Synchronizing Data with TDI

by Eddie Hartman

How To Use TDI to Synchronize Data

The basic goal of data synchronization is to detect changes in one data source and then propagating these to one or more targets. Capturing changes in data sources is not as easy as you might think. Some systems have detailed change logs, but most do not. How do you know if a single value from a multi-value field has been deleted? Fortunately, TDI provides the framework to let you deal with this at a comfortable high level. However, a certain amount of understanding is required to take full advantage of TDI's capabilities.

This document outlines the features in TDI designed for building data synchronization solutions. It also provides insight into how to use them. However, you must already have a some experience with TDI; At the very least completed the Getting Started tutorial [1].

1.1 Introduction

TDI's Delta Handling features are designed to facilitate efficient data synchronization. This means passing on only changes – that is, if it's possible to detect changes in the source system. At the other end of the data synch pipe, changes should be made to targets only as needed to minimize system and network traffic, and to avoid triggering unnecessary replication.

In summary, Delta Handling can be thought of as three distinct activities:

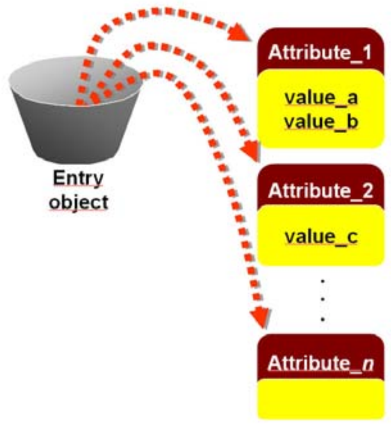
Delta Detection	This is two operations: 1) discovering that a change has occurred in a data source and 2) retrieving the information needed to propagate the change. This is discussed in section *2 Delta Detection* starting on page 7.
Delta Tagging	*Tagging* the retrieved data with this delta information. This is done by assigning (tagging) *delta operation codes* to the data, describing the type of change: e.g. *add modify*, *delete,* and so on. These are also referred to as "operation codes" and "delta tags" in TDI literature. More on this subject in section *3 Delta Tagging*, page 13.
Delta Application	Using these operation codes to propagate the changes to other stores/systems as efficiently as possible. Delta Application is detailed in section *4 Delta Application.*

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 saw above, the remainder of this document is divided into three sections; one for each of the aspects of Delta Handling.

But first, this next section on the TDI Entry data model.

1.2 The Entry Object

In order to master delta handling in TDI, you must first understand 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 (or none at all).



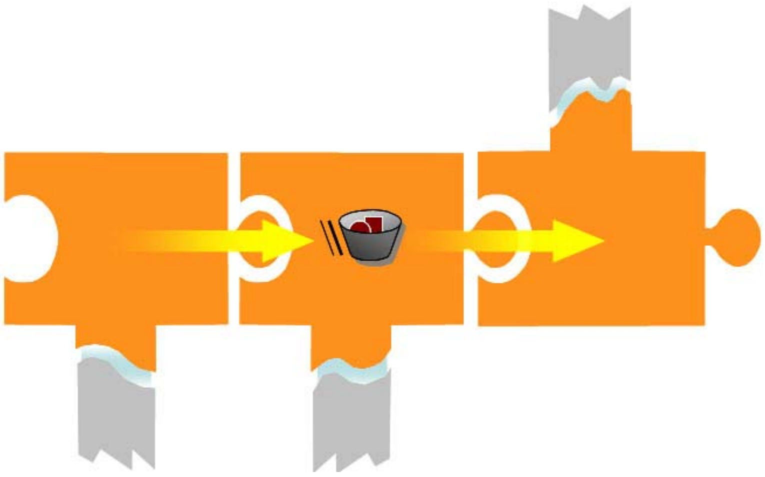
Attributes are also bucket-like objects in TDI. Each Attribute can contain zero or more *values,* these being the actual data values that are read from (and written to) connected systems. These Attribute *values* are Java objects as well – like strings, integers and timestamps[1] – and a single Attribute 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 be of the same type.

Although the Entry-Attribute-value paradigm matches nicely to the concept of 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.

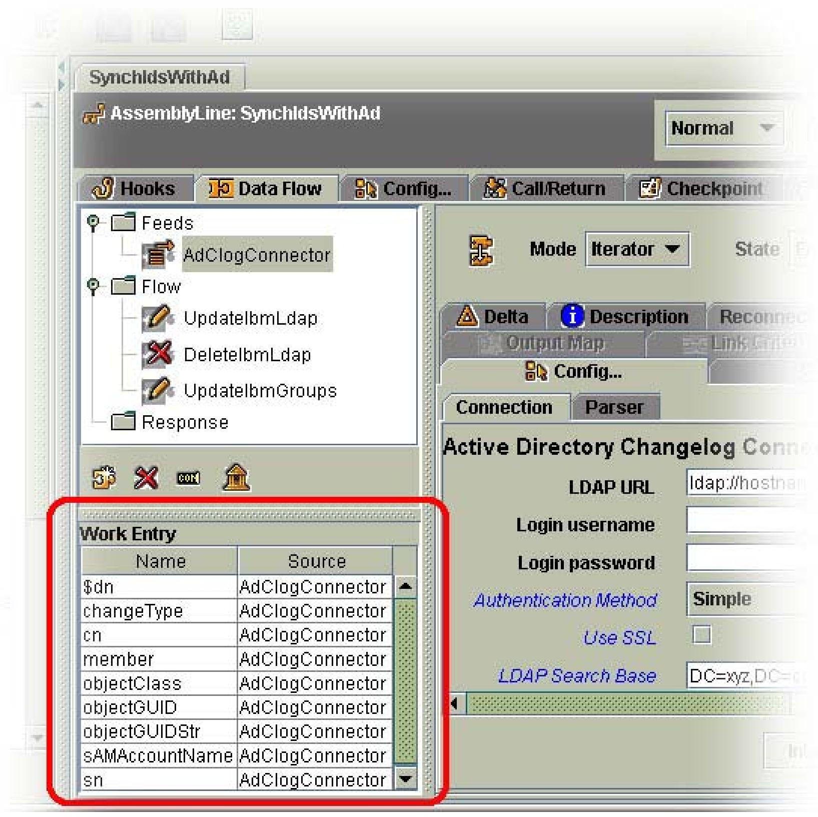
There are a handful of Entry objects that are created and maintained by TDI. The most visible instance is called the *Work Entry*, and it serves as the main data carrier in an AssemblyLine (AL). This is the bucket used to transport data down an AssemblyLine, passed from one component to the next during 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 AssemblyLine[3].



Mode: Iterator

State

LDAP URL: ldap://hostna

Login username:

Login password:

Authentication Method: Simple

Use SSL: ☐

LDAP Search Base: DC=xyz,DC=

Work Entry

Name	Source
$dn	AdClogConnector
changeType	AdClogConnector
cn	AdClogConnector
member	AdClogConnector
objectClass	AdClogConnector
objectGUID	AdClogConnector
objectGUIDStr	AdClogConnector
sAMAccountName	AdClogConnector
sn	AdClogConnector

So in summary, an Entry holds Attributes which in turn contains data *values*. Operation codes that describe the delta status of the *bucket* itself are kept at the Entry level, while those that apply to its Attributes and associated *values* are maintained by the Attribute objects themselves.

[3] Note that only 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.

In addition to holding Attributes, an Entry object also keeps track of its operation code. Similarly, Attributes maintain an operation code for itself, as well as one for each value it contains. More detail on this is found in section *3 Delta Tagging* on page 13.

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

2 Delta Detection

As stated above, Delta Detection is the discovery and retrieval of changes. The change information is then used to Delta Tag retrieved data with 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 handful that have changed.

Delta Detection is automatically handled by the following TDI features:

- **Change Detection Connectors**, like the IBM Directory Server Changelog Connector, RDBMS Changelog Connector and Domino Change Detection Connector.
- The **Delta Engine**: Any Connector in Iterator mode has a Delta tab where the System Store[4] is used to keep snapshots of data which are used to detect changes made between subsequent AL runs.
- The **LDIF Parser** sets operation codes in the returned Entry object using the delta info stored in *incremental* LDIF[5] files. Unlike the two preceding items above, the LDIF Parser does not detect changes. Instead, it interprets the delta codes found in an incremental LDIF file, which itself only contains information about changes.

The full list of Delta Detection features can be found in

Table 3 - Change Detection Mechanisms and Tagging Levels on page 18. Regardless of the mechanism used, the end result is an Entry *bucket* with delta operation codes set.

2.1 Change Detection Connectors

A Change Detection Connector leverages features available in 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 offer a **timeout** parameter to control how long the Connector will wait for new changes to appear.

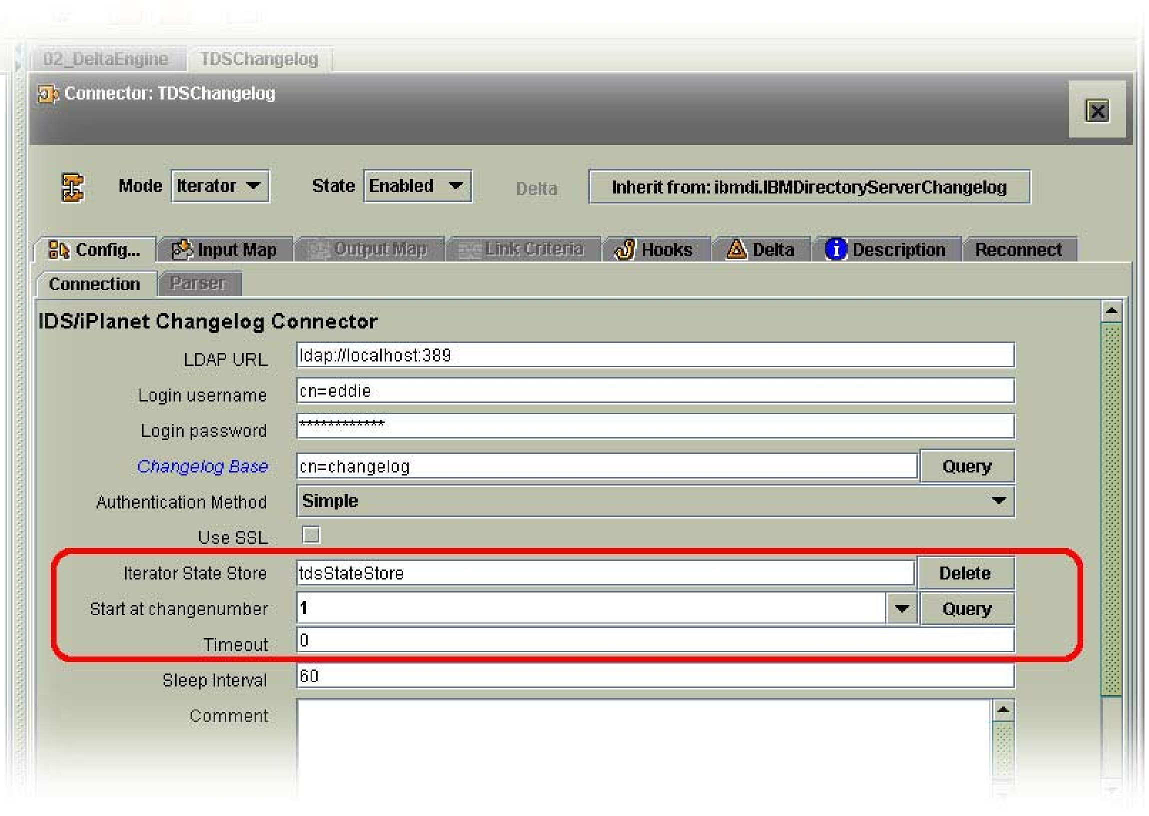
Where supported, a Change Detection Connector registers with the data source for change notifications, receiving a signal whenever a change is made. Other Connectors have to poll the connected system periodically looking for new changes. Those that rely on polling also provide a **Sleep interval** option to define how often polling occurs.

[4] 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 configured to use DB2, Oracle, Microsoft SQL Server, or any other compatible RDBMS.

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

2.1.1 Iterator State

Another important feature is that all Change Detection Connectors provide an **Iterator State Store** parameter for keeping track of the next change to be processed, even between runs of the AssemblyLine.



Mode: Iterator
State: Enabled
Inherit from: ibmdi.IBMDirectoryServerChangelog

IDS/iPlanet Changelog Connector

LDAP URL	ldap://localhost:389
Login username	cn=eddie
Login password	*************
Changelog Base	cn=changelog
Authentication Method	Simple
Use SSL	☐
Iterator State Store	tdsStateStore
Start at changenumber	1
Timeout	0
Sleep Interval	60
Comment	

This feature uses the System Store 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, so that if you have multiple ALs that use Change Detection Connectors, they will 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. For example, in the IBMDirectoryServer Changelog Connector, there is a **Start at changenumber** parameter where you can enter the changelog number where processing is to start. This parameter can be set to either a specific value, to the first change (i.e. changenumber = 1), or to "**EOD**" (End of Data). The EOD setting places the cursor at the end of the change list in order to only process new deltas.

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 performs its selectEntries() operation – for example, when the Iterator is initialized at AL startup, or in a Loop component. The same will occur if there is no value stored for the specified Iterator State Store identifier.

So, the very first time you run the AL with the Change Detection Connector there will be no Iterator State Store value yet, so the **Start at...** parameter will be used. On subsequent executions, the **Start at...** setting will be ignored and the Iterator State Store value applied instead.

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

This is of particularly interest given the fact that the Iterator State Store value is saved *just after it is read*, and not at the end of each AssemblyLine cycle[6]. As a result, if the AssemblyLine fails before changes are applied to all targets then the next time it is started again, the Iterator state key will cause the Connector to start iterating *after* this failed change entry. In other words, the change entry being handled when the AssemblyLine stopped is skipped on the next run.

You can safeguard against this by managing your own Iterator state values. As an example the following code snippets in an AssemblyLine will help safeguard against “losing” change entries as described above. The first two blocks of code shown here are intended for the Hooks of the Change Detection Connector itself, while the last one is added as a Script Component at the end of the AssemblyLine.

Prolog – Before Initialize Hook of the Change Det. Connector	`// Apply my own Iterator state store value.`
	`//`
	`var myStateKey = system.getPersistentObject(“myStateKey”);`
	
	`if (myStateKey != null)      // Has this been set before?`
	`   system.setPersistentObject(“itrStateKey”, myStateKey);`

Before GetNext Hook of the Change Det. Connector	`// Save the current setting for the next change`
	`// in the myStateKey variable. The Iterator State Store`
	`// param is set to “itrStateKey” in my Connector.`
	`//`
	`// Note that this persistent object will not have been`
	`// set the first time the AL is run.`
	`//`
	`myStateKey = system.getPersistentObject(“itrStateKey”);`
	
	`if (myStateKey == null)       // First time AL is run?`
	`   myStateKey = firstChange;  // Specific for type of`
	`                              //   Connector used.`

Script Component at end of AL	`// Persist the saved state key when the AL cycle reaches`
	`// the end (this SC should be the last AL component).`
	`//`
	`system.setPersistentObject(“myStateKey”, myStateKey);`

Since the **Iterator State Store** value is used during Connector initialization to set the starting point for change detection, we need to set this value in a Prolog Hook so it is executed before initialization. As a result, the first snippet above could just as well be coded in the AssemblyLine Prolog – Before Init Hook. However, it is best practices to keep component-specific code tied as closely to the component itself as possible.

2.2 The Delta Engine

When the underlying data store does not provide any delta information, you can use the Delta Engine to discover changes for you. One example is detecting differences between daily HR dump files without having to process the entire dump in your AssemblyLine.

The Delta Engine can be set up for any Connector that is in **Iterator** mode and works by keeping snapshots of data read in the System Store. Each time you run the AL these snapshots are compared with new Entries read in by the Iterator. Based on this comparison, all

[6] This shortcoming will be addressed in future releases of TDI and its Change Detection componentry, making the technique shown above unnecessary.

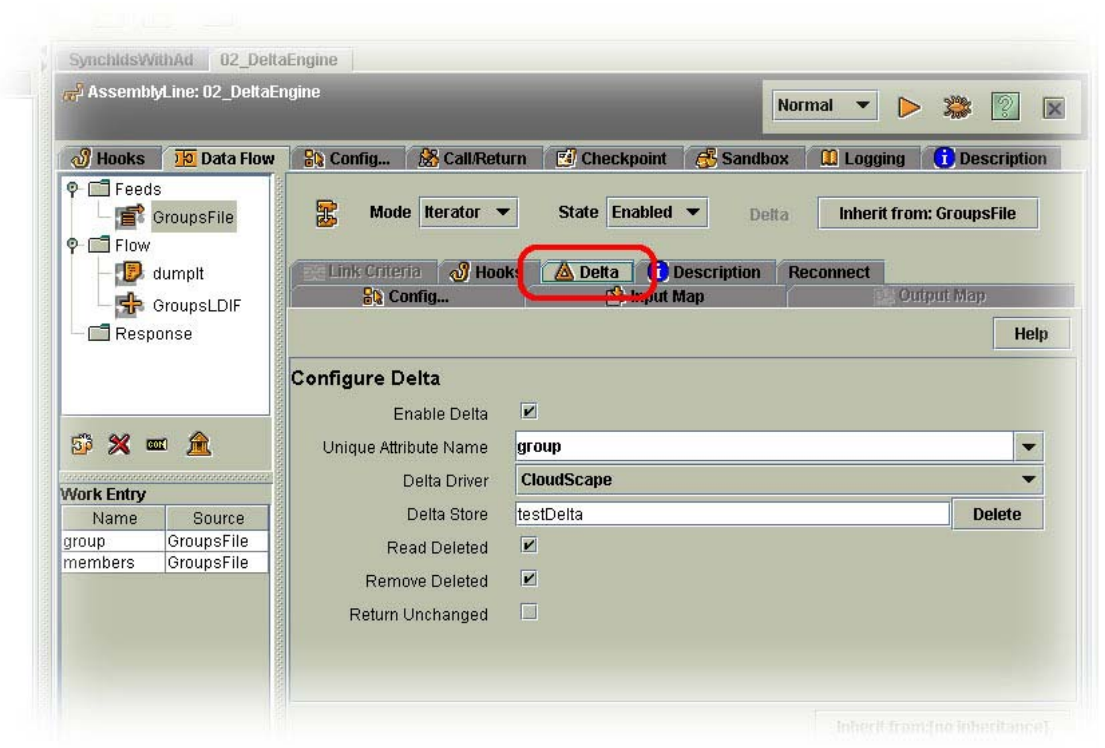
differences are noted and tagged in the Entry, and the snapshot database is updated to reflect the new state of this data.

This means that the first time you run an AssemblyLine with a Delta Engine-enabled Iterator, all Entries will be regarded as *new* ones since there are no previous snapshots of them. After this initial execution, the snapshot database is populated and the Delta Engine has a basis for comparison on future runs.

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

The Delta Engine works in two passes. 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*. 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.

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



Name	Source
group	GroupsFile
members	GroupsFile

Configure Delta

Enable Delta	✓
Unique Attribute Name	group
Delta Driver	CloudScape
Delta Store	testDelta
Read Deleted	✓
Remove Deleted	✓
Return Unchanged	

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[7].
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.
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 *unchanged*.

Although Delta tables can be access 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*).

2.2.1 Performance and Risk

Although 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. Running with the Delta Engine will impact that performance of an AssemblyLine.
2. 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.
3. Should the snapshot database get out-of-synch (for example, if the synchronization AL fails before changes are propagated to all targets) then this will not be automatically detected by the Delta Engine[8]. Instead, you will need to delete the snapshot database in order to build a new baseline. Note that you should create an AssemblyLine for this purpose that not only initializes the snapshot database, but also updates targets with necessary changes. This will require defensive configuration of Delete mode Connectors to deal with data that is already removed from targets. Additionally, setting the Compute Changes flag for Update mode Connectors will ensure that only necessary write operations are performed (see section *4.1.3 Compute Changes* on page 24 for more details on Compute Changes).

[7] 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.

[8] Future releases of TDI will help alleviate this potential problem by ensuring that the snapshot database is updated *after* the AL has successfully completed its cycle.

2.3 LDIF Parser

The LDIF Parser can be used to both write incremental LDIF output based on Delta operation code tagging, as well as to read these files and return the corresponding Delta Entries.

An Incremental LDIF file contains 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) that perform Attribute and value tagging use this Parser on the LDIF information kept in the changelog to do this. To understand what Attribute and value tagging means, continue on to the next section.

3 Delta Tagging

Delta Tagging is the process of marking retrieved data with delta operation codes (also referred to as simply "operation codes"), and is typically done during Delta Detection.

Each operation code describes how the information it is attached to has been 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 delta operation codes themselves.

3.1 Delta Operation Codes

In addition to holding Attributes, an Entry object also carries an *operation code*. This internal Entry variable is set during *Delta Tagging* to a value corresponding to the type of change detected. These operation codes are not visible in the Config Editor, but you can access them from JavaScript through methods (function calls) found in the Entry object, as shown later in this section.

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 will want or need to manipulate 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 a message queue); Or if a TDI component or feature does not provide the level of delta handling that you require.

3.1.1 Entry Operation Codes

The operation code for an Entry can be accessed directly via the getOp() and setOp() methods of the Entry object. These function calls use Java *char* values for the various change types. These values are defined in the Entry Java class (along with their *String* equivalents). The table below is copied from the TDI JavaDocs page for the Entry object:

Field Summary	
`static char`	**OP_ADD** The entry contains an entry which is supposed to be added
`static char`	**OP_DEL** The entry contains an entry which is supposed to be removed
`static char`	**OP_GEN** The entry contains an entry with no explicit knowledge of operation
`static char`	**OP_MOD** The entry contains an entry which is supposed to be modified
`static char`	**OP_UNCHANGED** The entry contains an entry which is unchanged

These codes have the following meanings:

Table 1 - Entry-level Delta Operation Codes

OP_ADD	Signals that the Entry object is new and should be added to the target(s).
OP_DEL	Indicates that the Entry was deleted from the source.
OP_GEN	This code value means that there is no delta tagging available. This is the default operation code for Entry objects returned by any other means than from one of the Delta Detection mechanisms. An Entry with this operation code is considered *untagged* and not a Delta Entry.
OP_MOD	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).
OP_UNCHANGED	Unlike OP_GEN, 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 *unchanged* code.

As an example, the following JavaScript snippet will write a log message if the Work Entry has the *delete* operation code tag:

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

The pre-defined set of operation code values are easily accessible through any instance of an Entry object – as with the reference "`work.OP_DEL`" shown in the above example.

3.1.2 Attribute and Value Operation Codes

Attributes have a similar set of operation codes. Here is the table found at the top of the TDI JavaDocs page for the Attribute object:

Field Summary	
`static char`	**ATTRIBUTE_ADD** Add value
`static char`	**ATTRIBUTE_DELETE** Delete value
`static char`	**ATTRIBUTE_MOD** Values modified
`static char`	**ATTRIBUTE_REPLACE** Replace value
`static char`	**ATTRIBUTE_UNCHANGED** Unchanged

The meaning of these codes is listed in the table below:

Table 2 – Attribute-level Delta Operation Codes

ATTRIBUTE_ADD	Signals that the Attribute is new and should be added to the entry in target(s).
ATTRIBUTE_DELETE	Indicates that the Attribute was deleted from the entry in the source.
ATTRIBUTE_MOD	The Attribute has been modified. This operation code also implies that there may be more delta tags available for the *values* of this Attribute.
ATTRIBUTE_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.
ATTRIBUTE_UNCHANGED	Signals that this Attribute is unchanged.

These codes are read and written in JavaScript code by using the Attribute methods `setOper()` and `getOper()`. For example, the following script first gets[9] the "FullName" Attribute from the Work Entry and then sets the operation code to *modify*:

```
var fullName = work.getAttribute("FullName");
fullName.setOper(fullName.ATTRIBUTE_MOD);
```

Drilling down to the next and final level, Attribute *values* are tagged with the same set of codes used for Attributes. The methods for working with *value*-level delta tags are also found in the Attribute object – `setValueOper()` and `getValueOper()` – both of which require an index parameter that indicates which value to apply the tag to.

```
fullName.setValueOper(0, fullName.ATTRIBUTE_DEL);
```

This snippet sets the operation code for first value[10] 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.

[9] It is important to understand that when you *get* an Attribute from an Entry (as with the getAttribute() method as shown in the example above, you actually get a *reference* to this Attribute – not a copy. So any changes you make are applied directly to this instance of the Attribute that is stored in the Entry.

[10] 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.

3.1.3 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 handled by Delta Application logic as indicated by the Attribute's operation code.
  - If an Attribute has an operation code of *modify*, then its *values 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 21.

3.1.4 Displaying 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'   'alanbrau106'
13:32:34         group (replace):   'Coffee Drinkers'
13:32:34  *** End Entry Dump
```

The above dump shows that Entry itself tagged as *generic*[11], 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'
```

[11] Functions like task.dumpEntry() use the more legible String variants of the delta operation codes. For example, OP_MOD is displayed as "modify".

```
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() );
```

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

```
modify {
    members {
            type: modify
            count: 3
            values [
                    add: abnevanm408
                    unchanged: abdaburr393
                    unchanged: alanbrau106
            ]
    }
    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*".

3.1.5 Manual Delta Code Tagging

Although all Change Detection components and features return information on how data is changed, tagging of Entries, Attributes and *values* is not done by all of them[12]. The following table lists all Change Detection mechanisms, along with their level of delta code tagging.

Note that although the current set of Changelog Connectors does not provide full tagging of returned data, an example is included later in this section on how to correct this manually. This technique takes advantage of the Attribute (named for each Connector in the table below) that holds this change information.

Table 3 - Change Detection Mechanisms and Tagging Levels

LDIF Parser	Tags Entries and Attributes/values.
Delta Engine	Tags Entries and Attributes/values. 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.
IBMDirectoryServer Changelog Connector	Does not tag the Entry (generic tagging), but does tag Attributes and Attribute values. Returns type of change in an Attribute called “**changeType**” with the value “add”, “modify” or “delete” (or “rename” for a rdn/$dn change).
Netscape/iPlanet Changelog Connector	Does not tag the Entry (generic tagging), but does tag Attributes and Attribute values. Returns type of change in an Attribute called “**changeType**” with the value “add”, “modify” or “delete” (or “rename” for a rdn/$dn change).

[12] Note that future releases of TDI and its Delta Detection components will improve the handling of Delta Tagging, further harmonizing the way that delta information is returned.

Active Directory Changelog (v.2) Connector	Does not tag Entries (generic tagging), Attributes or their *values*. Returns type of change in an Attribute called "**changeType**" with the value "update" (for both *add* and *modify*) or "delete". 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.
Domino Change Detection Connector	Does not tag Entries (generic tagging), Attributes or their *values*. Returns type of change in an Attribute called "**$$ChangeType"** with the value "add", "modify" or "delete". 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.
RDBMS Changelog Connector	Does not tag Entries (generic tagging), Attributes or their *values*. Returns type of change in an Attribute called "**IBMSNAP_OPERATION**" with values "I" for Inserted (*add*), "U" for Updated (*modify*) or "D" for Deleted (*delete*).
Exchange Changelog Connector	Does not tag Entries (generic tagging), Attributes or their *values*. Returns type of change in an Attribute called "**changeType**" with the value "update" (for both *add* and *modify*) or "delete". 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.

As shown in the above table, the current set of Changelog Connectors do not tag returned Entries. But don't panic. As listed above, they all return an Attribute that describes how data is changed. Once you have this, it's easy to set the operation codes yourself.

As an example, consider an AssemblyLine using the IBMDirectoryServer Changelog Connector. This component returns Entries containing an Attribute called "changeType", which is then used in the following script code to set the operation code of the Entry itself[13]:

```
if (work.getString("changeType") == "delete")
    work.setOp(work.OP_DEL)
else
if (work.getString("changeType") == "add")
    work.setOp(work.OP_ADD)
else
if (work.getString("changeType") == "modify")
    work.setOp(work.OP_MOD);
```

Since this Connector does tag Attributes and their values, the manually tagged Work Entry is now ready to be passed to Delta Application with correct codes in place.

[13] 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 Changelog Iterator. By naming this Script Component descriptively, for example "PerformDeltaTagging", your AssemblyLine becomes easier to read and maintain. Furthermore, this scripted tagging can easily be identified and removed once an enhanced version of the Changelog Connector becomes available.

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.

4.1.1 Manual Delta Application

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 the first method makes use of the Branch component in TDI 6.0, 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 will need two Branches. One will have the following Condition:

changeType EQUALS delete

where changeType above is the name of the Attribute that carries this information, and delete is literal string value in this Attribute that indicates a deleted Entry.

If the delta operation tag of the Entry has been set, you could alternatively check this instead, but then you would need to script the Condition like this:

```
ret.value = work.getOp() == work.OP_DEL
```

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

Just after the above Branch you will have another one with the reciprocal Condition, for example:

```
ret.value = work.getOp() != work.OP_DEL
```

and nested under this Branch will be your Update mode Connector.

Before Execute Hook

This 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.

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

This snippet drives a Connector called "targetConnector"[14], 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.

Prior to the advent of Delta mode in version 6.0, these were the methods available for applying delta to target systems.

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 LDAP directories. This can represent a significant performance improvement since load on the LDAP Server and network is minimized, especially when working with group membership or other massively multi-valued Attributes.

In order for Delta Mode to work, the Connector must receive a Delta Entry (e.g. with an operation code value other than *generic*). 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

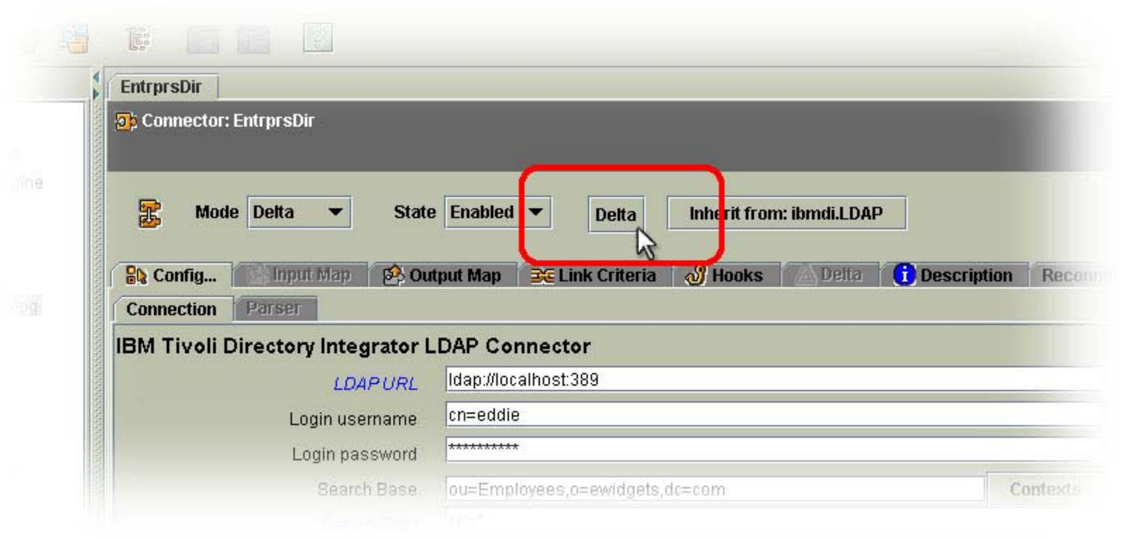
[14] All AssemblyLine Components are automatically registered as script variables, which is why it is important to name them as you would a variable.

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" 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. This is not a problem when the Delta Detection mechanism used only provides a single change per entry, as with the Domino Change Detection Connector. (see

Table 3 - Change Detection Mechanisms and Tagging Levels on page 14 for more details).

Which operation codes Delta mode should handle, as well as how it deals with unwanted entries, is configured by pressing the Delta button[15] at the top of the Connector Details panel.


EntrprsDir

Connector: EntrprsDir

Mode: Delta

State: Enabled

Delta

Inherit from: ibmdi.LDAP

Config... | Input Map | Output Map | Link Criteria | Hooks | Delta | Description

Connection | Parser

IBM Tivoli Directory Integrator LDAP Connector

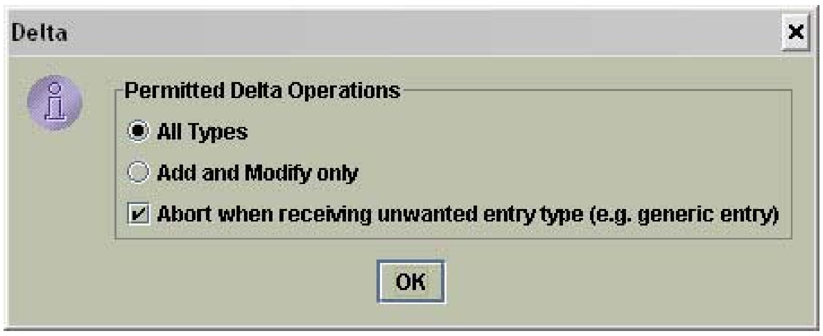
LDAP URL: ldap://localhost:389

Login username: cn=eddie

Login password: **********

Search Base: ou=Employees,o=ewidgets,dc=com

This brings up the Permitted Delta Operations dialog.


Delta

Permitted Delta Operations

● All Types

○ Add and Modify only

☑ Abort when receiving unwanted entry type (e.g. generic entry)

OK

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 sent directly to the LDAP directory. A common scenario where this

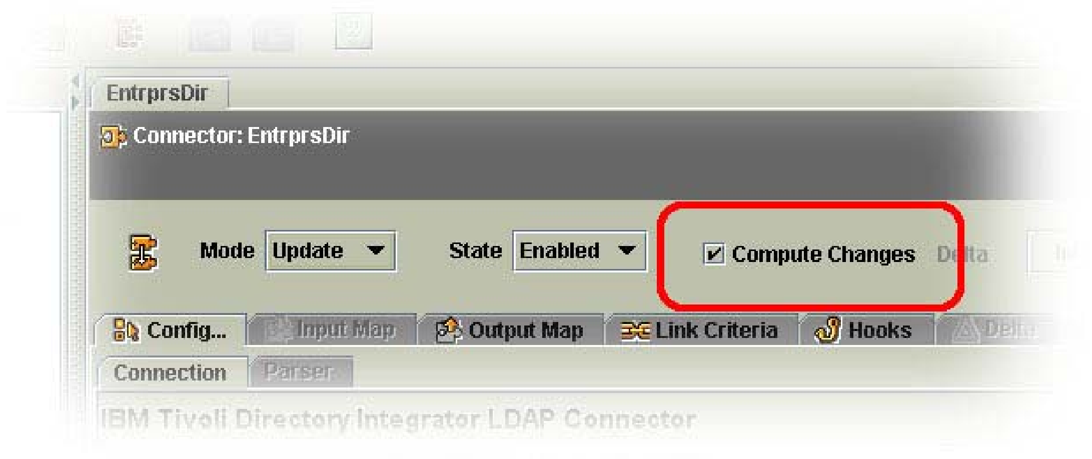
[15] This button only appears to be a button when you move the mouse over it.

improved functionality is important is when changing group memberships, where you are typically adding or removing values from large multi-valued Attributes.

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* Entry object by the lookup operations. If no differences are detected, then the modify operation is not carried out.


EntrprsDir
Connector: EntrprsDir
Mode Update
State Enabled
☑ Compute Changes

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

5 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 for Delta Detection/Tagging, Delta Application or both, they provide building blocks for laying the foundation of your solution faster.

6 References

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