**Net Change Pattern Overview**

This document describes four different options for tracking changes (net-change) within an application. These net-change mechanisms, or patterns, can be used to synchronize data with another system. Many Scribe Insight implementations fall into one of these categories; however, it should be noted that these four patterns are not exhaustive. An integration can incorporate elements from one or more of these patterns.

The four net changes patterns are,

- **Pattern 1** — Application publisher
- **Pattern 2** — Update source
- **Pattern 3** — Modified date/time stamp
- **Pattern 4** — Snapshot comparison

Each of these patterns is explained in detail, below.

**Pattern 1 — application publisher**

**Pattern 1 overview**

This pattern uses an application-specific publisher that plugs into an application notification (or call-out) mechanism. Few application architectures support a notification capability, so the cases where this pattern can be applied are limited. However, when available, this pattern combines the best real-time support with the greatest efficiency.
Pattern 1 pros

Provides real-time notification of changes (event driven).

No polling burden or cost to either the application or database server (more efficient).

Usually supports deletes.

Not prone to the missed updates issue (see pattern 3, below) when filtering by user to avoid bounce back.

Pattern 1 cons

Requires special application and adapter specific publisher support. This capability can only be provided in an adapter if the application architecture provides a change notification or call-out capability. Most application architectures do not support or provide change notification or call-out capability.

Pattern 1 implementation tips

Changes tend to be queued in the order in which they occur, but this is not guaranteed. Design your integration processes and DTS files to handle out-of-sequence messages (for example, make use of automated retries via message queue support).

If trying to send header/line-item (or master/detail) changes together, determine whether required line-item changes will always trigger a header notification. If not, register for notification of line-item changes as well. You may be able to
configure both header/line-item and line-item only changes to return messages with the same structure. The preferred method for capturing header/line-item changes is to wait for a specific user-requested action (like clicking a Submit button). This will prevent the system from sending multiple messages when a header and several line-items are entered or modified.

**Pattern 2 — update source**

**Pattern 2 overview**

This pattern uses the update source feature to toggle a synchronized flag or field. This approach depends on application (or source system) logic to toggle the synchronized flag off whenever a change is made. This discussion assumes that database triggers and a shadow table have been added to the application database to perform this function.

Triggers are added to each application base table for which you need to track changes. The triggers add or update a record in the shadow table to track the change. One shadow table tracks changes to records in multiple application tables by using a combination of an object (table) name field and the object key (record primary key). If the base table uses a compound primary key, the fields are concatenated in the shadow table object key field. The shadow table record and the base table record have a one-to-one relationship. When a base table record is created, the shadow table record is also created. After that, when the base table record is updated or deleted, the corresponding shadow table record is updated.

To synchronize changes to another system, a query publisher (or query integration process) is used. The source query joins the shadow table to the base tables to get all of the current field values. The shadow table contains multiple SyncStatus fields that toggle back and forth when a record is modified and when it is synced (if using a query integration process) or sent for sync (if using a query publisher).

The following diagram shows two separate approaches to implementing this pattern.

Pattern 2a shows the Scribe query publisher used to retrieve changed records from the source tables, creating messages in the ScribeIn message queue. Messages in the ScribeIn queue are then processed by a queue based integration process.
Pattern 2b shows a query-based integration process retrieving the changed records directly from the source and then processing the rows sequentially.

**Pattern 2 pros**

Supports deletes by keeping the shadow table record after the base table record is deleted.

Supports two-way integrations and prevents bounce back by including a reliable ignore-user mechanism. Not prone to the missed updates issue (see pattern 3, below) when filtering by user to avoid bounce back.

The shadow table provides an extra match key field that can be utilized for natural key matching (such as with the StripCompany algorithm).

Supports multi-cast of changes by providing multiple SyncStatus fields. These fields track the status of changes with up to five other systems. If a query publisher is used, a change can be published to the queue once, and then can be applied to (or subscribed to by) multiple queue integration processes.

Retry mechanism is inherent to the SyncStatus and query approach. This pattern doesn’t require separate rejected row
DTS files and integration processes. Retry can be configured to stop after greater than \( n \) minutes by comparing the modified date/time stamp to the current time minus \( n \) minutes.

The shadow table provides helpful information and history when troubleshooting problems.

**Pattern 2 cons**

If the application can not be configured to toggle a field value on every insert and update, database engine triggers will probably be required. To do this, the database engine must support triggers.

The database trigger approach requires detailed knowledge of the physical database schema in order to develop the script and triggers. Also knowledge of trigger coding is required.

If using the trigger and shadow table approach, a direct connection to the database is required.

Triggers can be affected or dropped when an application upgrade is performed. The trigger script must be saved so it can be reapplied if this happens.

**Pattern 2 implementation tips**

Use the sample script (ScribeTriggers.sql) that is provided by Scribe as a head start. This script can be found in your Scribe program files folder. Follow the comments to copy and paste triggers for each table and then find and replace table and field names.

Application logic or trigger code should be designed to ignore the change (don’t toggle the sync flag) if the change was applied by a specified “ignore user.”

If you want to track changes to parent and children records together, and bring the header and detail data over as a single “transaction,” change the triggers for the child table in the following ways:

- Do not have a separate trigger for insert and another for update and delete. Use the update/delete trigger code to create an insert/update/delete trigger on the child table.
- Use the foreign key to the parent record as the value for **ObjectKey**.
Set the ObjectName = ‘ParentTable’ (replace the formula with a constant string).

Set the Operation = ‘U’ (vs. a dynamic U or D value).

This will result in a child table trigger that always updates the shadow table entry for its parent record as modified (and has no matching shadow table entry on its own).

Set the IgnoreUser value appropriately within the ScribeSettings table. You can set a global ignore user value across for all five SyncStatus fields (see ScribeIgnore as an example), or you can set an ignore user value just for one conversation with one other system (see Scribe2 as an example).

Multi-segment keys: If you need to concatenate multiple fields into the ObjectKey field, always space pad the field values to a consistent length. This will match the capability built into the key cross reference feature.

If you want to enable special matching for a particular object or table, use a stored procedure like SCRIBE_SP_STRIPCOMPANY to generate your match value and then place the returned string into the MatchKey field.

You may sometimes find situations where a table has a “record type” field that distinguishes different types of records. An example is a Company table that can include info for two types of companies: customers and vendors. If you only want to track changes for one type, then you can add filtering into the “select … from inserted” and “select … from deleted” queries within the trigger. Be sure to add the same where filter into the query used to initially populate the shadow table. If you want to track changes for both types, but you want the multiple types to map to separate integration processes, then prefix the type onto the ObjectKey value. This way you can add filtering into your source queries (for example, “... and where ObjectKey like ‘Vendor%’

Do not rely on the last Operation field (in the shadow table) to always be accurate. It can be misleading due to timing issues and when tracking child records changes as a change to the parent record. This field is intended primarily for troubleshooting information.

Use a retry timeout if using a query integration process. This is not needed when using a query publisher.

Sample where clause with a 15-minute timeout.
For example,

```sql
where (z.SYNCSTATUS1 = 'N' or z.SYNCSTATUS1 = 'M') and datediff(minute, MODIFIEDDATE, getdate()) < 15
```

If using a query publisher, the “and datediff(...) < 15” can be dropped.

The sample ScribeTriggers script uses ‘N’ to tag new records, ‘M’ to tag modified records, and updates the source with ‘S’ to tag sent and synchronized records.

**Pattern 3 – modified date/time stamp**

**Pattern 3 overview**

This pattern uses a modified date/time stamp field in the application base table. Many application tables have these built in. Make sure the date/time stamp includes time information (not just date). If your application table does not have a modified date/time field then you may be able to add one, but this will generally require application modifications, database triggers, or both. Scribe provides built-in system variables that track the execution time of each publisher or integration process, which can be used as “bind variables” within a source SQL query and compared against the modified field to get “what’s new since the last time this publisher/process ran”. This capability can be used in conjunction with either a query publisher or query integration process.

This pattern is subject to a few different reliability problems (outlined below). If the synchronization will only be done one-way, then one of the issues is avoided. Integrations which use a modified date/time stamp are therefore best suited for one-way environments (and not two-way).
**Pattern 3 pros**

Supports multi-cast of changes to multiple systems, since the system variables are tracked independently for each publisher and each integration process.

Setup is simple if the application tables already include a modified date/time field.

This pattern can be used with any adapter or data source that supports filtered queries, and does not require a direct connection to the SQL database.

**Pattern 3 cons**

If you are implementing a two-way synchronization you need to prevent bounce-back (endless looping). Applications that have a modified date/time stamp often include a **ModifiedBy** or **ModifiedUser** field as well. This can be used prevent bounce-back, but it can result in missed updates. For example, when one user makes a change and the “ignore user” changes the
same record before the query integration process (or query publisher) detects the change, the first change would be ignored.

This method presents significant challenges with clock synchronization if the ModifiedDate field is not set using the clock of the source database server (or application server, if appropriate).

This pattern typically does not support deletes. If the application performs “soft deletes,” deletes may be accessible. However, deletes can be unreliable in this case, since the cleanup of soft deletes is usually unpredictable.

If using a query integration processes (3b), this pattern requires separate rejected row DTS files and integration processes to support automated retry.

Source queries can get complex if you are tracking changes on parents and children together (joining them as one result set). In this case the modified stamps on the parent and child records need to be checked.

Pattern 3 implementation tips

Be sure the Modified field includes both date and time (if only date is available then the frequency of synchronizations can not be greater than daily).

Be sure the Modified field is database (or application) server based. Sometimes the modified stamp field will be set using the clock on the network client computer, and this will cause significant problems when changes are made by clients with clocks which are out of sync. If the application supports remote or intermittent synchronization of laptops, sometimes the modified stamp field is the time entered or modified on the remote laptop. To capture changes accurately and reliably, this field must be set using the server clock! If Scribe is using a direct database connection, then the modified stamp must be set using the database server clock to be reliable. If Scribe is using an adapter connection to an application API (not at the database layer), the stamp should be set using the application server clock.

Use the system variables LastRunDateTime and ThisRunDateTime.

For example,

\[
\text{where ModifiedOn} \geq \text{:LastRunDateTime}
\]
and ModifiedOn < :ThisRunDateTime
and ModifiedBy <> ‘Scribe’

Be sure that the modified date/time field is an indexed field within the database. If it is not indexed individually, but only as part of a multi-segment index, then make sure it is the first segment (in at least one index).

If you are implementing a two-way synchronization, investigate whether the updating of the modified stamp field can be suppressed for changes coming from Scribe. If so, a ModifiedBy field is not required to prevent bounce-back, and the missed updates problem will be eliminated.

If capturing changes to parent and children records together (header and line-items), then check to see if all child required record changes result in a change to the parent modified stamp. If so, the parent stamp can be used exclusively.

Pattern 4 – snapshot comparison

Pattern 4 overview

This pattern uses a feature within the Scribe query publisher called snapshot comparison to track a complete copy of the source data, and compare the current source result set to the result set from the last execution. By comparing all source data row-by-row and field-by-field, all new, modified, and deleted records can be detected. However, this approach can be very inefficient, and is best suited for synchronization once a day (or every few hours). The processing time required and the inefficiency of this approach is determined by the number of records in the source and the percentage of records that typically change within the scheduled time. To maximize the efficiency of this process, this pattern requires that the source result set be ordered by a unique key field (or set of fields). The snapshot data is retrieved from the Scribe internal database in the same order, and the result sets are traversed in lock-step.

This pattern can be used by querying the source data directly, or by processing data that has been exported to a file or staging table.
**Pattern 4 pros**

This pattern can be used with almost any data source. Special system and application requirements are minimal. This approach works equally well when accessing source data directly and when processing an extracted copy of the source data.

This pattern can be used with any adapter or data source that supports ordered queries, and does not require a direct connection to a SQL database. If the data source does not support ordered queries, the data must have been pre-sorted.

This pattern is not prone to the missed update problem that can occur when filtering on user to avoid bounce-backs.

Supports reliable delete detection.

Processing can be timed or sequenced to prevent dependant record sequencing issues.

If a change is made to a source field that is not included in the integration, then no processing is performed. This reduces the volume of no-change messages that are processed, and thus reduces the potential for update conflicts.

Header and line-item records can be easily grouped together into one message (and one comparison) to detect any changes to the
group, and generate a single header and line-item message or transaction.

Setup of this mechanism is the simplest and least error prone.

**Pattern 4 cons**

This approach is a very inefficient way to detect changes, and can take many minutes of processing to find just a few changes.

Due to the long time required to process all source records, the frequency support is low (usually only daily).

This pattern does not include any mechanism to prevent bounce-back. Therefore all changes originating in the target system will make a complete cycle back to the target system. However, the cycle will end there, even if this same net-change pattern is utilized to synchronize changes from the target system. The bounce-back messages can cause update conflict issues when competing changes happen in a short time-frame and in a specific sequence.

**Pattern 4 implementation tips**

The snapshot comparison feature of the query publisher can also be used with patterns 2a and 3a to ignore changes to source fields that are not needed for the integration. This can be particularly useful in cases where a regular batch process touches many of the source records, but doesn’t affect any fields involved in the integration. However, this can add a significant amount of processing time whenever the source is polled for changes. When using this method, the source result set must be ordered by unique key fields. This hybrid approach does not add support for detecting deletes (in cases where deletes are not already supported).
## Pattern comparison summary

<table>
<thead>
<tr>
<th>Pattern Number</th>
<th>Pattern Name</th>
<th>1 Application Publisher</th>
<th>2 Update Source</th>
<th>3 Modified Stamp</th>
<th>4 Snapshot Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Near real-time (low latency)</td>
<td>H</td>
<td>M</td>
<td>M (if indexed)</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>Reliability (data consistency)</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>Availability</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>Supports deletes</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ease of development, deployment, an upgrades</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>Ease of administration and troubleshooting</td>
<td>M</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>Overall weighting</td>
<td>17</td>
<td>15</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

### Key

- **H**: High (very beneficial, 3 points)
- **M**: Moderate (2 points)
- **L**: Low (less beneficial, 1 point)

---

© 2007 Scribe Software Corporation

13