

Synchronizing Data with TDI 6.0 Fixpack 3

by Eddie Hartman

Revised: 15 May 2006

Table of Contents

1	How to use TDI to Synchronize Data	3
1.1	Introduction	3
1.2	The Entry Object	4
2	Delta Detection.....	7
2.1	Change Detection Connectors	8
2.1.1	Iterator State – keep track of changes handled.....	9
2.2	The Delta Engine.....	10
2.2.1	Setting Up the Delta Engine.....	11
2.2.2	Performance and Risk	12
2.3	LDIF Parser with incremental LDIF	13
3	Delta Tagging	14
3.1	Delta Operation Codes	14
3.2	Delta Mechanisms and Tagging Levels	14
4	Delta Application	18
4.1.1	Manual Delta Application	18
4.1.2	Delta Mode.....	20
4.1.3	Compute Changes	21
5	Working with Delta Operation Codes	23
5.1	Attribute and Value Operation Codes	24
5.2	Tagging Rules for Delta Operation Codes	25
5.3	Debugging Delta Operation Codes	26
5.4	Manual Delta Code Tagging	27
6	Conclusion.....	29
7	References	30

1 How to use TDI to Synchronize Data

The basic goal of data synchronization is to detect changes in one data source and then propagate these to one or more targets.

Discovering and then applying changes is not as easy as you might think. Some systems provide change event notifications, most do not. Many maintain some sort of modifications list, but the level of detail available here varies greatly. A few systems allow you to incrementally modify the values of selected attributes. However, the majority require you to build a full data entry with all updates in place and then write this in a single operation.

So how do you deal with the differences in both feature sets and change resolution found in the systems you want to sync? Who you gonna call? TDI gives you a framework for handling this at a comfortably abstract level. To take full advantage of these capabilities a certain amount of understanding is still required.

This document outlines the (updated) features in TDI 6.0 designed for building data synchronization solutions. It also provides insight into how to use them. However, you must already have some experience with TDI. At least, you should work through the video tutorials found here: <http://www.tdi-users.org/twiki/bin/view/Integrator/LearningTDI>.

1.1 Introduction

Data synchronization must be as fast and efficient as possible. This means propagating *only data that has been modified*. At the other end of the data-sync pipe at the synchronization targets, only *necessary changes* should be written. This minimizes system and network traffic, and to avoids triggering unnecessary replication. These two activities are called *Delta Detection* and *Delta Application* in TDI terms, respectively:

Delta Detection Discovering that a change has occurred in a data source and retrieving the information needed to propagate the change. This is discussed in section 2 *Delta Detection* starting on page 7.

Delta Application Using these operation codes to drive the changes to other stores/systems as efficiently as possible. Delta Application is detailed in section 4 *Delta Application*.

In addition, when Delta Detection reads in changes, it *tags* this data with information about *what* has changed and *how*. This information is then used during Delta Application to bring targets in sync with the source, and is called *Delta Tagging*:

Delta Tagging Storing change information in the retrieved data Entry so that it can be used in the next step, Delta Application. This is done by assigning (i.e. tagging) *delta operation codes* to the data. These codes describe the type of change: e.g. *add*, *modify*, *delete*, and so on. They are also referred to as “operation codes” and “delta tags” in TDI literature. More on this subject in section 3 *Delta Tagging*, page 14.

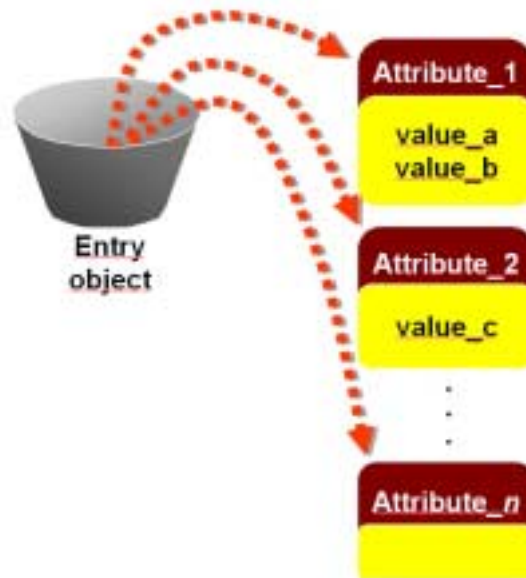
This may be starting to sound complex, so let me back up and say that some of the information here (for example, the details of Delta Tagging) may not play a vital part in your synchronization work with TDI. Don’t panic. Just keep reading.

There are specific features in TDI for detecting changes, just as there are for tagging data with delta operation codes and applying these tags to drive changes to target systems. As you may already have guessed, the remainder of this document is divided into three main sections, one for each of the aspects of Delta Handling.

But first, a little story about the TDI Entry data model.

1.2 The Entry Object

One of the cornerstones of understanding TDI is knowing how data is stored and transported internally in the system. This is done using an object called an *Entry*. The Entry object can be thought of as a “Java bucket” that can hold any number of Attributes (none, one or many).



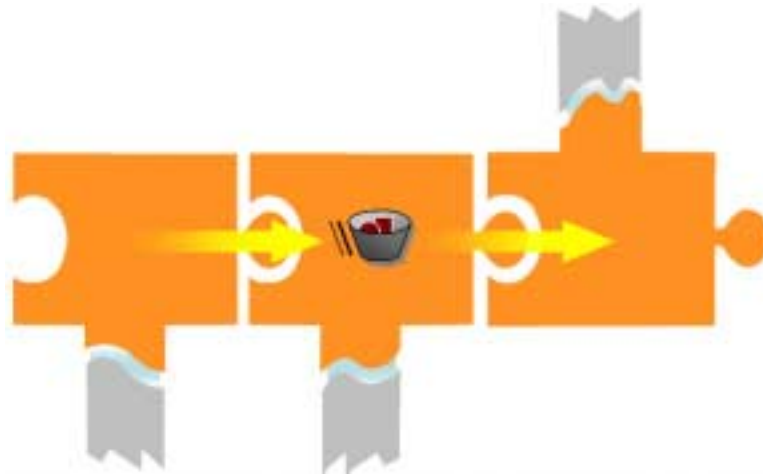
Attributes are also bucket-like objects in TDI. Each Attribute can contain zero or more *Values*, and these carry the actual data values that are read from (and written to) connected systems. These Attribute Values are Java objects as well – like strings, integers and timestamps¹ – and a single Attribute can readily hold Values of different types. However, the type of object used to store a Value is chosen by the component that reads it in, and is usually made at the Attribute-level. As a result, all the Values of a single Attribute will tend to be of the same type in most data sources.

Although the Entry-Attribute-Value paradigm matches nicely to the concept of LDAP directory *entries*², this is also how rows in databases are represented inside TDI, as are records in files, IBM Lotus Notes documents and HTTP pages received over the wire. All data – from any source that TDI works with – is stored internally as Entry objects with Attributes and their Values.

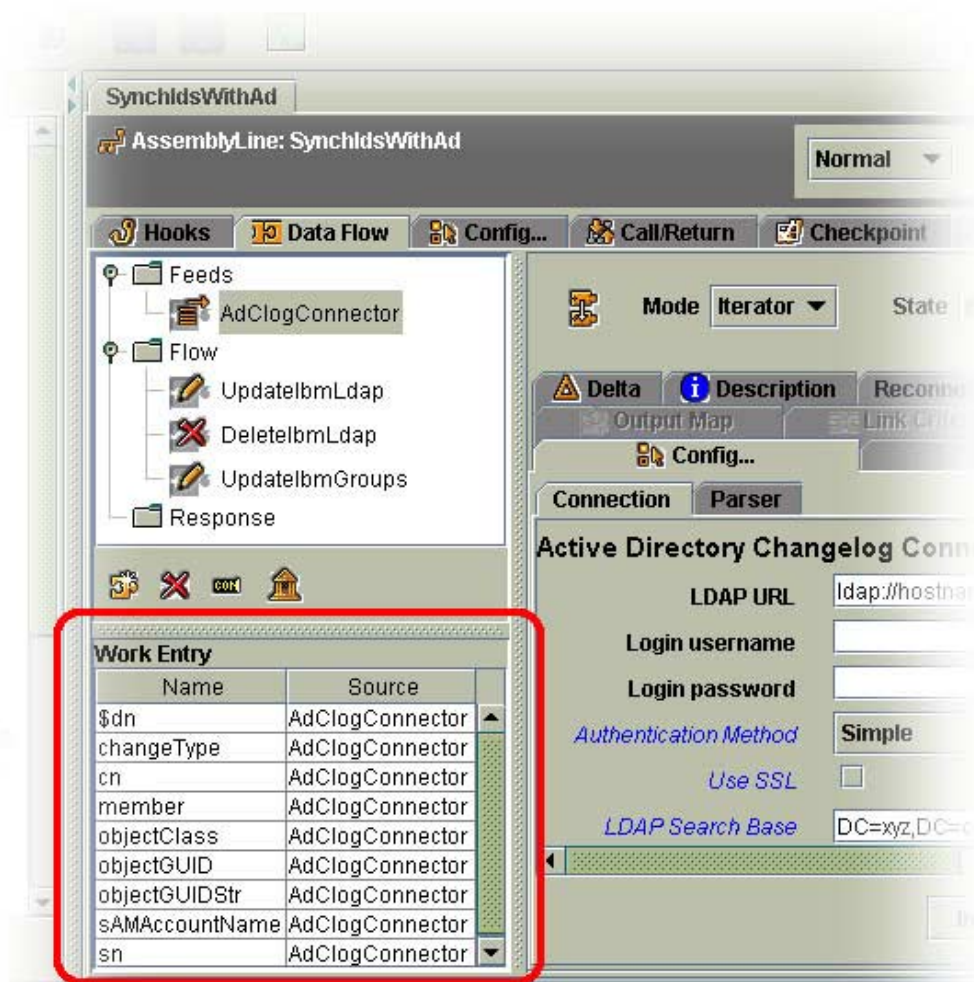
There are a handful of Entry objects that are created and maintained by TDI. The most visible instance is called the *Work Entry*, and it serves as the main data carrier in an AssemblyLine (AL). This is the bucket used to transport data down an AssemblyLine, passed from one component to the next during AL execution.

¹ Although the Config Editor does not support its display, an Attribute value can conceivably be another Entry object – complete with its own Attributes and *values*.

² Since TDI has borrowed a good deal of terminology from the directory space, you will see a distinction between terms that represent objects in the system (like an “Entry” object, which will be capitalized) and those that refer to concepts (e.g. a directory “entry”, written in lowercase).



The Work Entry is available for use in scripting through the pre-registered variable *work*, giving you direct access to the Attributes being handled by an AssemblyLine (and their Values). Furthermore, all Attributes carried by the Work Entry are displayed in the Config Editor in a window under the Component List of an AssemblyLine³.



³ Let me elaborate on this point: all Attributes that appear in Connector Input Maps and AttributeMap Components will be shown in the Work Entry window. If you add or remove Attributes from *work* using direct calls, then these will not be visible here.

So in summary, an Entry holds Attributes which in turn contains Values. These Values are Java objects used to represent data values in the connected system.

In addition to holding Attributes, an Entry can also have a delta operation code that describes how this record/row/entry has been changed (tagged by Delta Detection). The same applies to Attributes, which can also hold delta tags: one for itself, and one for each Value it contains. This is discussed in more detail in section [3 *Delta Tagging*](#) on page 14.

But first: armed with this knowledge of the TDI Entry data model, it's time to look at Delta Detection.

2 Delta Detection

Delta Detection is the discovery and retrieval of changes. The change information is then used to tag retrieved data with delta operation codes that reflect the type of changes made.

More often than not, the goal of a Delta Detection implementation is to return *only* the deltas. So if you are reading from a data source with thousands or millions of data entries, a typical run of your AssemblyLine will process only the subset that have changed.

Since different systems offer different change notification features (or not), TDI has a variety of ways of setting up Delta Detection:

- **Change Detection Connectors** are used for LDAP Directories, Microsoft ActiveDirectory, SQL databases and Domino/Notes.
- The **Delta Engine** is used to *compute* changes for systems that don't otherwise provide change notification (for example, flat files or other less sophisticated data stores). Any Connector in Iterator mode has a Delta tab for configuring Delta Engine behavior.
- The **LDIF Parser** tags Entries as it reads through an *incremental* LDIF⁴ file. Unlike the two preceding items above, the LDIF Parser does not “detect” changes. Instead, it interprets the delta codes found in incremental LDIF files, as these contains only information about changes to entries. If this Parser is used on a *full* LDIF (i.e. one that is not incremental) then no tagging will occur⁵.

The full list of Delta Detection features, along with details on how they operate is found in

⁴ Full LDIF files hold complete entries, while incremental LDIF files contain only *changes* to data.

⁵ Of course, you use the Delta Engine while reading full LDIF files (like those created from directory dumps) to detect changes between subsequent iterations.

Table 1 - Change Detection Mechanisms on page 15.

Regardless of the mechanism used, the end result is an Entry *bucket* with delta operation codes set, also known as a *Delta Entry*. Let's take a closer look at how each of these mechanisms does its magic.

2.1 Change Detection Connectors

A Change Detection Connector leverages features available in the underlying data source for locating and returning changed entries.

Some data sources provide full delta mechanisms – like LDAP directory changelogs – which are accessed via API or protocol-based calls. Other Change Detection Connectors need to do more *heavy lifting*, or rely on logic that must be plugged into the connected system. For example, the RDBMS Changelog Connector depends on stored procedures that maintain *changelog* data for specified tables. These shadow changelog tables are handled by the RDBMS Changelog Connector in much the same way that the LDAP Changelog Connector deals with a directory changelog.

Common to all these TDI components is that they operate in **Iterator** mode. Furthermore, they all poll the connected system looking for changes. As a result, they each offer a **timeout** parameter to control how long the Connector will wait for new changes to appear; as well as a **Sleep interval** option to determine how long to wait between polls.

Where supported, Change Detection Connectors also offer a **Use notifications** checkbox that instructs the Connector to register for change events instead of polling (for example, doing a persistent search into a directory). Note that this type of synchronous configuration can result in lost changes if the connection to the data source fails. Fortunately, TDI remembers the last change it handled and starts looking for new changes after the previous one, regardless of which change the notification indicated.

2.1.1 Iterator State – keep track of changes handled

Another important feature is that all Change Detection Connectors provide an **Iterator State Store** parameter for keeping track of the *last change handled*, so that next time you start your Sync AL, it can pick up processing changes where it left off.

This feature uses the System Store⁶ to keep track of the starting point for a Change Detection Connector (for example, the changenumber of a directory changelog). The value of the **Iterator State Store** parameter must be globally unique as it serves as the *key* in the System Store under which the Iterator State is kept. If you have multiple ALs that use Change Detection Connectors, they must each have their own Iterator state data.

The content of the Iterator State Store works in combination with Connector configuration settings provided for selecting the next change to process – the “Start at...” parameter. For example, the IBMDirectoryServer Changelog Connector provides a “Start at changenumber” parameter where you can enter the changelog number where processing is to begin. This parameter can be set to either a specific value (e.g. 42), to the first change (i.e. 1), or to “EOD” (End of Data). The EOD setting places the cursor at the end of the changelog in order to only process new changes.

⁶ The System Store is a feature of TDI used to persist operational data (like Delta Engine snapshots). By default the System Store feature uses the bundled Cloudscape/DB2e database, but can be configured to use DB2, Oracle, Microsoft SQL Server, or any other compatible RDBMS.

As long as no Iterator State Store is specified, the Change Detection Connector will continue to use the “Start at...” setting each time the Connector initializes and prepares to retrieve changes⁷. The same will happen even if Iterator State Store does have a setting, but there is no state value stored yet.

So, the very first time you run the AL with the Change Detection Connector there will be no previously set Iterator State Store information in the System Store, so the “Start at...” parameter(s) will be used. As the Connector cycles through the input source, it updates the Iterator State, keep track of changes processed. On subsequent executions, the “Start at...” settings will be ignored and the Iterator State Store value applied instead⁸.

Another important setting is “State Key Persistence” parameter, which controls *when* the snapshots are committed to the System Store. The default setting, which is recommended for most situations, is “End of cycle” which means that this Iterator state value is only persisted if the AL cycle handling the current change completes.

Where supported, using a Change Detection Connector is the simplest solution. However, the Delta Engine (described next) is also a proven, enterprise-strength tool for detecting modifications in data sources that do not provide change notification features.

2.2 The Delta Engine

When the underlying data store does not provide any delta information then you can use the Delta Engine to discover changes for you.

This feature is extremely handy when dealing with large sequential data sources (for example, HR files dumps), but can also be used in conjunction with other Delta Discovery components⁹.

The Delta Engine is available for any Connector in *Iterator* mode. It works by taking snapshots of incoming Entries and writing these to the System Store.

The first time the AssemblyLine executes, there are no snapshots. All Entries read in by the Iterator are written to the Snapshot Database and returned for AssemblyLine processing with the operation code “add”.

On each successive run, Entries read are compared with earlier snapshots. Based on this comparison, all differences are noted and tagged in the Entry, and the snapshot database is updated to reflect the new state of the data.

Note that only Attributes in the Input Map of the Iterator will be stored in the Delta table and used to identify changes. This means that altering the Input Map between one AL execution

⁷ Also known as performing the Iterator operation `selectEntries()`, where the data set for iteration is selected.

⁸ TDI stores Iterator state values in the System Store like any other persistent objects, so you can access this information through the `system.getPersistentObject()`, `system.setPersistentObject()` and `system.deletePersistentObject()` methods, using the **Iterator State Store** value as the *key* parameter.

⁹ As you will see in a bit, not all Change Detection Connectors return the same level of detail when it comes to delta information. Systems like Microsoft ActiveDirectory and Domino return only change status for the whole Entry, and not for individual Attributes and *Values*. The Delta Engine can be used here to make up for limited delta notification capabilities.

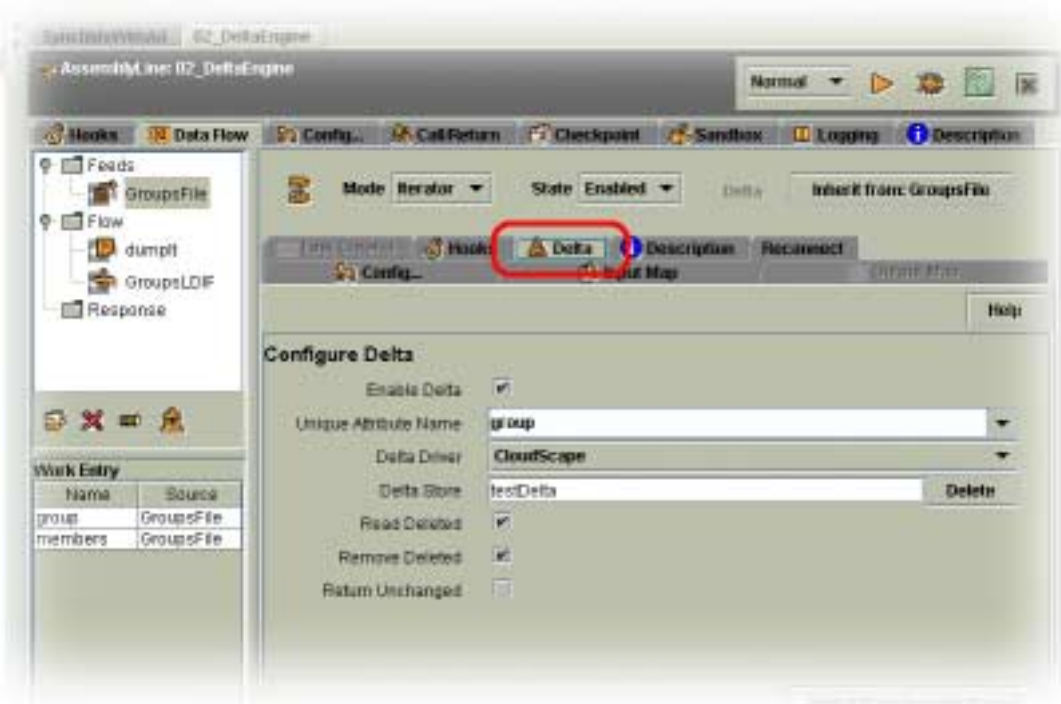
and the next will affect Delta operations. Best practice is to delete the Delta table for an Iterator if the Input Map is changed.

The Delta Engine works in two passes.

1. First, as the Iterator reads through the input data, each Entry is compared with its corresponding snapshot (if one is found). Based on snapshot absence or comparison, the Delta Engine returns this data tagged with the relevant operation codes: *add*, *modify* or *unchanged*.
2. Once End-of-Data is reached by the Iterator, the Delta Engine makes a second pass through the Delta table looking for those snapshots not accessed during the first pass. These are then returned as *deleted* Entries.

2.2.1 Setting Up the Delta Engine

You set up Delta Engine parameters in the Delta tab of an Iterator mode Connector.



This tab has the following settings:

Enable Delta

This checkbox must be selected in order to turn on the Delta Engine and give you access to the other parameter settings.

Unique Attribute Name

The Input Map Attribute that uniquely identifies each entry in the snapshot database. Note that you can use an Advanced Mapped Attribute to combine multiple Attributes to create a unique value.

Delta Driver	This is a backwards-compatibility option which allows you to access the deprecated BTree Delta store ¹⁰ used in older solutions. Only the “System Store” option can be set in TDI 6.0.
Delta Store	Name of the System Store table that will be dedicated to this Iterator’s Delta snapshot data. The Delete button next to this parameter will drop the snapshot table, so the next AL run will create a fresh baseline for delta detection.
Read Deleted	Must be selected for the Delta Engine to return deleted Entries.
Remove Deleted	This flag tells the Delta Engine to remove snapshots for deleted Entries as they are returned. Otherwise they will be reported as deleted on the next run of the AL as well.
Return Unchanged	If this flag is set then Entries that have not been added, modified or deleted will be returned. These are tagged with the operation code <i>unchanged</i> .
Commit	This setting controls <i>when</i> the snapshot is saved to the Snapshot Database. It’s recommended to set this parameter to “ One end of AL cycle ”.

Although Delta tables can be accessed with both the JDBC Connector and the Persistent Entry Store (PES) Connector, it is unadvisable to make changes without a deep understanding of how these tables are structured and handled by the Delta Engine (in other words, *do so at your own risk*).

2.2.2 Performance and Risk

Although the Delta Engine will work with all types of input data sources (as opposed to the Change Detection Connectors) there are issues associated with the Delta Engine that you need to be conscious of:

1. The Delta Engine maintains a *shadow copy* of your input data source. If you have a large input data set, then the snapshot database will also be big.
2. Running with the Delta Engine will impact the performance of your AssemblyLine.
3. As opposed to the Changelog Connectors, the Delta Engine is based on iterating through your entire input dataset. So although you spare target systems from unnecessary updates, your solution will read extensively from your input source.
4. Should the snapshot database get out-of-synch (for example, if updates are performed directly to the target systems) then this will not be automatically detected by the Delta Engine. Instead, you may need to delete the snapshot database in order to build a new baseline¹¹.

¹⁰ Note that for 6.0 and earlier versions, the “**Cloudscape**” option in this drop-down indicates that the System Store will be used, which can easily be configured to use another compliant RDBMS, like DB2, Oracle or Microsoft SQL Server.

¹¹ The best course of action is to build an AssemblyLine for this purpose. It should be one that not only initializes the snapshot database, but also updates targets with necessary changes: a “Baseline” AL.

Even so, the Delta Engine gives you a fast and reliable way of detecting changes where you otherwise couldn't.

2.3 LDIF Parser with incremental LDIF

The LDIF Parser can be used to both write “incremental” LDIF output based on Delta operation code tagging in the Work Entry and its Attributes, as well as to read these files and return the corresponding Delta Entries. Of course, it can also be used with “full” LDIF files, although the Parser will do no tagging of delta operation codes in this case.

The difference between these two types (*incremental* and *full*) is that incremental LDIF files contain only information about changed entries:

```
version: 1

dn: All Employees
changetype: modify
add: members
members: abnevanm408
-

dn: Coffee Drinkers
changetype: delete
-
```

The above example has two entries: the first one (with the `dn` value “All Employees”) signals that the value “abnevanm408” is added to the `members` Attribute. The second entry indicates deletion of the “Coffee Drinkers” entry itself.

Those LDAP Changelog Connectors (like the IBMDirectoryServer Changelog Connector) perform Attribute and Value tagging by using this Parser on the incremental LDIF information kept in the changelog.

Maybe it's time to get some more coffee or stretch your back. Then it's on to Delta Tagging.

Technical note follows:

If you write your Sync AL correctly, then you can easily call it using the AssemblyLine Function (AL FC) from a second “Set Delta Baseline” AL. Note that if you call an AL with the AL FC using manual/cycle mode, it disables the called AL's *Feeds* section. Instead, the Attributes you pass into the called AL (via the AL FC's Output Map) are fed directly into the called AL's *Flow*. So, by setting delta operation codes manually in the calling AL, you can “fool” your called Sync AL Flow into handling the data you pass it just like it does the Delta Entries returned by its own Iterator. The end result is that you reuse the logic in the *Flow* section of your Sync AL to handle updating targets for your “Baseline” AL.

This will also require defensive configuration of Delete mode Connectors to deal with data that is already removed from targets. Additionally, leveraging the Compute Changes feature of Update mode Connectors will ensure that only necessary write operations are performed (see section 4.1.3 *Compute Changes* on page 21 for more details on Compute Changes).

3 Delta Tagging

Delta Tagging is the process of marking retrieved data with delta operation codes, and is typically done during Delta Detection.

Each operation code describes how the unit of information it is attached to has changed in the source system. As mentioned previously, Delta Tagging is done by all Change Detection features. As you will see in section 4 *Delta Application*, these delta operation codes are used by TDI to correctly apply changes to target systems. Before we take a look at how tagging is done by TDI components, we will dig into what the handful of delta operation codes mean.

3.1 Delta Operation Codes

Not that you will necessary need to know much about this – it is probably enough to understand this concept in more general terms. So at least skim this section. If you want to know more, check out section 5 *Working with Delta Operation Codes* on page 23. Knowing a little about the delta operation codes and their meanings can go a long way when you start debugging your Sync AL.

As mentioned earlier, an Entry object carries a *delta operation code*. During Delta Tagging, this code is set to a value corresponding to the type of change detected: add, delete, modify or unchanged. These operation codes are not visible in the Config Editor, but you can access them from JavaScript through a couple of handy function calls found in the Entry and Attribute objects¹².

An Attribute also has an operation code. This code is analogous to that found in Entry objects, and indicates whether an Attribute has been added, replaced, deleted, modified or is unchanged. Furthermore, Attributes keep track of operation codes for the *Values* they contain. As with the Entry object, an Attribute offers functions for reading and setting its own operation code, as well as those of its *Values*.

In addition to tagging done automatically by TDI, there are situations where you may want set these values yourself; For example, when you are getting your change information from some other source than one of the Delta Detection mechanisms (like receiving SOAP over IP, or reading messages from MQ). More likely however is that you will want to branch your AssemblyLine flow logic based on this code (as shown in the section on [Delta Application](#)).

If you want more technical insight into delta operation codes, as well as details into how each Delta Detection feature assigns them, see section 5 *Working with Delta Operation Codes* on page 23.

3.2 Delta Mechanisms and Tagging Levels

Below is a list of the various Delta Mechanisms along with details on what they return.

¹² For example, in the Debugger you could enter the following script into the Evaluate field:

```
task.logmsg( work.toDeltaString() )
```

This displays an exploded view of the Entry object, including all delta operation codes. You can also use this in script code used in your AL as well.

Table 1 - Change Detection Mechanisms

LDIF Parser	Tags Entries and Attributes/values.
Delta Engine	<p>Tags Entries and Attributes/values.</p> <p>Note that the Delta Engine will only report a single change per entry. For example, if this data has been added, modified and then deleted since the last iteration, only a single “delete” tagged Entry is returned.</p>
IBMDirectoryServer Changelog Connector	<p>Tags Entries and Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “changeType” with the value “add”, “modify” or “delete” (or “rename” for a rdn/\$dn change).</p>
Netscape/iPlanet Changelog Connector	<p>Tags Entries and Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “changeType” with the value “add”, “modify” or “delete” (or “rename” for a rdn/\$dn change).</p>

¹³ This information is included for the sake of completeness. Although you may see these Attributes used in pre-6.0 Fixpack3 Configs to differentiate between change types, it is recommended that you use the delta operation code of the Work Entry to do this in your own solution.

<p>Active Directory Changelog (v.2) Connector</p>	<p>Tags Entries, but not Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “changeType” with the value “update” (for both <i>add</i> and <i>modify</i>) or “delete”.</p> <p>Note that this Connector will only report a single change per entry. For example, if this data has been added, modified and then deleted since the last iteration, only a single “delete” tagged Entry is returned.</p> <p>Furthermore, only a <i>stub</i> is retained for deleted entries. As a result, deleted entries returned by this Connector do not contain any attributes except the following:</p> <ul style="list-style-type: none"> • USN • targetdn • changeType <p>You must use USN to reference corresponding entries in synchronization targets.</p>
<p>Domino Change Detection Connector</p>	<p>Tags Entries, but not Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “\$\$ChangeType” with the value “add”, “modify” or “delete”.</p> <p>Note that this Connector will only report a single change per entry. For example, if this data has been added, modified and then deleted since the last iteration, only a single “delete” tagged Entry is returned.</p> <p>Furthermore, only a <i>stub</i> is retained for deleted entries. As a result, deleted entries returned by this Connector do not contain any attributes except the following:</p> <ul style="list-style-type: none"> • \$\$UNID • \$\$NoteID • \$\$ChangeType <p>You must use \$\$UNID to reference corresponding entries in synchronization targets.</p>
<p>RDBMS Changelog Connector</p>	<p>Tags Entries, but not Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “IBMSNAP_OPERATION” with values “I”</p>

	<p>for Inserted (<i>add</i>), “U” for Updated (<i>modify</i>) or “D” for Deleted (<i>delete</i>).</p> <p>Note that for each Entry returned, control information (counters, operation, time/date) is moved into Entry <i>Properties</i>¹⁴, while data values are stored as Attributes.</p>
<p>Exchange Changelog Connector</p>	<p>Tags Entries, but not Attributes/values.</p> <p>¹³Returns type of change in an Attribute called “changeType” with the value “update” (for both <i>add</i> and <i>modify</i>) or “delete”.</p> <p>Note that this Connector will only report a single change per entry. For example, if this data has been added, modified and then deleted since the last iteration, only a single “delete” tagged Entry is returned.</p> <p>Furthermore, only a <i>stub</i> is retained for deleted entries. As a result, deleted entries returned by this Connector do not contain any attributes except the following:</p> <ul style="list-style-type: none"> • USN • targetdn • changeType <p>You must use USN to reference corresponding entries in synchronization targets.</p>

¹⁴ Properties are accessible through the Entry methods `getProperty()` and `setProperty()`.

4 Delta Application

Regardless of how you get hold of change information, the ultimate goal is to apply the delta to one or multiple targets.

Writing to data sources is done with the appropriate Connector in one of the output modes: AddOnly, Update, Delete or Delta. When using AddOnly, Update or Delete modes, it is up to you to set up the AssemblyLine flow logic so that the correct data operation is performed depending on the type of changes reflected in the Delta Entry. Delta mode on the other hand does this for you. However, only the LDAP Connector supports this mode. So if you are synchronizing to an LDAP directory, you can skip straight to sub-section 4.1.2 *Delta Mode* on page 20.

4.1.1 Manual Delta Application

Prior to the advent of Delta mode in version 6.0, the methods outlined in this section were all you had for applying a delta to target systems. If you can use Delta mode, then do so. It will make your ALs smaller and simpler, leaving the grunt work to TDI.

Without the option of Delta mode, the AssemblyLine must be set up to differentiate between add/modify and delete change types. If the Connector you plan to use for output supports Update mode, then this will deal with both *add* and *modify* changes for you. Deleting data is the job of Delete mode.

Configuring your AssemblyLine to handle *add*, *modify* and *delete* operation codes can be done in a number of ways. Note that the first method outlined below makes use of AL Branches, and is best practice for building legible, maintainable solutions. The other two approaches are included here for the sake of completeness, and to help you decipher Configs built with earlier versions.

Branches

Your AssemblyLine can either use a Switch based on the Work Entry operation code, or two Branches: An *IF* Branch followed by an *ELSE IF* Branch. The *IF* Branch needs a scripted Condition that is then set to the following:

```
ret.value = ( work.getOperation() == "delete" );
```

This must be done as a scripted Condition since it is recommended to use the delta operation code instead of a change-type Attribute (which varies between systems).

If for some reason you can't use the delta operation code, then you can set up a simple Condition to check the change-type Attribute value instead.

```
changeType EQUALS delete
```

Under this Branch you have your Delete mode Connector to remove entries from the connected system.

Just after the "*IF*" Branch you add the "*ELSE IF*" Branch to handle your Update mode Connector to deal with "add" and

“modify” changes.

Before Execute Hook Just to repeat myself again (and again): this is not recommended practice in 6.0; use Branches instead. However, since you may have to read and maintain an older Config, read on.

The “Before Execute” Hook is present in every Connector, regardless of mode. If enabled, the Hook script is executed on each AL cycle before any other action is taken by this component.

So one pre-6.0 approach is using the **Before Execute** Hook to conditionally *ignore* the current Entry if the change type is inappropriate for the mode of this Connector:

```
if (!work.getString("changeType").equals("delete"))
    system.ignoreEntry();
```

The above example script would be in the **Before Execute** Hook of a Delete mode Connector, and would pass control to the next component if the Attribute called `changeType` did not have the value “delete”.

Script Component (SC) Another common pre-6.0 tactic was to set up a Connector in Passive State, with the correct Output Map and Link Criteria.

Passive State ensures that the Connector is initialized at AL startup and closed when the AssemblyLine terminates, but not executed automatically during AL cycling. Instead, the Connector is manually called from a Script Component in the AL:

```
if (work.getString("changeType").equals("delete"))
    db2Connector.deleteEntry(work)
else
    db2Connector.update(work);
```

This snippet drives a Connector called “db2Connector”¹⁵, using either the `deleteEntry()` or `update()` method as needed.

Note that this can be done from any block of script, like a Hook or a scripted Connector. However, placing this kind of flow logic in an SC makes your AssemblyLine more legible. Even so, as stated before, use Branches. ‘Nuff said.

¹⁵ All AssemblyLine Components are automatically registered as script variables, which is why it is important to name them as you would a variable.

4.1.2 Delta Mode

Delta Mode not only combines Update and Delete mode handling (including offering many of the same Hooks), it will also perform *incremental modify operations* to the connected system. This means only writing the specific values that have changed.

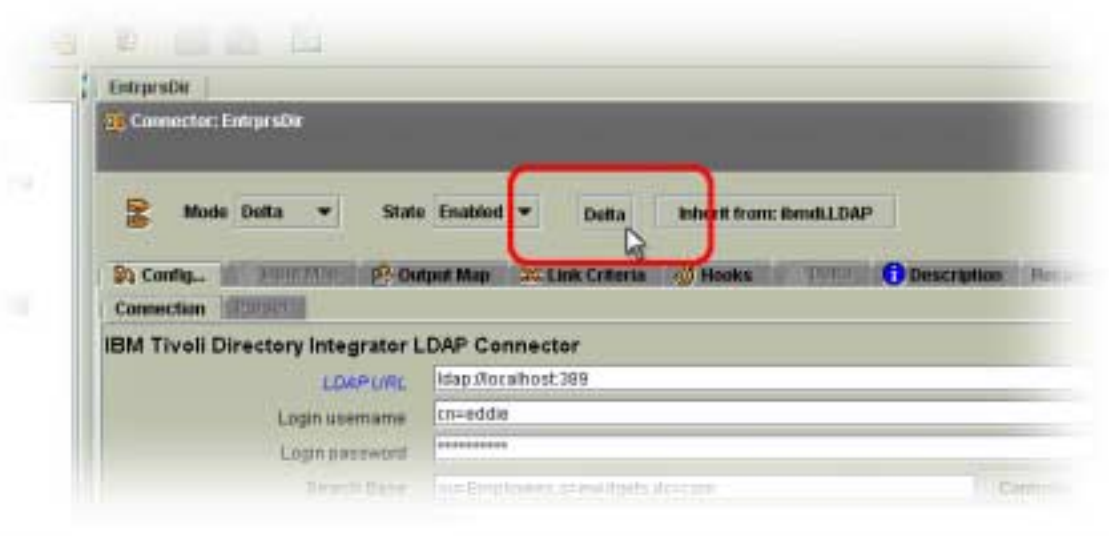
Incremental modifies represent a significant performance improvement since load on the data source and network is minimized, and particularly when working with changes to group membership or other massively multi-valued Attributes.

In order for Delta Mode to work, the Connector must receive a Delta Entry (i.e. an Entry tagged with a valid delta operation code¹⁶). This is the only mode that requires (and uses) these delta tags, and highlights a basic difference in how Update and Delta modes function.

Update mode differentiates between *add* and *modify* operations by first performing a lookup using the Connector's Link Criteria. If a match is found then the Connector modifies this entry. Should the lookup fail to find matching data, a new entry is added. In other words, changes are applied based on the current state of the target system.

Delta mode on the other hand “assumes”¹⁷ that the source and target were previously in sync, and that any differences are encapsulated in the Entry object. One side-effect is that delta information must be applied in the same order as it occurred in the source.

Deciding which operation codes Delta mode should handle, as well as how it deals with untagged (unwanted) entries, is configured by pressing the **Delta** button¹⁸ at the top of the Connector Details panel.



¹⁶ In fact, all Entries have some sort of delta operation tag set. However, unless the Entry is coming from some Delta Detection mechanism, it will carry the default tag of *generic*. This non-delta tag simply indicates that the Entry carries no information about changes – merely data.

¹⁷ This assumption of source and targets being in sync prior to Delta mode operation may also limit the applicability of Delta mode to a specific scenario – e.g. when changes can occur to targets outside Delta Handling in TDI.

¹⁸ This button only appears to be a button when you move the mouse over it.

This brings up the Permitted Delta Operations dialog.



If the checkbox at the bottom of the panel shown above is unselected, then the Connector will simply ignore Entries tagged as *generic*, passing control to the next component. Otherwise, it results in an error; i.e. an exception is thrown and must be handled in an Error Hook or the AssemblyLine will stop.

In addition to simplifying data synchronization AssemblyLines, Delta Mode also makes the most effective use of your LDAP server when performing delta operations. Instead of first retrieving the entire entry to be modified, applying changes and then writing all this data back to the target (like Update mode does), only the changes prescribed by the Attribute and value operation codes are propagated.

4.1.3 Compute Changes

No treatise on data synchronization with TDI would be complete without a note on the Compute Changes option.

This flag is available for Update mode, and instructs TDI to compare the Attributes in the Output Map with the corresponding ones read into the *current*¹⁹ Entry object by the initial lookup operation. If no differences are detected, then the modify operation is not carried out.



Using this option is an easy way to avoid triggering the replication features in your target system due to unnecessary changes.

¹⁹ Any lookup operation (as performed in Lookup, Update or Delete Modes) reads data into both *conn* and an additional Entry object called *current*. The rationale behind this comes from the needs of Compute Changes: After the lookup, data is read into *conn* and *current*. Then the Hook Flow of the Connector gets to the Output Map, where the data to write is copied from *work* into *conn*. Then Compute Changes behavior kicks in (if turned on) and each Attribute value mapped into *conn* is compared with data in *current* in order to decide if anything is different. If no differences are detected then the *modify* operation is skipped; otherwise the *conn* Entry is written.

WARNING: this next section may require some level of technical geekness to enjoy.
Read on if you gotta know more.

5 Working with Delta Operation Codes

The operation code for an Entry can be accessed directly via the `getOperation()` and `setOperation()` methods of the Entry object. These function calls use String values for the various change types²⁰. Here is the list over valid Entry operation codes:

Table 2 - Entry-level Delta Operation Codes

generic	The default operation code for Entry objects returned by any other means than from one of the Delta Detection mechanisms. This code means that there is no delta tagging available. An Entry with this operation code is considered <i>untagged</i> and not a Delta Entry.
add	Signals that the Entry object is new and should be added to the target(s).
modify	The entry has been modified. This operation code also implies that there may be more delta tags available for contained Attributes, and possibly even the Attribute values (discussed more in detail below).
delete	Indicates that the Entry was deleted from the source.
unchanged	Unlike <i>generic</i> , this is an actual delta tag that is used for unmodified Entries. Only some Delta Detection mechanisms, like the Delta Engine, give you the option to return Entries with the <i>unchanged</i> code.

Note that these codes are case-insensitive when you set them – so “ADD” is equivalent to “add”. If an invalid code is used in the `setOperation()` method (for example, null or an unrecognized value like “whatever”) then the Entry is tagged as *generic*.

Although `setOperation()` does not care about case, `getOperation()` calls return lowercase values as shown in this example which writes a log message if the Work Entry has the *delete* operation code tag:

```
if (work.getOperation() == "delete")
    task.logmsg( "Work Entry is tagged for deletion." );
```

The `getOperation()` call can be used in Branches in order to differentiate between Deletes and Adds/Modifies (as mentioned in the section on [Delta Application](#)). You do this by using the scripted Condition at the bottom of the Branch dialog. Here you would use code like this:

²⁰ Although only the first character of any code is necessary to make the call, it is good practice to spell the operation code out completely when using it. This will improve the legibility of your Config. Note also that these values are localized and will always use the English codes shown here.

```
ret.value = work.getOperation() != "delete";
```

This would be the equivalent of the simple Condition: `changeType NOT EQUALS "delete"`.

5.1 Attribute and Value Operation Codes

Attributes have a similar set of operation codes. The meaning of these codes is listed in the table below:

Table 3 – Attribute-level Delta Operation Codes

replace	The default operation code for Attributes, this tag means that the Attribute should be written as-is, with all values to the target(s) – i.e. replacing whatever is already there.
add	Signals that the Attribute is new and should be added to the entry in target(s).
modify	The Attribute has been modified. This operation code also implies that there may be more delta tags available for the <i>values</i> of this Attribute.
delete	Indicates that the Attribute was deleted from the entry in the source.
unchanged	Signals that this Attribute is unchanged.

These codes are read and written in JavaScript code by using the Attribute methods `setOperation()` and `getOperation()`. Setting an invalid code will result in default tagging (*replace*).

For example, the following script first gets the “FullName” Attribute from the Work Entry and then sets the operation code to *modify*²¹:

```
var fullName = work.getAttribute("FullName");
fullName.setOperation("modify");
```

²¹ It is important to understand that when you *get* an Attribute from an Entry (as with the `getAttribute()` method) you actually get a *reference* to this Attribute, not a copy. So any changes you make directly affect the actual Attribute itself that is stored in the Entry.

Drilling down to the next and final level, Attribute *values* can be tagged as either *unchanged*, *add* or *delete*. The methods for working with Value-level delta tags are also found in the Attribute object – `setValueOperation()` and `getValueOperation()` – both of which require an index parameter that indicates which value to apply the tag to.

```
fullName.setValueOperation(0, "delete");
```

This snippet sets the operation code for first value²² of the `fullName` Attribute to *delete*.

This is a lot of technical information to digest, but keep in mind that TDI will take care of most operation code tagging and interpretation for you. However, you should at least be familiar with the details of how this is done in order to handle those situations where the built-in features fall short of your requirements.

Now that we've looked at the various levels of operation code tagging, the next logical step is to see how these relate to each other.

5.2 Tagging Rules for Delta Operation Codes

Even though an Entry object, its Attributes and their *Values* can all carry different operation codes, these tag values work together in concert to describe how data has been changed. To do so, they must all follow the TDI operation code tagging rules:

- If an Entry is tagged as *generic*, *add*, *delete* or *unchanged*, then its Attributes and their Values will not be tagged with significant operation codes. Although these should be set to default values, they will regardless be ignored by Delta Application logic.
- If an Entry carries the *modify* tag, then its Attributes *may* be tagged as *replace*, *add*, *delete* or *modify*.

Furthermore:

- If an Attribute has an operation code of *add*, *delete*, *replace* or *unchanged*, then any tags set for its values will be ignored – in other words, all values will be handled by Delta Application logic as indicated by the Attribute's operation code.
- If an Attribute has an operation code of *modify*, then *at least one* Value *must* be tagged as either *add* or *delete*

As you can see, the *modify* tag has special significance at both the Entry and Attribute level in that it implies the presence of delta operation codes for objects it contains. However, TDI does not enforce these rules when you tag data yourself. Delta Application logic provided by TDI may ignore incorrect operation codes, or even throw an error. More on this in section 4 *Delta Application* starting on page 18.

²² Indexes in Java, as with many programming languages, start with zero (0). So if an Attribute has four values, these are accessed as indexes 0, 1, 2 and 3.

5.3 Debugging Delta Operation Codes

If you've ever used the `task.dumpEntry()` method, then you've probably seen operation codes without knowing it:

```
13:32:34 *** Begin Entry Dump
13:32:34      Operation: generic
13:32:34      [Attributes]
13:32:34          members (replace): 'aglessan150'    'alanbraul06'
13:32:34          group (replace):  'Coffee Drinkers'
13:32:34 *** End Entry Dump
```

The above dump shows that Entry itself tagged as *generic*²³, indicating that it did not originate from Delta Detection. Since the Entry bucket is tagged with the *generic* delta operation code, it is not surprising that the Attributes shown – `members` and `group` – have default delta tags (which for Attributes is *replace*).

If we dump an Entry received from Delta Detection (in this example, read from an LDIF Parser) then you can see that the format has not changed; just the codes shown:

```
13:44:21 *** Begin Entry Dump
13:44:21      Operation: modify
13:44:21      [Attributes]
13:44:21          members:    'abnevanm408'
13:44:21          group (replace): 'Coffee Drinkers'
13:44:21 *** End Entry Dump
```

The Entry shown here carries the *modify* tag. But while the `group` Attribute still has the default *replace* code, the tag for `members` is not displayed by `dumpEntry()`. That is because the `members` Attribute has the *modify* tag, which in turn means that the values it contains are also tagged. However, the `dumpEntry()` function is designed to display the contents of an Entry object in condensed format, not its full delta information.

In order to display all operation codes you can either query these values using the methods listed in the previous section, or you can get the Entry object to represent itself as a Java String that includes all delta info. This is done with the Entry's `toDeltaString()` method. As an example, the following snippet will log the delta representation for the Work Entry:

```
task.logmsg( work.toDeltaString() );
```

²³ Functions like `task.dumpEntry()` use the more legible String variants of the delta operation codes. For example, `OP_MOD` is displayed as “modify”.

The resulting output looks like this (after the log timestamp is removed):

```

modify {
  members {
    type: modify
    count: 3
    values [
      add: abnevanm408
      unchanged: abdaburr393
      unchanged: alanbraul06
    ]
  }
  group {
    type: unchanged
    count: 1
    values [
      unchanged: Access To The Executive Washroom
    ]
  }
}

```

The topmost `modify` in the above listing is the operation code of the Entry itself. Attributes contained in this Entry are listed inside a set of curly braces `{}`.

For each of the Attributes shown here (`members` and `group`), further details are displayed inside additional curly braces. Looking at the `members` Attribute, the first item listed shows the operation code of the Attribute (displayed as “`type: modify`” above). Next comes the number of values (`count: 3`) followed by the values themselves, each with its own operation code.

A shorthand description of the above listing would be “*add the value ‘abnevanm408’ to the ‘members’ Attribute of this Entry*”.

5.4 Manual Delta Code Tagging

Although all Change Detection components and features return information on how data is changed, tagging of Attributes and Values is not done by all of them. So even if there is not a suitable Changelog Connector for your needs, you can leverage the Delta Engine for this purpose.

However, if the target system provides information about change events (for example, as Attribute values, or via API calls) then you can easily set the operation codes yourself.

As an example, consider an `AssemblyLine` reading from a CSV file. Each line in the file is the record of some change, and one of the fields is called “`changeType`” and can have a value of “`delete`”, “`add`” or “`modify`”. Armed with this Attribute value, we can use the following script to tag the Work Entry accordingly.²⁴:

```
work.setOperation( work.getString( "changeType" ) );
```

²⁴ Although you can put this code in a Connector Hook (like `After GetNext`), a better choice is to drop this snippet in a Script Component that appears in the AL just after the Iterator. By naming this Script Component descriptively, for example “`PerformDeltaTagging`”, your `AssemblyLine` becomes easier to read and maintain.

The manually tagged Work Entry is now ready to be passed to Delta Application.

6 Conclusion

So there you have it, or at least a piece of it. As is the nature of development tools, there are multiple approaches to building synchronization solutions. In addition to the topics covered here, the adventurous user can extend TDI's integration reach by creating new components (in Java or JavaScript) and leveraging vendor-specific functionality available in your systems. Or making calls to these APIs directly from Script code in your AssemblyLines.

Whether you use the Delta Handling features in TDI 6.0 for Delta Detection/Tagging, Delta Application or both, they provide building blocks for laying the foundation of your solution faster.

7 References

1. Getting Started. Part of the official TDI documentation. See <http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.IDI.doc/gettingstarted.htm>