don’t exist. This is particularly useful for initializing large tables, which can be loaded as part of a separate subscription. Keep in mind parallelism happens only within the scope of a subscription, so if a subscription has only one table then only one thread will occur.

The more threads the less the total execution time will be, provided we don’t overwhelm storage/network throughput. On copies in the same network we may use one thread per subscriber CORE. On copies across data centers we observed that the network was the bottleneck.

Notice a table may be marked for initialization individually, or by initializing the whole subscription. In the first case the data gets copied to stage then merged to prod. If a table las a lot of large text fields and we are using the Apply Compression feature this may be a good strategy. When we initialize the entire subscription the data gets copy straight into the target table, non-clustered indexes are disabled and rebuilt, avoiding fragmentation, and log growth is minimized, however compression cannot be used.

Both strategies have their advantages, and initializing big tables should be planned carefully.

If a table has a lot of rows, and not many columns candidate for compression, this is a strategy to consider:

1. Disable the existing subscription (to prevent it from running)
2. Create a new subscription
3. Move tables that need initialization to a new subscription (update subscriptionid on rpl.substricptionTable)
4. Run import
5. Move tables back to the original subscription
6. Delete or Disable the new subscription and re-enable the old one.

Note: A savvy reader may raise concerns about inconsistencies in the RV ranges for the different subscriptions. Please notice that when we move tables back to the old subscription that RV range will be earlier than the new one, so there will be some data transferred twice, which brings an overhead, but does not risk inconsistency. Depending on the size of the table, the tradeoff between this overhead versus the need to copy rows twice and defrag indexes may be well justified.

# Distribution

## How It Works

The data gets copied by a console application written in C#. This console application is a blind data pump. It has no awareness of where the data comes from, where it goes to, and what goes in it. All the wiring is built at runtime from the metadata tables.

The program is named MSync.exe, and it takes 4 input parameters, separated by spaces, and surrounded by double quotes.

1. The first parameter represents the connection string to the subscriber, and is mandatory. The connection parameters to the publisher are stored at the subscriber, so this information is not needed for the command line.
2. The second parameter represents the number of threads, and it’s optional, with a default value of 4. The more tables are processed in parallel the faster replication will run, but the more server and network resources will be drained. For subscriptions with a lot of tables we can use 10 to 12 threads for initialization and 4 to 6 for incremental loads.
3. The third parameter represents a priority group, and it’s optional.
4. The fourth parameter represents a subscriptionid, and it’s optional. This is useful when we need to initialize a particular subscription which contains large tables. For instance to initialize a catalog without affecting the others, we may set one subscription to active=0, initialize=1, then run a manual sync passing the specific subscriptionid.
5. The fifth and last parameter represents batch size, and it’s optional, with a default value of 1000. If the table in the subscription has very small rows, for instance key columns plus a date, we may benefit from using larger batch sizes. Conversely, if a table has large rows, such as with long text values, or the bandwidth to the subscriber is slow, we may benefit from using smaller batch sizes.

Example:

MSync.exe "Data Source=.\S16;Initial Catalog=MyDb;Integrated Security=true;"

The blue section represents the subscriber database and access credentials.

**Moreover:**

* It can be run from any shell, but our preferred method is SQL Server Agent.
* It can run from computer that can communicate with both publisher and subscriber servers, as long as it has the .NET framework 4.5, although our preferred method is to keep it at the subscriber side. On catalogs however all jobs run on the distributor server.

**Sequence of Steps**

This sequence of steps outlines the replication logic, which is useful for troubleshooting errors and documenting the components:

1. spGetSubscriptions - Returns all active subscriptions that haven’t executed since “Subscription.Frequency” minutes. These will be executed in a loop.
2. Publisher.*spGetCurrentTimestamp* - returns the current timestamp at the publisher and defines the upper boundary (rvto) for all get procs. If new changes come during the load they will be included on the next load.
3. Subcriber.*spStart* – generates entry in rpl.ImportLog
4. If Initialize (rvfrom = 0x, or there is no log where success=1, and subscription.initialize=1)
   1. Subscriber.*spDisableIndexes*
   2. Subscriber.*spCleanProdTables* -- this deletes all data from the subscription db tables!
5. Subscriber.*spTruncateStage*
6. Subscriber.*spDisableConstraints*
7. Subscriber. *spGetTableList* – for each table, in parallel according to the number of threads
   1. Subscriber. spReturnGetProcName
   2. Pulisher.*GetProc*
   3. Subscriber.*spGetColumnList*
   4. BulkCopy into Stage (or Prod if load is initial)
8. If load is not initial (for each table, in parallel according to the number of threads)
   1. Subscriber.*spMerge*
9. If load is initial
   1. Subscriber.*spEnableIndexes*
10. Run Subscription Routines order by Sequence
    1. Subscriber.*spGetRoutineList*
11. Subscriber.*spEnableConstraints*
12. Subscriber.*spEnd* – updates end time, message, rowcounts and unmarks tables set for initialization.