# Endeca® MDEX Engine

## Partial Updates Guide

# Contents

# Copyright and disclaimer

## Trademarks

Endeca Partial Updates Guide • December 2010

Version 6.1.4

# Preface

Oracle Endeca's Web commerce solution enables your company to deliver a personalized, consistent customer buying experience across all channels — online, in-store, mobile, or social. Whenever and wherever customers engage with your business, the Oracle Endeca Web commerce solution delivers, analyzes, and targets just the right content to just the right customer to encourage clicks and drive business results.

Oracle Endeca Commerce is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Oracle Endeca Commerce enables businesses to help guide and influence customers in each step of their search experience. At the core of Oracle Endeca Commerce is the MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Endeca Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. Endeca Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

Oracle Endeca Experience Manager is a single, flexible solution that enables you to create, deliver, and manage content-rich, cross-channel customer experiences. It also enables non-technical business users to deliver targeted, user-centric online experiences in a scalable way — creating always-relevant customer interactions that increase conversion rates and accelerate cross-channel sales. Non-technical users can control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Oracle Endeca Experience Manager, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

## About this guide

This guide describes different types of updates. It concentrates on the basic tasks involved in configuring partial updates and understanding when to use them.

It assumes that you have read the *Endeca Concepts Guide* and the *Endeca Getting Started Guide* and are familiar with the Endeca terminology and basic concepts.

Use the *Partial Updates Guide* to learn how to run record updates in the MDEX Engine:

1. Learn about baseline, partial and delta updates, what changes you can make within each type of update, and when to use each type of update.
2. Learn about update requirements and the MDEX Engine processing of updates.
3. Create your partial updates pipeline in Developer Studio.
4. Back up your update files.
5. Troubleshoot the updates process.

## Who should use this guide

This guide is intended for developers who are creating pipelines for running partial updates.

## Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

## Contacting Oracle Support

Oracle Support provides registered users with important information regarding Oracle Endeca software, implementation questions, product and solution help, as well as overall news and updates.

You can contact Oracle Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1

# Types of Updates

This section gives an overview of the types of Endeca updates.

## Updates processed by the MDEX Engine

The MDEX Engine processes three types of updates. While this topic introduces all three types of updates, this guide focuses on partial updates.

- Baseline updates. Baseline updates (also called full updates) include reindexing of the data and require stopping and restarting the MDEX Engine.
- Delta updates. Delta updates are a variation of baseline updates. In delta updates, only added, updated, or removed source records (and not all source records) are joined in the Forge pipeline with the output of the previous baseline update. Delta updates require stopping and restarting the MDEX Engine.
- Partial updates. Partial updates are incremental changes to the data set in the MDEX Engine. Partial updates run in a perpetual mode (that is, they do not require the MDEX Engine to be restarted) and therefor are referred to as perpetual partial updates.

**Related Links**

*MDEX Engine processing for partial updates* on page 14
At a high level, the MDEX Engine performs the following operations.

*About baseline updates* on page 10
A baseline update produces a complete re-indexing of the entire data set. It runs the update process for the whole data set.

*About partial updates* on page 11
A partial update is a change in the overall data set that does not require restarting the MDEX Engine. Partial updates allow you to update only those portions of the MDEX Engine index that have changed since the last baseline update.

*About delta updates* on page 11
A delta update produces a full baseline index, similar to the baseline update, but does so by joining a smaller extraction of source data (only the added, updated, or removed source records) with the output from the previous baseline update.

## About baseline updates

A baseline update produces a complete re-indexing of the entire data set. It runs the update process for the whole data set.

In your baseline update pipeline, you can add, change, or remove records, dimensions, dimension values, and properties. In addition, configuration changes, such as dimension reordering or stop word changes require a baseline update.

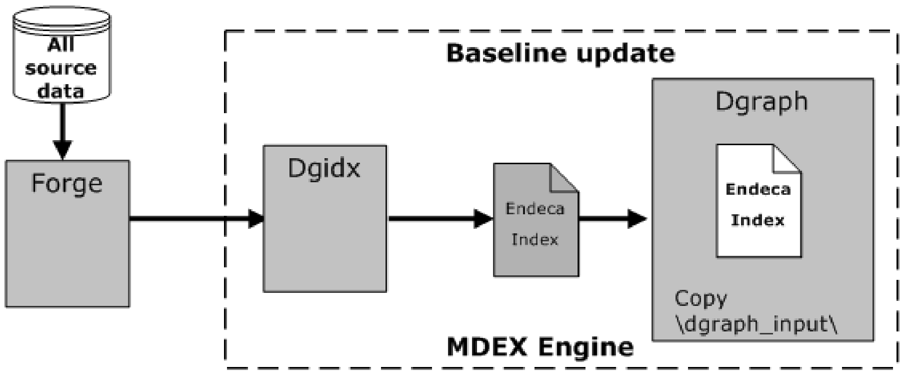
In many Endeca implementations, you can run baseline updates nightly and use this method as your update strategy, skipping other types of updates. For small to medium-sized data sets, baselines can be run frequently, as often as every few minutes.

Alternatively, you can run as many partial updates as needed for those changes that can be done through partial updates, and periodically run baseline updates for those changes that require a baseline update.

### Baseline update processing

In the baseline update process, Forge takes as its input the data from the pipeline configuration files and all the source data.

As a result of ITL processing, the Endeca index is created. A copy of the index is added to the `dgraph_input` directory. (The directory name used here is arbitrary and is provided as an example only. You can specify your own name for this directory.) This is the index that the Dgraph takes as its input to start processing queries.



**Note:** The MDEX Engine modifies the `dgraph_input` directory with the information received from each successfully applied partial update. That is, this directory is not only read by the MDEX Engine upon a restart (after a baseline update), but is also modified by the MDEX Engine at run time.

### Speeding up baseline updates

There are several techniques for speeding up the baseline update process.

- Speeding up the extraction process by using the Endeca Content Acquisition System (CAS) module to enable multithreaded document conversion, for example, from PDF or Word.
- Speeding up the indexing time during a baseline update by:

- Analyzing your pipeline for any preprocessing steps that can be reused, such as Perl scripts or large joins.
- Analyzing the incoming data for the presence of any unnecessary wildcarding.

## About partial updates

A partial update is a change in the overall data set that does not require restarting the MDEX Engine. Partial updates allow you to update only those portions of the MDEX Engine index that have changed since the last baseline update.

**Related Links**

*Partial Updates Processing and Requirements* on page 13
This section describes how partial updates work and the requirements for source data and the pipeline.

## About delta updates

A delta update produces a full baseline index, similar to the baseline update, but does so by joining a smaller extraction of source data (only the added, updated, or removed source records) with the output from the previous baseline update.

A delta update could be an option for you when you need to reduce the time required for loading the source data repository and for extracting the data.

For more information on delta updates and when to use them, see the solution article “Implementing Delta Updates” available online from the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

**Note:** Starting with version 6.0, the MDEX Engine can accommodate high volumes of changed data during partial updates without significant performance degradation. Therefore, implementing a delta updates pipeline may no longer be worthwhile. Instead, you can run as many partial updates as needed for those changes that can be done through partial updates, and periodically run baseline updates for those changes that require a baseline update.

## Which update to run

In your project, you can have changes to the source data, or changes to the project configuration, such as changes to the way you order or organize dimensions. Depending on the type of changes you make to the source data and to the Endeca project configuration, your implementation may require a different type of update.

Baseline and delta updates let you implement all types of changes, both to the source data and to the project configuration, but can be time consuming. A partial update is faster and lets you implement a number of the source data changes but not project configuration changes, such as dimension reordering. For these types of changes, periodically run baseline updates.

Partial updates with high turnover and high frequency perform fast. High turnover means that a large portion of data can be updated or deleted. Any mix of add, delete, and update operations on a large number of records is handled gracefully during partial updates.

In addition, you can combine record updates into larger batches. Running such large-batch partial updates results in better overall throughput for the MDEX Engine.

Chapter 2

# Partial Updates Processing and Requirements

This section describes how partial updates work and the requirements for source data and the pipeline.

## Introduction to partial updates

A partial update is a change in the overall data set that does not require restarting the MDEX Engine.

Partial updates are run concurrently with processing incoming queries. They allow you to update only those portions of the MDEX Engine index that have changed since the last baseline update.

A partial update lets you implement a number of the source data changes, but does not affect project configuration. For project configuration changes, run a baseline update.

Even if you are only making source data changes, keep in mind that some configuration information that is derived from the data, such as dictionary or wildcarding information, can become outdated. Therefore, to keep dictionaries up-to-date, periodically run baseline updates.

## How partial updates work

This topic enumerates how the MDEX Engine version 6.x treats partial updates.

- The MDEX Engine processes partial updates concurrently with processing queries. This function is also known as *continuous query*.
- You can use partial updates to perform a large number of changes to a considerable portion of records.
- You can run partial updates frequently.
- Partial updates can affect any number of records in the system.

## Partial updates and the Deployment Template

An EAC partial update script is created and managed for you when you use the Endeca Deployment Template; you can change this script to suit your needs. You can also create your own partial update script in the Endeca Application Controller (EAC) environment and provision it to the EAC using Endeca Workbench.

Endeca recommends using the Deployment Template, which is available as a free download from the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

## Introduction to partial updates processing

In the partial update process, Forge takes as its input the data from the partial pipeline configuration files and the updates data.

As a result of ITL processing, the update files are created. These update files are applied to the MDEX Engine index.

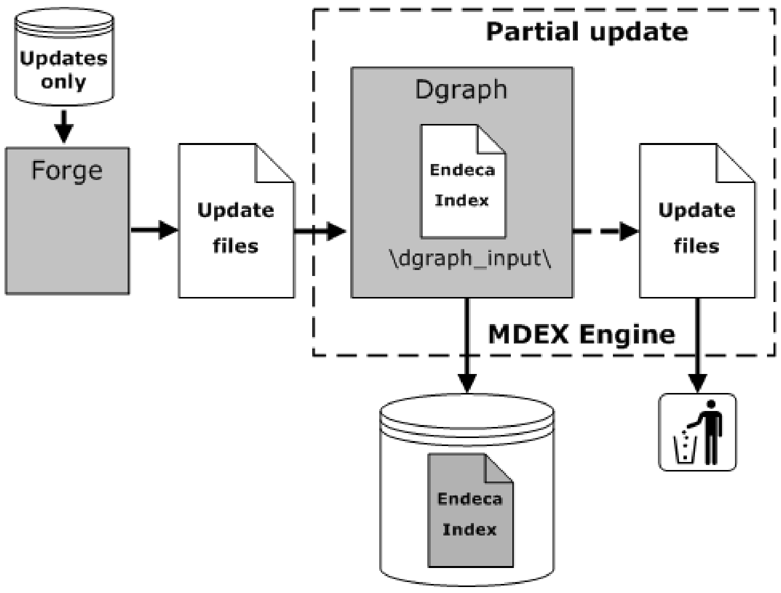
The MDEX Engine does not close its port to incoming queries while processing partial updates.

When applying a partial update, the MDEX Engine modifies the `dgraph_input` directory and updates it with the new data received from a partial update. After a partial update completes successfully, the MDEX Engine automatically deletes the contents of this update from the `dgraph_input/updates` directory.

Although the MDEX Engine deletes the update files after it applies updates to the index, it continues to check the `dgraph_input/updates` directory each time it restarts. If you start up the MDEX Engine with update files in the updates directory (that have not been applied yet), the MDEX Engine applies these initial updates first, before starting to answer queries.

Partial updates are continuously applied to the in-memory representation of the data structures in the MDEX Engine, and to the index structures on disk that the MDEX Engine uses for processing queries.

This diagram describes partial updates processing:



## MDEX Engine processing for partial updates

At a high level, the MDEX Engine performs the following operations.

1. Once it receives the update files, the MDEX Engine modifies the on-disk representation of the Endeca index to reflect the updates.
2. After the update files are applied to the Endeca index, the MDEX Engine deletes the contents of the partial update from the `dgraph_input/updates` directory.

### Integrity of generation files upon recovery

If the server crashes while a partial update is being applied, when the MDEX Engine starts up again, it will go through its list of generation files to determine if any generation file is in a bad state. If any are bad, the MDEX Engine will automatically delete them before applying the next partial update. In other words, no manual intervention is required in the MDEX Engine's generations directory (and any user modification to the generation files in an attempt to rollback the generations is not supported).

## Continuous query

The MDEX Engine processes partial updates concurrently with processing query requests. This function is also known as continuous query.

During continuous query processing, the MDEX Engine Dgraph port remains open for both query processing and partial updates processing.

Continuous query is enabled for all types of queries to the MDEX Engine, including navigation, aggregated records and record queries, queries with text search, queries that contain filters (EQL, range and record filters), queries containing Web services and XQuery, and all other types of queries.

A few administrative queries are processed differently during continuous query processing. For details, see the topics in this section.

Since the MDEX Engine continues to process all incoming queries while partial updates are running, queries are processed against either the pre-update or post-update state of the index data, depending on when they arrive. The MDEX Engine never processes queries against the data that is in the state of being updated through a partial update.

With continuous query, the MDEX Engine maintains its query processing performance levels, including low query latency and partial updates latency.

### MDEX Engine startup behavior

Continuous query begins at startup time for the MDEX Engine. For example, assume a scenario where a server outage occurred during the application of a partial update. As a result, several large partial files remained in the updates directory. When the MDEX Engine is restarted, it opens its port for both query processing and partial updates processing. This means that the MDEX Engine's startup behavior is to process updates in parallel with queries, that is, the MDEX Engine starts processing queries immediately even when updates are found in the updates directory at startup time.

## Continuous query processing and administrative queries

You can issue administrative queries to the MDEX Engine concurrently with running updates, without any interruptions caused by partial updates processing, except for the following administrative and configuration queries that are processed differently.

- `admin?op=exit`
- `admin?op=restart`
- `admin?op=updateaspell`

- `admin?op=reload-services`
- `config?op=update`

`admin?op=exit` and `admin?op=restart` queries cause the MDEX Engine to close its Dgraph port for accepting future queries. Next, the MDEX Engine processes all previously received queries and shuts down (or restarts, depending on which of these two commands is issued).

The `admin?op=updateaspell` operation causes the MDEX Engine to finish processing all existing preceding queries, temporarily stop processing other queries and begin to process `admin?op=up¬dateaspell`. After it finishes processing this operation, the MDEX Engine resumes processing queries that queued up temporarily behind this request. Only one `admin?op=updateaspell` operation can be processed at a time.

`config?op=update` and `admin?op=reload-services` operations cause the MDEX Engine to drain all existing preceding queries, temporarily stop processing other queries and begin to process `config?op=update` and `admin?op=reload-services`. After it finishes processing these operations, the MDEX Engine resumes processing queries that queued up temporarily behind these requests.

Only one `config?op=update` operation can be processed at a time.

**Note:** `config?op=update` and `admin?op=reload-services` can be time-consuming operations. This depends on the number of configuration files the MDEX Engine has to process for an update (during `config?op=update`), or the number of XQuery modules that you have created and that have to be compiled (during `admin?op=reload-services`).

Processing time for the `admin?op=updateaspell` administrative query can be higher compared with the time it takes the MDEX Engine to process partial updates. Processing time depends on the scope of updates to the spelling dictionary sent to the MDEX Engine with this operation.

You can issue all other administrative queries to the MDEX Engine concurrently with updates, without any interruptions caused by partial updates processing.

## The dgraph_input directory

The Dgraph writes to the `dgraph_input` directory during normal MDEX Engine operation.

The `dgraph_input` directory contains regular data that is read by the MDEX Engine on startup. The data includes the Dgidx output indices, spelling correction dictionaries, thesaurus files and language-encoding files. In previous releases, this directory was read-only.

The data in `dgraph_input` is modified using the data from the `dgraph_input/updates` directory each time a partial update completes successfully.

**Related Links**

*The dgraph_input/updates directory* on page 16

The `dgraph_input/updates` directory contains partial updates data that have yet to be processed by the MDEX Engine. The MDEX Engine checks this directory for update files, when the `--updatedir` flag is specified for the Dgraph.

## The dgraph_input/updates directory

The `dgraph_input/updates` directory contains partial updates data that have yet to be processed by the MDEX Engine. The MDEX Engine checks this directory for update files, when the `--updatedir` flag is specified for the Dgraph.

After a partial update completes successfully, the Endeca index is updated with the changes from that update. The update files are no longer needed by the MDEX Engine and are automatically removed from the `dgraph_input/updates` directory.

The default MDEX Engine behavior includes checking the `dgraph_input/updates` on restart. When the Dgraph starts, it checks for the update files in the `dgraph_input/updates` directory, in case this directory contains any of them. Under normal conditions, this directory should be empty since the MDEX Engine deletes update files after applying them.

Checking the `dgraph_input/updates` directory is also useful if partial updates become available immediately after a baseline update. For example, consider a case when a baseline update runs, followed by a partial update that captures data that became available while the baseline was running. If you restart the Dgraph immediately after this partial update, the MDEX Engine reads these most recent updates after it restarts and then proceeds to answer queries.

If the Dgraph server crashes in the middle of a partial update, the files from that update are not deleted from `dgraph_input/updates`. When the MDEX Engine is restarted, it retains the index state it had before this partial update was attempted. After a restart, the MDEX Engine checks the contents of `dgraph_input/updates` for the presence of last partial update files that were not applied and applies them.

Endeca recommends that you back up this directory to ensure that you can recover after a disk failure.

**Related Links**

*Backing Up Baseline and Partial Updates* on page 49

Endeca recommends that you back up your MDEX Engine index files periodically. This lets you revert to a specific partial or baseline update. This section describes types of backups that you can perform for the MDEX Engine index files, lists backup recommendations, and describes how to recover the index by reverting to a previous state of the MDEX Engine index.

*The dgraph_input directory* on page 16

The Dgraph writes to the `dgraph_input` directory during normal MDEX Engine operation.

# Partial update capabilities

You can perform a limited number of actions within partial updates. This topic lists these actions and also lists those changes that you cannot perform with partial updates.

You can perform the following actions with partial updates or while running partial updates:

- Add an entirely new record with a new set of property values and dimension values to an existing index.
- Remove a specific record from an existing index.
- Modify selected property and dimension values in an existing record.

**Note:** When you remove a record or modify property and dimension values for the record, the dimension values that are no longer associated with any records remain in the system.

- Update an existing record, selectively adding and removing dimension and property values. Specifically, you can:
  - Add property values to a record.
  - Remove all property values of a property from a record.
  - Add leaf dimension values (but not mid-hierarchy values) to a record.

- Remove specific dimension values from a record.
- Remove all dimension values of a dimension from a record.
- Add new auto-generated dimension values to an existing dimension.
- Add new leaf dimension values that have been created as a result of Term Discovery extraction.
- Add synonyms to a new leaf dimension value, when you are adding this dimension value. The first synonym becomes a dimension value's name. (Also, the same string can be used as a synonym for multiple dimension values.)
- Update spelling dictionaries. Use the `admin?op=updateaspell` operation to update the spelling dictionary while running partial updates.

**Note:** Use the baseline update for these operations.

You cannot do the following actions with partial updates or while running partial updates:

- Add new dimensions.
- Add new mid-hierarchy dimension values.
- Add, delete, or change any aspect of an existing dimension value. For example, you cannot add, change, or remove dimension value properties. You also cannot add or change bounds for range or sift dimension values, change whether a dimension value is inert (non-navigable), or whether it is collapsible.
- Add dimension properties to any dimension values.
- Add new properties.
- Update the index configuration files (such as the thesaurus and stop words files).
- Update dynamic word forms. Dynamic word forms are calculated at index time and are not updated with partial updates.

## Requirements for baseline and partial updates

Forge processing for baseline updates and partial updates is done in separate pipelines, but is coordinated and synchronized. The processing of partial updates affects a running MDEX Engine.

The required coordination between the baseline and partial update pipelines, coupled with the resource restrictions on the partial update pipeline, impose constraints on the baseline update pipeline. This section lists the requirements for running partial and baseline updates.

### Related Links

*General requirements for partial and baseline updates* on page 19
This section lists the general requirements for baseline and partial updates.

*Adding new leaf dimension values in partial updates* on page 19
This section describes one method of adding new leaf dimension values for existing dimensions in partial updates. In partial updates, you can create new leaf dimension values. (You cannot add new dimensions or new mid-hierarchy dimension values.)

*Record specifier attribute required for partial updates* on page 20
A partial updates deployment must have the `RECORD_SPEC` attribute of one property set to `TRUE`. If no property is marked as the `RECORD_SPEC` property, the MDEX Engine will not process partial updates.

## General requirements for partial and baseline updates

This section lists the general requirements for baseline and partial updates.

- Baseline updates are needed for configuration changes.

  Periodically run baseline updates to ensure that the index representation in the MDEX Engine is in sync with those configuration changes that can be applied only with a baseline update, such as dictionary changes or wildcarding.

- Partial updates require a separate partial update pipeline to process the update files. You also start the MDEX Engine with an additional command-line flag.
- All records in the partial update pipeline must be identified by a single record specifier property that is unique for each record.
- Baseline updates must not overlap. A new baseline cannot be started until processing of the prior baseline has been completed (completed means that the baseline update has been loaded into the MDEX Engine).
- Baseline and partial updates must not overlap. Do not run a baseline update at the same time as a partial update, since both processes use the `autogen_dimensions.xml` file that can be accessed by only one process at a time.

**Related Links**

*Record specifier attribute required for partial updates* on page 20

A partial updates deployment must have the `RECORD_SPEC` attribute of one property set to `TRUE`. If no property is marked as the `RECORD_SPEC` property, the MDEX Engine will not process partial updates.

## Adding new leaf dimension values in partial updates

This section describes one method of adding new leaf dimension values for existing dimensions in partial updates. In partial updates, you can create new leaf dimension values. (You cannot add new dimensions or new mid-hierarchy dimension values.)

Before adding a new leaf dimension value, ensure that a dimension already exists in your record set for the leaf dimension value. For example, if you are planning to add a record with a new leaf dimension value of `“Australia”`, ensure that a dimension `“region”` is already specified for your records.

To do this, in your baseline update pipeline, add a new dimension in the **Dimension** editor in Developer Studio, and select the **Auto Generate** mode in the **Property Mapper** editor. This generates dimension values for the dimension. Run a baseline update.

In order to add leaf dimension values, you can use the **Property Mapper** editor option to automatically generate new values for existing dimensions.

To add a new leaf dimension value in a partial update:

1. Add a new record to your record set that includes a leaf dimension value for the dimension that already exists in your project (and that was added before).
2. Modify the record manipulator to indicate that you are updating dimension values. Use the `UP¬DATE_RECORD` expression with `<EXPRNODE NAME="DIM_ACTION" VALUE="ADD"/>`.
3. Run a partial update. The newly added leaf dimension value is added to the project during a partial update.

For example, when you add a new record that has a value `“Australia”`, the partial update generates a new value for `region=”Australia”` when the new record goes through the property mapper in the partial update pipeline.

The output XML files in Forge that result from the partial update include the new record, as well as the new dimension value for it. When the Dgraph is updated with that file, it includes the new record, as well as the new dimension value of `“Australia”`.

## Record specifier attribute required for partial updates

A partial updates deployment must have the `RECORD_SPEC` attribute of one property set to `TRUE`. If no property is marked as the `RECORD_SPEC` property, the MDEX Engine will not process partial updates.

The `RECORD_SPEC` attribute specifies the property that is used to identify specific records in partial updates. For example, you may wish to use a field such as `UPC`, `SKU`, or part number to identify a record. You may set the `RECORD_SPEC` attribute’s value to `TRUE` in any property where the values for the property meet the following requirements:

- The value for this property on each record must be unique.
- Each record should be assigned exactly one value for this property.

Only one property in the project may have the `RECORD_SPEC` attribute set to `TRUE`. All updates that add new records must include a valid value (that is, a value that fulfills the above criteria) for the `RECORD_SPEC` property.

Chapter 3

# MDEX Engine Configuration and Processing

This section describes how the Endeca MDEX Engine handles partial updates.

## Enabling the MDEX Engine for partial updates

You must start the MDEX Engine with the Dgraph `--updatedir` flag to enable it to process partial updates.

The flag takes as an argument the path of the directory into which completed partial update files (from Forge) are placed. During normal operation, the MDEX Engine does not automatically load update files placed into this directory. The MDEX Engine checks this directory by default upon restart, after a baseline update. The scripts that you use for partial updates must be configured to notify the running MDEX Engine to check for new updates. After the MDEX Engine reads these files, they are deleted from this directory.

Update files are read at startup (after a baseline) as well as when the MDEX Engine receives the update signal. Because the MDEX Engine looks for update files automatically at startup, recovery from server failure can be achieved easily by ensuring that the MDEX Engine is provided the same `--up¬datedir` directory on recovery as it had prior to failure. (This is true only if you restore partial updates first.) The MDEX Engine then reads the existing files in the directory, restoring the MDEX Engine to its pre-failure state.

**Related Links**

*Backing Up Baseline and Partial Updates* on page 49

Endeca recommends that you back up your MDEX Engine index files periodically. This lets you revert to a specific partial or baseline update. This section describes types of backups that you can perform for the MDEX Engine index files, lists backup recommendations, and describes how to recover the index by reverting to a previous state of the MDEX Engine index.

## Using the URL update command

To instruct the MDEX Engine to begin processing the partial update files, use the URL `update` command.

- In your Web browser, issue the `update` command with this URL syntax:

```
http://hostname:dgraphport/admin?op=update
```

For example:

```
http://localhost:8000/admin?op=update
```

**Note:** If you are using HTTPS mode, use `https` in the URL.

On receiving the URL `update` command, the MDEX Engine by default performs the following sequence of operations:

1. Continues processing queries concurrently with processing the update.
2. Checks the updates directory and uploads all partial updates that have not yet been uploaded.
3. Processes the update files and deletes them.

**Note:** The MDEX Engine also deletes any update files that are empty. This includes files that have opening and closing XML tags but no actual update content.

When you issue the URL `update` command, wait until it finishes before you issue another `admin` or `config` URL command (such as an `config?op=update` command).

## Running updates on a single file

In some cases, you may need to run a partial update by pointing the Dgraph to a single file by using the `admin?op=update&updatefile=filename` option.

The recommended way of running partial updates is by using the `admin?op=update` URL command that applies all files residing in the `dgraph_input/updates` directory (or the directory that you specify for updates with the `--updatedir` flag). However, pointing the Dgraph to a single updates file may be useful for performance testing purposes, such as when you plan to run Eneperf in the two-stream mode to test MDEX Engine performance with partial updates.

**Note:** For running Eneperf in the two-stream mode, you first need to obtain a separate request log that contains only partial update requests issued to the MDEX Engine. You can obtain such a log when you run several partial updates on single update files. For more information on running Eneperf for testing updates performance, see the *Performance Tuning Guide*.

To run a partial update on a single file:

1. Add the update file to the `dgraph_input/updates` directory or the directory specified using the `--updatedir` flag.
2. In your Web browser, issue the update command with this URL syntax:

   ```
   http://hostname:dgraphport/admin?op=update&updatefile=filename
   ```

   where filename is the name of an update file residing in the updates directory.

   You can run this command on a single file only. If you have more than one file, rerun this command once for each file.

   The MDEX Engine deletes the update file after successfully applying the results of the partial update.

**Note:** For performance reasons, Endeca recommends running partial updates in batch mode, by only using the `admin?op=update` command. This command applies all update files present in the `dgraph_input/updates` directory.

## Setting the merge policy

You can set the merge policy of the MDEX Engine via a Dgraph command flag or a URL command.

An MDEX Engine's merge policy determines how frequently it merges partial update generations. The data layer that stores the index is a versioning data store. As a result:

- Old versions can be accessed while new versions are created.
- Old versions are garbage-collected when no longer needed.

A version is persisted as a sequence of generation files. A new version appends a new generation file to the sequence. Query latency depends, in part, on the number and size of generation files used to store the index.

Generation files are combined through a process called *merging*. Merging is a background task that does not affect the MDEX Engine functionality, but may affect its performance. Because of this, you can set the policy that dictates the aggressiveness of the merges; this policy is called the *merge policy*.

The merge policy can be controlled through a Dgraph flag or through the admin interface via a URL command. Using these controls, you can set the merge policy to one of two settings:

- `balanced` – This policy that strikes a balance between low latency and high throughput. This is the default policy of the MDEX Engine.
- `aggressive` – This policy merges frequently and completely to keep query latency low at the expense of average throughput.

The balanced policy is recommended for the majority of applications. But aggressive merging may help those applications that meet the following criteria:

- Query latency is the primary concern.
- A large fraction of the records (for example, 20%) are either modified or deleted by partial updates before re-baselines.
- The time to perform an aggressive merge is less than the time between partial updates.

**Note:** Under normal conditions, you do not need to change the default balanced policy. However, you may need to change to an aggressive policy based on a recommendation from Endeca Support.

### Dgraph mergepolicy flag

The Dgraph `--mergepolicy` flag allows you to set the default merge policy of the MDEX Engine at startup time.

The format of the flag is:

```
--mergepolicy <policy>
```

where *policy* is one of these two arguments:

- `balanced` – Sets a policy that strikes a balance between low latency and high throughput. This is the default policy, which means that a balanced policy is used if you do not specify the `--mergepolicy` flag when the MDEX Engine starts up.
- `aggressive` – Sets a policy that merges frequently and completely to keep query latency low at the expense of average throughput.

The MDEX Engine will exit with a fatal error if the `--mergepolicy` flag is used without an argument or with an argument other than `balanced` or `aggressive`.

## URL mergepolicy command

Use the URL `mergepolicy` command to force a merge, and (optionally) to change the merge policy of a running MDEX Engine.

You use the URL `mergepolicy` command to:

- Manually force a merge. This is called a one-time version because after the merge is performed (via a temporary `aggressive` change to the merge policy), the merge policy reverts to its previous setting.
- Change the merge policy of the running MDEX Engine.

### Performing one-time merges

The one-time version of the command is intended for the use case of performing a complete merge of all generations without making a change to the default merge policy.

The format of the one-time version of the command is:

```
/admin?op=merge&mergepolicy=aggressive
```

The assumption is that the MDEX Engine is using a balanced merge policy, and you want to temporarily apply an aggressive policy so that the merge can be performed. After the merge is performed, the merge policy reverts to its previous setting.

When you issue the command, the resulting Web page will look like this example:

```
Dgraph admin, OK.
Dgraph Manual merge started at Sat Jul 17 09:52:47 2010
```

### Changing the current merge policy

The sticky version of the command is intended to change the merge policy of the running MDEX Engine. The duration of the policy change is for the life of the current Dgraph process (that is, until the MDEX Engine is restarted) or until another sticky change is performed during the current Dgraph process.

The format of the sticky version of the command is:

```
/admin?op=merge&mergepolicy=<policy>&stickymergepolicy
```

where *policy* is either `balanced` or `aggressive`.

The command also performs a merge operation if warranted.

This example:

```
http://localhost:8000/admin?op=merge&mergepolicy=aggressive
```

forces a merge (if one is needed) and changes the current merge policy to an aggressive policy.

## Using the URL updatehistory command

The `updatehistory` URL command returns the list of update files processed since this Dgraph was started for the first time.

This command has no options.

**Note:** This command does not track the history of empty update files.

To display the update history:

- Use the Dgraph `updatehistory` URL command, similar to the following example:

  ```
  http://localhost:8000/admin?op=updatehistory
  ```

The command displays a "Endeca Dgraph Server update directory history contents" page with results similar to this example:

```
Checking history for update directory for directory "C:\Endeca\Apps\wine\da¬
ta\dgraphs\Dgraph1\dgraph_input\updates"

          Files in update directory history
"updates/wine-sgmt0.records.xml_2010.07.12.16.29.28"
```

Chapter 4

# Partial Updates Pipeline

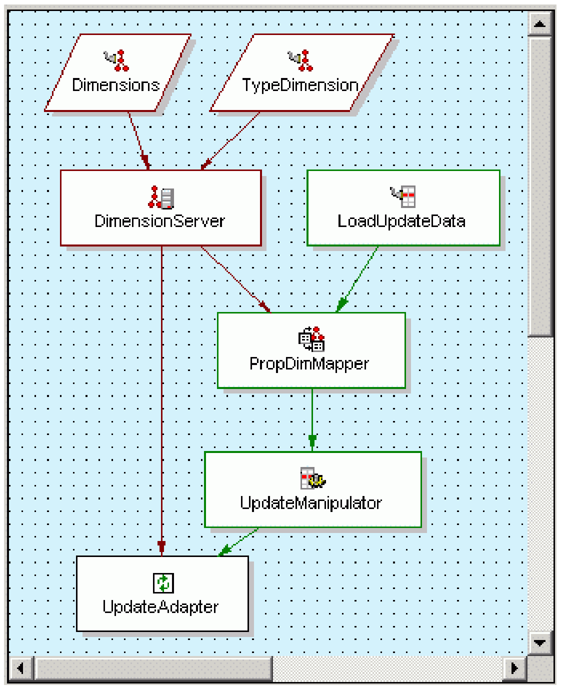
This section describes how to create a partial updates pipeline.

## About the partial update pipeline

A partial update requires its own pipeline (separate from the baseline update pipeline) that only deals with partial updates. Use Developer Studio to create the partial update pipeline.

Each input record in a partial update pipeline describes a transformation to be performed on a single record in the running application. This means, for example, that a single update cannot change the spelling of a property on many records; instead, a separate update must be generated to change the spelling on each record in the application.

The following partial update pipeline is used as an example:



The partial update pipeline is executed at frequent intervals. Between runs, updates are queued. When the partial update process starts, all the queued updates are processed and written to a staging area. When Forge is complete, the updates are read from the staging area into the running application.

The sample partial update pipeline works as follows:

1. The partial update pipeline reads its input, using a record adapter (named LoadUpdateData) with the Multi File field checked.
2. The input records are transformed into record updates by a record manipulator (named UpdateManipulator) using IF and `UPDATE_RECORD` expressions.

   **Note:** In this diagram, the dimension server is not connected to the record manipulator (called the UpdateManipulator in the diagram). While this configuration of the partial update pipeline works for the pipeline used here as an example, in cases when your partial update pipeline is updating dimension values, you also must connect the dimension server to the record manipulator that handles the transformations for the record updates. This way, the record manipulator will know with which dimensions it is working.

3. The record updates are written out to the `dgraph_input/updates` directory (or another directory that you specify) using an update adapter. After the files are applied to the MDEX Engine, they are removed from this directory. Therefore, it is important to back up the update files to another parallel directory, in case you want to replay them against the data in the MDEX Engine.

## Configuring a partial update pipeline

To configure a project for partial updates, create a separate partial update pipeline.

This pipeline can be based on the existing baseline pipeline, although it requires its own components. One of the ways to start creating a partial update pipeline is to copy your existing baseline update pipeline and modify it. If you copy the baseline update pipeline that uses its own record adapter, cache, and assembler components, remove these components in your partial update pipeline.

This section lists high-level tasks required to create a partial update pipeline. For information on a specific task, such as adding a new record manipulator component, see the related sections in this chapter.

To configure a partial update pipeline:

1. Add a new record adapter component. Its purpose is to load only the updates that occurred since the last baseline update.
2. Add a new record manipulator component configured specifically for the partial update pipeline. The record manipulator decides whether the record is going to be added, replaced, updated or removed.
3. Add a new update adapter component. The update adapter instructs Forge where to temporarily place the update-related processed data files (such as the `dgraph_input/updates` directory). These files are removed after being applied to the MDEX Engine. Next, the update adapter writes out the record file(s) that define the new, removed, or modified records.
4. Add additional dimension components, if you are updating dimension values. Ensure that you have a dimension server in your partial update pipeline that is connected to the record manipulator and the update adapter.

## Creating the record adapter

You start building your partial update pipeline by adding a new record adapter component. Its purpose is to load only the updates that occurred since the last baseline update.

**Note:** The following procedure for configuring the record adapter is specific to an example where a record adapter is configured for a file-based record source with multiple wildcard-matching files. Your partial update pipeline may not necessarily contain a record source like this, but you still need to configure a record adapter to load the updates that occurred since the last baseline update.

To configure the record adapter:

1. In Developer Studio, specify the following basic settings in the **General** tab of the **Record Adapter** editor:

| Option | Description |
|---|---|
| **Direction** | Must be **Input**. |
| **URL** | Enter an input URL as a path, with the filename being a pattern.<br><br>For example, a URL pattern of `../incoming/updates/*.txt.gz` means that Forge will read any file in the `incoming/updates` directory that has the `txt.gz` suffix. Each file that matches the pattern will be read in strict lexicographic order of their filenames. |
| **Multi File** | Select this option to specify that Forge can read data from more than one input file and that the input URL is to be interpreted as a pattern. |

2. You can leave the other tabs (**Sources**, **Record Index**, and so on) in their default state.

## Creating the record manipulator

The record manipulator in the partial update pipeline examines whether the record is going to be added, replaced, updated or removed.

To configure the record manipulator:

1. In Developer Studio, specify the following settings in the **Sources** tab of the **Record Manipulator** editor:

| Option | Description |
|---|---|
| **Record source** | Select the name of the property mapper. |
| **Dimension source** | Select a dimension server if the record manipulator is performing any `DIM_ACTION` or `DVAL_ACTION` operations; if not, you can select **None** (as in the sample pipeline). |
| **Record Index** | You can leave the tab empty. |

2. In the **Expression** editor, add expressions similar to those described in "About the IF expression for the record manipulator." You open the **Expression** editor by double-clicking the record

manipulator component in the **Pipeline Diagram.** You can add expressions after the record manipulator is created.

## About the IF expression for the record manipulator

The record manipulator used in a partial update pipeline is essentially an `IF` expression that calls one of three `UPDATE_RECORD` expressions based on a conditional evaluation of the incoming record.

**Note:** The following sample code shows one example of what an expression for the record manipulator might look like. Depending on the field values in your data, the logic that you add to your expressions may be different from this example.

The `IF` expression is coded as follows:

```
<RECORD_MANIPULATOR FRC_PVAL_IDX="TRUE" NAME="UpdateManipulator">
    <RECORD_SOURCE>PropDimMapper</RECORD_SOURCE>
    <EXPRESSION LABEL="" NAME="IF" TYPE="VOID" URL="">
  <COMMENT>
- If the record has a "Remove" field value equal to "1", then delete it.
- If the record has an "Update" field value equal to "1", then update it
- Otherwise, add the new record.
  </COMMENT>
      <EXPRESSION LABEL="" NAME="MATH" TYPE="INTEGER" URL="">
        <EXPRNODE NAME="TYPE" VALUE="STRING"/>
        <EXPRNODE NAME="OPERATOR" VALUE="EQUAL"/>
        <EXPRESSION LABEL="" NAME="IDENTITY" TYPE="PROPERTY" URL="">
          <EXPRNODE NAME="PROP_NAME" VALUE="Remove"/>
        </EXPRESSION>
        <EXPRESSION LABEL="" NAME="CONST" TYPE="STRING" URL="">
          <EXPRNODE NAME="VALUE" VALUE="1"/>
        </EXPRESSION>
      </EXPRESSION>
      <EXPRESSION LABEL="" NAME="UPDATE_RECORD" TYPE="VOID" URL="">
        <EXPRNODE NAME="ACTION" VALUE="DELETE_OR_IGNORE"/>
      </EXPRESSION>
      <EXPRNODE NAME="ELSE_IF" VALUE=""/>
      <EXPRESSION LABEL="" NAME="MATH" TYPE="INTEGER" URL="">
        <EXPRNODE NAME="TYPE" VALUE="STRING"/>
        <EXPRNODE NAME="OPERATOR" VALUE="EQUAL"/>
        <EXPRESSION LABEL="" NAME="IDENTITY" TYPE="PROPERTY" URL="">
          <EXPRNODE NAME="PROP_NAME" VALUE="Update"/>
        </EXPRESSION>
        <EXPRESSION LABEL="" NAME="CONST" TYPE="STRING" URL="">
          <EXPRNODE NAME="VALUE" VALUE="1"/>
        </EXPRESSION>
      </EXPRESSION>
      <EXPRESSION LABEL="" NAME="UPDATE_RECORD" TYPE="VOID" URL="">
        <EXPRNODE NAME="ACTION" VALUE="UPDATE"/>
        <EXPRNODE NAME="PROP_ACTION" VALUE="REPLACE"/>
        <EXPRNODE NAME="PROP_NAME" VALUE="P_Price"/>
      </EXPRESSION>
      <EXPRNODE NAME="ELSE" VALUE=""/>
      <EXPRESSION LABEL="" NAME="UPDATE_RECORD" TYPE="VOID" URL="">
        <EXPRNODE NAME="ACTION" VALUE="ADD_OR_REPLACE"/>
      </EXPRESSION>
    </EXPRESSION>
  </RECORD_MANIPULATOR>
```

## About the UPDATE_RECORD expression

The `UPDATE_RECORD` expression updates existing records by adding, removing, or replacing dimensions, dimension values, or property values. The expression can also delete existing records and add new ones.

If different types of partial updates are processed using separate pipelines, the `UPDATE_RECORD` expression can be written to perform the same action on all of the input.

For example, a partial update pipeline written to handle only price and availability changes would always generate `UPDATE`-type record updates. If the same partial update pipeline needs to handle `REPLACE` updates (that is, reclassification of a record), the input data must contain some indication of what type of update to perform. Most commonly, this will simply be a property on the input record, which is checked inside an `IF` expression.

The `UPDATE_RECORD` expression takes a snapshot of the record at the time it is evaluated and generates a corresponding record update. Thus, the update contains the property names and values, as well as classifications, that are in effect at the time of evaluation. If properties are renamed, have their values changed, or classifications are added or deleted after the record update expression has been evaluated, the changes have no impact on the record update that will be generated. Only one record update can be generated per record.

Note the following:

- For `ADD` record updates, a complete record must be set up before the expression is evaluated.
- For `REPLACE` record updates, all the necessary property values and dimension values (as well as the property specifying the `RECORD_SPEC`) must be on the record.
- For `ADD_OR_REPLACE` record updates, if no record exists with the specified property value for the property that has been designated as the `RECORD_SPEC`, the system adds a new record. If the record exists, it is replaced.
- For `DELETE` record updates, the `RECORD_SPEC` property must be on the record. This property is used to identify the record to be deleted. All other properties and dimension values are ignored.
- For `DELETE_OR_IGNORE` record updates, if a record exists with the specified property value for the property that has been designated as the `RECORD_SPEC`, the system removes the record. If the record does not exist, the action is ignored and no error message is generated.
- For `UPDATE` record updates, further specification is necessary to describe how to handle the property values and dimension values on the record. `UPDATE`-type record updates must also include the `RECORD_SPEC` property with each record. Each property or dimension can have only one type of update performed, but a single record update may impact any or all of the properties and dimensions on a record.

**Related Links**

*Obtaining IDs for the DIM_ACTION and DVAL_ACTION expression nodes* on page 33
The `DIM_ACTION` expression node requires the use of a `DIMENSION_ID` node to specify the numeric ID of the dimension to be modified.

*Expression nodes supported by the UPDATE_RECORD expression* on page 32
This section provides a reference table of all expression nodes supported by the `UP¬DATE_RECORD` expression.

*UPDATE_RECORD expression reference examples* on page 34
This section provides reference examples of the `UPDATE_RECORD` expression.

## Expression nodes supported by the UPDATE_RECORD expression

This section provides a reference table of all expression nodes supported by the `UPDATE_RECORD` expression.

| EXPRNODE name | Description |
|---|---|
| `ACTION` | The type of action to perform on the record, as indicated by the `VALUE` attribute. Valid values for this attribute are:<br>• `ADD` – Adds a new record if it does not exist, or generates an error message if it already exists.<br>• `ADD_OR_REPLACE` – Adds a new record if it does not exist, or replaces it if it already exists.<br>• `REPLACE` – Replaces a record if it exists, or generates an error message if it does not exist.<br>• `DELETE` – Removes a record if it exists, or generates an error message if it does not exist.<br>• `DELETE_OR_IGNORE` – Removes a record if it exists, but does not generate an error message if it does not exist.<br>• `UPDATE` – Updates a record if it exists, or generates an error message if it does not exist.<br>Examples:<br>`<EXPRNODE NAME="ACTION" VALUE="UPDATE"/>`<br>`<EXPRNODE NAME="ACTION" VALUE="ADD_OR_REPLACE"/>` |
| `PROP_ACTION` | If `ACTION=UPDATE`, the `VALUE` attribute specifies the type of update to perform on all the values of the named property. Valid values for this attribute are as follows:<br>• `ADD` – All values for the property on the update record are added to the current record.<br>• `DELETE` – All values for the property on the update record are removed from the current record.<br>• `REPLACE` – All values for the property are removed from the current record, then all values for the property on the update record are added to the current record. This node must be followed by a `PROP_NAME` expression node that names the property to be modified. For example:<br>`<EXPRNODE NAME="PROP_ACTION" VALUE="REPLACE"/>`<br>`<EXPRNODE NAME="PROP_NAME" VALUE="P_WineType"/>` |
| `DIM_ACTION` | If `ACTION=UPDATE`, the `VALUE` attribute specifies the type of update to perform on all the values of the named dimension. Valid values for this attribute are as follows:<br>• `ADD` – All dimension values in the dimension on the update record are added to the current record. |

| EXPRNODE name | Description |
|---|---|
| | • `DELETE` – All dimension values in the dimension on the update record are removed from the current record. **Note:** Previously-existing parental dimension values are also removed upon a delete operation. For example, assume you have a parent dimension value (dval) with `id=1`, a child dval with `id=2`, and a record with dval 1 assigned. With the update operation, you first add dval 2 to the record (it replaces dval 1, since it is more specific), and then remove dval 2. The record now has no dvals attached to it, despite its initial assignment to dval 1 which was never explicitly deleted. The parental dval is removed at the time when a child dval is added. However, this change is not visible until the child dval is deleted, at which point no dvals remain on the record. • `REPLACE` – All dimension values in the dimension are removed from the current record, then all dimension values in the dimension on the update record are added to the current record. This node must be followed by a `DIMENSION_ID` expression node that specifies the numeric ID of the dimension to be modified. For example: `<EXPRNODE NAME="DIM_ACTION" VALUE="ADD"/>` `<EXPRNODE NAME="DIMENSION_ID" VALUE="8000"/>` |
| `DVAL_ACTION` | `If ACTION=UPDATE`, removes the dimension value from the record. Note that the `VALUE` attribute only supports `DELETE`. This node must be followed by a `DVAL_ID` expression node that specifies the numeric ID of the dimension value to be removed. For example: `<EXPRNODE NAME="DVAL_ACTION" VALUE="DELETE"/>` `<EXPRNODE NAME="DVAL_ID" VALUE="8031"/>` |

**Related Links**

*UPDATE_RECORD expression reference examples* on page 34
This section provides reference examples of the `UPDATE_RECORD` expression.

*Obtaining IDs for the DIM_ACTION and DVAL_ACTION expression nodes* on page 33
The `DIM_ACTION` expression node requires the use of a `DIMENSION_ID` node to specify the numeric ID of the dimension to be modified.

*About the UPDATE_RECORD expression* on page 31
The `UPDATE_RECORD` expression updates existing records by adding, removing, or replacing dimensions, dimension values, or property values. The expression can also delete existing records and add new ones.

## Obtaining IDs for the DIM_ACTION and DVAL_ACTION expression nodes

The `DIM_ACTION` expression node requires the use of a `DIMENSION_ID` node to specify the numeric ID of the dimension to be modified.

Similarly, the `DVAL_ACTION` expression node requires a `DVAL_ID` node to specify the numeric ID of the dimension value to be modified.

To obtain IDs for the `DIM_ACTION` and `DVAL_ACTION` expression nodes:

- To obtain the dimension IDs for the `DIM_ACTION` expression, use the **Dimension editor** in Developer Studio.
- To obtain the IDs for the `DVAL_ACTION` expression, open the `Dimensions.xml` configuration file with a text editor and look for the specific dimension value.

For example, consider this dimension named Designation:

```
<DIMENSION NAME="Designation" SRC_TYPE="INTERNAL">
 <DIMENSION_ID ID="7"/>
 <DIMENSION_NODE>
  <DVAL TYPE="EXACT">
  <DVAL_ID ID="7"/>
  <SYN CLASSIFY="FALSE" DISPLAY="TRUE"
    SEARCH="FALSE">Designation</SYN>
  </DVAL>
   <DIMENSION_NODE>
    <DVAL TYPE="EXACT">
    <DVAL_ID ID="8031"/>
    <SYN CLASSIFY="TRUE" DISPLAY="TRUE"
      SEARCH="TRUE">Best Buy</SYN>
    </DVAL>
   </DIMENSION_NODE>
...
 </DIMENSION_NODE>
</DIMENSION>
```

The dimension `ID` is `7`, while the ID of the dimension value named `Best Buy` is `8031`. If you want to use a `DVAL_ACTION` expression node to modify the Best Buy dimension value, the corresponding a `DVAL_ID` expression node would use a value of `8031`.

**Related Links**

*About the UPDATE_RECORD expression* on page 31
The `UPDATE_RECORD` expression updates existing records by adding, removing, or replacing dimensions, dimension values, or property values. The expression can also delete existing records and add new ones.

*Expression nodes supported by the UPDATE_RECORD expression* on page 32
This section provides a reference table of all expression nodes supported by the `UPDATE_RECORD` expression.

*UPDATE_RECORD expression reference examples* on page 34
This section provides reference examples of the `UPDATE_RECORD` expression.

## UPDATE_RECORD expression reference examples

This section provides reference examples of the `UPDATE_RECORD` expression.

**Example 1**

An expression configured to convert input records to `ADD_OR_REPLACE RECORD` updates:

```
<EXPRESSION TYPE="VOID" NAME="UPDATE_RECORD">
 <EXPRNODE NAME="ACTION" VALUE="ADD_OR_REPLACE"/>
</EXPRESSION>
```

**Example 2**

An expression configured to convert input records to replace the Price property, and the price range and availability classifications:

```
<EXPRESSION TYPE="VOID" NAME="UPDATE_RECORD">
 <EXPRNODE NAME="ACTION" VALUE="UPDATE"/>
 <EXPRNODE NAME="PROP_ACTION" VALUE="REPLACE"/>
 <EXPRNODE NAME="PROP_NAME" VALUE="Price"/>
 <EXPRNODE NAME="DIM_ACTION" VALUE="REPLACE"/>
 <EXPRNODE NAME="DIMENSION_ID" VALUE="100"/><!--100=PriceRange-->
 <EXPRNODE NAME="DIM_ACTION" VALUE="REPLACE"/>
 <EXPRNODE NAME="DIMENSION_ID" VALUE="200"/><!--200=Availability-->
</EXPRESSION>
```

**Related Links**

*Expression nodes supported by the UPDATE_RECORD expression* on page 32
This section provides a reference table of all expression nodes supported by the UP¬DATE_RECORD expression.

*Obtaining IDs for the DIM_ACTION and DVAL_ACTION expression nodes* on page 33
The DIM_ACTION expression node requires the use of a DIMENSION_ID node to specify the numeric ID of the dimension to be modified.

*About the UPDATE_RECORD expression* on page 31
The UPDATE_RECORD expression updates existing records by adding, removing, or replacing dimensions, dimension values, or property values. The expression can also delete existing records and add new ones.

## Format of update records

The UPDATE_RECORD expression, as used in the sample partial update pipeline, requires that each incoming record have one of the Delimited formats described below.

### Format of records to be deleted

The first column in the header row must be a Remove column. The first column in each record must have a value of 1 to delete the record:

```
Remove|P_WineID|P_Year|P_Wine|P_Winery|...|
1|34699|1992|A Red Blend Alexander Valley|Lyeth|...|
```

### Format of records to be updated

The first column in the header row must be an Update column. The first column in each record must have a value of 1 to update the record properties:

```
Update|P_WineID|P_Wine|P_PriceStr|
1|34701|Albarino Rias Baixas|1000.00|
```

### Format of records to be added

The header row of records to be added do not begin with a Remove or Update column. Instead, they use the normal set of header row columns (`P_WineId`, `P_Year`, and so on). The first column in each record has a normal property value:

```
P_WineID|P_Year|P_Wine|P_Winery|P_PriceStr|...|
99000|1992|First New Wine Added|Lyeth|18.00|...|
```

**Format of records in your implementation**

If your implementation uses Delimited format records, you can use the above format to specify how the records are handled. If you use another format, you must use a record manipulator with the appropriate expressions to handle your source records.

## Dimension components

The sample partial update pipeline contains two dimension adapters and one dimension server.

### Dimension adapters

To support classification, the same dimensions that are loaded in the baseline update pipeline must be loaded in the partial update pipeline. To cut down on startup time, the dimensions can be split into multiple files, and only the dimensions actually used by the partial update pipeline need to be loaded. In the baseline update pipeline, multiple dimension adapters can feed into the same dimension server to consolidate the separate dimension files.

The sample pipeline uses two dimension adapters, one for the `dimensions.xml` file and the other for the `winetype_dimension.xml` file. For both dimension adapters, the **Dimension Source** field (on the **Sources** tab) is set to **None**.

### Dimension server

The dimension server uses the two dimension adapters as sources.

The **URL** field (**General** tab) specifies the location to which the `autogen_dimensions.xml.gz` file is written. This file contains persistent dimension data produced by auto-generation and also data produced by the record to dimension adapter.

There are special considerations when using AutoGen classification with partial updates. When new dimension values are generated in the partial update pipeline, the dimension changes are included in the updates sent to the MDEX Engine.

Because the baseline and partial update pipelines share the same autogen file, changes to `Autogen_dimensions.xml` are also shared between the two. However, at any given time, only one of the two update processes can modify the `Autogen_dimensions.xml` file.

Rather than suspend partial updates during baseline updates, Forge supports the `--noAutoGen` command-line option, which turns off the creation of new dimension values. Classification with existing dimension values continues normally, but classification failures result in no matching dimension values, rather than in the creation of new ones.

## Naming format of update source data files

When Forge processes update source data files, it is important to keep two issues in mind concerning the names of the data files.

- The update files should be processed by Forge in order of their creation. The reason is that if a specific record appears in more than one update file, you want the latest update to be processed last, so that it will override earlier versions when the Dgraph loads the update record files.
- Forge reads the files in strict lexicographic order of their filenames. Therefore, you should use a naming scheme that ensures the processing of the update files in chronological order of their creation (i.e., last created, last processed).

For these reasons, it is strongly recommended that you use a timestamp format as the naming scheme for the filenames. If necessary, use leading zeros to force the desired numeric order. For example, if you have two files named `9.xml` and `10.xml`, Forge will process `10.xml` before `9.xml`; therefore, you must rename `9.xml` to `09.xml` so that it is processed before `10.xml`.

**Related Links**

*Naming convention for source data files* on page 42
Whether you are using a random or deterministic distribution strategy, it is strongly recommended that you use a timestamp format as the naming scheme for the update source data files.

*Naming format of partial update files* on page 37
When Forge generates partial update files, they need to be named in a manner that allows the MDEX Engine to read them in the right order.

## Naming format of partial update files

When Forge generates partial update files, they need to be named in a manner that allows the MDEX Engine to read them in the right order.

The MDEX Engine reads update files that it receives in numeric-lexicographic order of their filenames.

Therefore, the scripts that you use for partial updates should rename the Forge output files with a timestamp. In other words, the scripts should name update files in ascending numeric-lexicographic order over time to ensure that updates are processed by the MDEX Engine in the order they are produced by Forge.

**Related Links**

*Naming convention for source data files* on page 42
Whether you are using a random or deterministic distribution strategy, it is strongly recommended that you use a timestamp format as the naming scheme for the update source data files.

*Naming format of update source data files* on page 37
When Forge processes update source data files, it is important to keep two issues in mind concerning the names of the data files.

*Examples of numeric-lexicographic and simple lexicographic order* on page 38
While the MDEX Engine reads files in numeric-lexicographic order, Forge reads them in simple lexicographic order. Keep this difference in mind when naming files.

## Examples of numeric-lexicographic and simple lexicographic order

While the MDEX Engine reads files in numeric-lexicographic order, Forge reads them in simple lexicographic order. Keep this difference in mind when naming files.

The following examples illustrate the ordering modes:

- Simple lexicographic order is the order in which Forge reads partial update files. Using this order, Forge compares the file names lexicographically. For example, when comparing `5.txt` and `10.txt`, “5” is compared with “1”. Based on this comparison, Forge first reads `10.txt` and then `5.txt`.
- Numeric-lexicographic order is the order in which the MDEX Engine reads partial update files. Using this order, the MDEX Engine breaks a file name into a numeric prefix and a non-numeric suffix, and compares the numeric prefixes numerically. It breaks ties in numeric prefixes by proceeding to compare suffixes lexicographically. For example, when comparing `10hello.txt`, `010jello.txt`, and `5z.txt`, “10” is compared with “010” and “5” numerically. This identifies `5z.txt` as the file name that should be ordered first. To resolve the tie between “10” and “010”, “h” is compared with “j”. As a result, `5z.txt` is processed first, `10hello.txt` is processed next, and `010jello.txt` is processed last.

**Related Links**

*Naming format of partial update files* on page 37

When Forge generates partial update files, they need to be named in a manner that allows the MDEX Engine to read them in the right order.

# Index configuration in the partial update pipeline

Using the partial updates, you can update only records and dimensions. You cannot update the index configuration files, such as the thesaurus and stop words files.

Chapter 5

# Partial Updates in Agraph Implementations

This section describes how to run partial updates in Endeca implementations that use the Endeca Aggregated MDEX Engine (Agraph).

## About Agraph implementations with partial updates

Implementing partial updates in Agraph implementations is similar to single-Dgraph deployments, with the important differences listed in the following sections.

For examples of the control scripts used for partial updates in the Agraph implementations, see the *Endeca Control System Guide*.

**Note:** Control scripts are deprecated.

## The Agraph and continuous query support

Starting with the MDEX Engine version 6.1.2, the Agraph supports the MDEX Engine feature known as continuous query.

The following statements describe how the Agraph supports continuous query:

- The Agraph can continually answer queries to its clients even while partial updates are applied across its different child Dgraphs. This eliminates the need to stop the Agraph when applying partial updates to the Dgraphs.
- Although an Agraph does not wait for all of its child Dgraphs to finish updating when querying them, it always ensures that it uses consistent results from a single child Dgraph to which partial updates are being applied.

  A query result is always returned by the Agraph after it aggregates the child Dgraph results, without specific guarantees that all child results reflect the partial updates.
- If a deployment requires consistent results from all child Dgraphs following a partial update, Endeca recommends to run multiple Agraphs (each with its set of child Dgraphs) within a load-balancer pool. This allows selectively removing an Agraph from the pool, and letting pending requests to complete before applying the updates to all its child Dgraphs. Once updates have been applied to the child Dgraphs, you can add the Agraph back to the load-balancer pool.

In Agraph implementations that utilize a load balancer between a single Agraph and its Dgraphs, if a deployment requires consistent results from all child Dgraphs following a partial update, ensure that re-queries made by an Agraph target the same child Dgraph. You can achieve this by configuring the load balancer to track session information, and ensuring that all requests associated with a session go to the same child Dgraph.

## Choosing a distribution strategy

The update record files produced by Forge contain XML definitions of the updated records, including information on how the records should be treated by the Dgraphs.

For example, records to be deleted are flagged with a `RECORD_DELETE` element in the file.

New records (that is, the ones that use an `ADD` or `ADD_OR_REPLACE` action for the `UPDATE_RECORD` expression) are defined with a `RECORD_ADD` element that contains the partition number (in a `PARTITION` attribute) to which the record is assigned. Both `ADD` and `ADD_OR_REPLACE` records are assigned partition numbers.

For example, this XML snippet shows a `RECORD_ADD` element that assigns a new record to Agraph partition1:

```
<UPDATE>
 <UPD_UNIT>
  <RECORD_ADD PARTITION="1">
   <PROP NAME="P_WineID">
    <PVAL>99005</PVAL>
   </PROP>
   <PROP NAME="P_Year">
    <PVAL>1992</PVAL>
   </PROP>
   ...
  </RECORD_ADD>
 </UPD_UNIT>
...
</UPDATE>
```

How the partition number is assigned to a new record depends on which distribution strategy you have chosen to implement:

- Random distribution, where you let Forge decide which partition gets the new record. That is, Forge uses the configured partition property (typically the record spec or rollup property) as a basis of assigning the partition number to the `PARTITION` attribute.
- Deterministic distribution, where you control the assignment of records to specific partitions. That is, you tell Forge which partition number it should assign to the `PARTITION` attribute.

The main advantage of random distribution is that you do not need to know exactly where the records should go in order for updates to be processed correctly. This scheme also simplifies operations because the same update record file is sent to all partitions, so there is less conditional logic in your script.

Which distribution strategy you chose depends on the needs of your implementation. In general, Endeca recommends that the distribution strategy for partial updates be the same as for baseline updates.

## How the Agraph partitions handle updates

Regardless of which distribution strategy you are using, the Agraph partitions (that is, the individual Dgraphs) handle the record update requests in one of two ways.

- If a `DELETE`, `REPLACE`, or `UPDATE` action request is sent to all partitions of the Agraph, only the partition that contains the record actually deletes, replaces, or updates the record. The other partitions issue a warning message, but continue to function as before.
- If an `ADD` or `ADD_OR_REPLACE` action request is sent to all partitions, only the designated partition (as specified in the `PARTITION` attribute of the record file) will add the record. The other partitions ignore the request.

Because any partition knows how to deal with any update request, this architecture allows you to send the record files to all partitions without having to worry about which partition is the correct one.

To summarize, record updates can be sent to all Dgraphs, although they have to be sent only to the specific partition to which they apply. Dimension updates must be distributed to all partitions. Since both types of updates typically are combined in a single file (that is, the partial update pipeline generates one output file from Forge, with both record and dimension updates), the typical recommended strategy is to distribute this file to every partition, ensuring that dimensions go everywhere and relying on the partition to read its record updates and ignore irrelevant record updates that apply to other partitions.

## About distributing the Forge output to the Dgraphs

For a random distribution strategy, partial updates in Agraph implementations do not require any special update distribution requirements. Both dimension modifications (such as, dimension value additions) and record modifications (updates, deletes, replaces, and adds) should be sent to all Dgraphs in the deployment.

Each Dgraph should then be notified to check for new updates. If a Dgraph cannot handle data that is associated with another Dgraph, it will simply log a warning but will otherwise continue working. The Agraph process itself does not process updates.

For a deterministic distribution strategy, the distribution of the record files depends on the use of auto-generated dimensions as follows:

- If you are using auto-generated dimensions, distribute all the record files to all the Dgraphs.
- If you are not using auto-generated dimensions, you can distribute each record file to its specific Dgraph.

To make sure that there is no interruption in servicing navigation requests, you may configure your Dgraphs to check for new updates at different times. Or you can also have smaller subgroups read in updates simultaneously (for example, three Dgraphs at a time in a six-Dgraph implementation).

## Use of the record specifier attribute

The RECORD_SPEC attribute in an Agraph deployment requires that the record property be unique across all records across all MDEX Engines.

## Naming convention for source data files

Whether you are using a random or deterministic distribution strategy, it is strongly recommended that you use a timestamp format as the naming scheme for the update source data files.

This format ensures that Forge processes the files in the proper order of their creation.

For both strategies, a Perl expression in the record manipulator can use the timestamp part of the filename for the name of the output record file.

### Random distribution format

For a random distribution strategy, a suggested format is:

```
YYYYMMDDHHNNSS.ext
```

where YYYY is the four-digit year, MM is the two-digit month, DD is the two-digit day, HH is the two-digit hour, NN is the two-digit minute, and SS is the two-digit second, as this example:

```
20051023161408.txt
```

These files may contain new records that are distributed randomly to the Agraph partitions.

### Deterministic distribution format

For a deterministic distribution strategy, a suggested format is:

```
YYYYMMDDHHNNSS-partX.ext
```

where X is the number of the Agraph partition for which these records are intended. For example, records in this source data file are intended for partition3:

```
20050717151408-part3.txt
```

The Perl expression in the record manipulator parses the filename for the partition number and uses it to assign new records to that partition.

The expression also uses the timestamp and `-partX` information for the name of the output record file. For example, the above input filename generates this output record file:

```
20050717151408-part3.records.xml
```

Keep in mind that if you pre-partition your baseline source files, you should also pre-partition the records to be added. That is, all `ADD` (or `ADD_OR_REPLACE`) records for the partition `0` Dgraph should be in one file, records for the `partition1` Dgraph should be in a second file, and so on.

**Related Links**

*Naming format of update source data files* on page 37
When Forge processes update source data files, it is important to keep two issues in mind concerning the names of the data files.

*Naming format of partial update files* on page 37
When Forge generates partial update files, they need to be named in a manner that allows the MDEX Engine to read them in the right order.

## Configuring the partial update pipeline

This section describes how to configure the partial update pipeline for either distribution strategy.

**Note:** The procedures described in this section require that you hand-edit the pipeline files with a text editor. After you edit these files, do not open the project in Developer Studio, because it will overwrite the settings of the update adapter.

## Configuring the record adapter

The record adapter in the partial update pipeline that uses an Agraph loads the data files that are updated.

To configure the record adapter:

In the record adapter, specify the following settings:

| Option | Description |
|---|---|
| **MULTI** | Set to `True` so that Forge can read multiple input data files. |
| **URL** | Set the attribute to the path of the incoming directory, with the filename being a pattern (such as `../incoming/updates/*.txt`). |
| **MULTI_PROP_NAME** | In order to use the naming format of the input file for the records file name, set this attribute to a value of `FILENAME`. |

These settings apply to both random and deterministic record adapters.

The following is an example of a record adapter for the partial update pipeline:

```
<RECORD_ADAPTER
  NAME="LoadUpdateData"
  URL="../incoming/updates/*.txt"
  FORMAT="DELIMITED"
  COL_DELIMITER="|"
  ROW_DELIMITER="|\n"
  DIRECTION="INPUT"
  FILTER_EMPTY_PROPS="TRUE"
  FRC_PVAL_IDX="TRUE"
  MULTI="TRUE"
  MULTI_PROP_NAME="FILENAME"
  REQUIRE_DATA="FALSE"
</RECORD_ADAPTER>
```

The `FILENAME` setting for the `MULTI_PROP_NAME` attribute is processed by both the update adapter and the Perl expression in the record manipulator.

## About configuring the record manipulator

For both random and deterministic pipelines, the record manipulator should contain the same `IF` and `UPDATE_RECORD` expressions that are documented for the single-Dgraph implementation.

In addition, you can add a Perl expression that parses the name of each input file (up to the file extension) and uses it to name the output record file (which has a `records.xml` extension). The exact Perl code in the expression depends on the distribution strategy.

**Related Links**

*Perl expression for random distribution* on page 44
For a random distribution pipeline, a Perl expression can be inserted into the record manipulator.

*Perl expression for deterministic distribution* on page 44
For a deterministic distribution pipeline, a Perl expression can be inserted into the record manipulator.

## Perl expression for random distribution

For a random distribution pipeline, a Perl expression can be inserted into the record manipulator.

For example:

```
<EXPRESSION TYPE="VOID" NAME="PERL">
 <COMMENT>
 This Perl expression handles taking the source input
 filename and outputting a record file with the same
 naming format.
 It assumes filenames of the format: timestamp.ext
 </COMMENT>
 <EXPRBODY><![CDATA[
  # Translate filename of input to filename of output.
  # Filename is everything after the last slash
if ($props[0]->value() =~ /[\/\\]((\w+[\.])*(\w+))\.[^\.]+$/ {
   my $filename = $1;
   $props[0]->value($filename);
   replace_prop("FILENAME", 0, $props[0]);
  } else {
    die("Could not parse filename: " . $props[0]->value());
  }
  ]]>
 </EXPRBODY>
</EXPRESSION>
```

The expression generates output record files with names similar to this example:

```
20050717180812.records.xml
```

This sample expression is for use on both Windows and UNIX platforms.

**Note:** Keep in mind that you must change the Perl regex code if you use another naming convention for the source input files.

## Perl expression for deterministic distribution

For a deterministic distribution pipeline, a Perl expression can be inserted into the record manipulator.

The Perl expression for the record manipulator in a deterministic distribution pipeline is similar to the random distribution example, with the addition of code that extracts the partition ID (the `partX` piece) from the input filename and stores it in the `X_PartitionNum` property. The partition ID will be assigned by Forge to that record in the record file (via the `PARTITION` attribute of the `ADD_RECORD` element).

The Perl expression is as follows:

```
<EXPRESSION TYPE="VOID" NAME="PERL">
 <COMMENT>
 This Perl expression handles taking the source input filename and
 determining the appropriate partition. It assumes filenames of the
 format: timestamp-partN.ext
 The expression extracts the N in the "partN" piece.
 </COMMENT>
 <EXPRBODY><![CDATA[
  # Translate filename of input to filename of output.
  my @props = get_props_by_name("FILENAME");
  # Filename is everything after the last slash
  if ($props[0]->value() =~ /[\/\\]((\w+[\.])*(\w+))\.[^\.]+$/ {
    my $filename = $1;
    $props[0]->value($filename);
    replace_prop("FILENAME", 0, $props[0]);
    # Extract the partition ID from the filename to determine
    # the partition number for the record.
    $filename =~ /part(\d+)$/;
    my $part_num = $1;
    # X_PartitionNum specifies the target partition for
    # this particular record.
    my $part_prop = new Zinc::PropVal("X_PartitionNum", $part_num);
    add_props($part_prop);
  } else {
    die("Could not parse filename: " . $props[0]->value());
  }
  ]]>
 </EXPRBODY>
</EXPRESSION>
```

This sample expression is for use on both Windows and UNIX platforms.

**Related Links**

## Configuring the update adapter

The configuration of the update adapter in a partial update pipeline that uses the Agraph is similar to that in single-Dgraph implementations.

To configure the update adapter:

1. Specify the following settings for the update adapter, for both random and deterministic distribution:

| Option | Description |
|---|---|
| **OUTPUT_URL** | Set to the path of the incoming directory, with the filename being a pattern. |
| **OUTPUT_PREFIX** | Set to an empty string, because the output filename begins with a timestamp format. |
| **MULTI** | Set to `True` so that Forge can read multiple input data files. |
| **MULTI_PROP_NAME** | Set to a value of `FILENAME`. |

2. Specify the settings for the `ROLLOVER` element depending on the type of the distribution strategy.

**Related Links**

*ROLLOVER element for random distribution* on page 46
In a random distribution pipeline, the are five recommended settings for the `ROLLOVER` element.

*ROLLOVER element for deterministic distribution* on page 47
In a deterministic distribution pipeline, the are five recommended settings for the `ROLLOVER` element.

## ROLLOVER element for random distribution

In a random distribution pipeline, the are five recommended settings for the `ROLLOVER` element.

| Option | Description |
|---|---|
| `NUM_IDX` | Although this attribute normally sets the number of Agraph partitions, it is recommended that you use the Forge `--numPartitions` flag for the Forge component to actually set the number of partitions. Therefore, leave the field blank or use any number. |
| `PROP_NAME` | Set to the partition property, which is the record spec or rollup property by which records are assigned to each partition. An empty field means that Forge will use a round-robin strategy to assign partitions to records. |
| `PROP_TYPE` | Set to the partition property's type (typically, AL¬ PHA). |
| `REMOVE_PROP` | Typically set to `FALSE`. |
| `CUTOFF` | Set to the default value of `2000000000`. |

The following is an example of an update adapter using the above settings:

```
<UPDATE_ADAPTER NAME="UpdateAdapter"
  OUTPUT_URL="../partition0/dgraph_input/updates"
  OUTPUT_PREFIX=""
  MULTI="TRUE"
  MULTI_PROP_NAME="FILENAME">
 <RECORD_SOURCE>UpdateManipulator</RECORD_SOURCE>
 <ROLLOVER NAME="RECORD"
  NUM_IDX=""
  PROP_NAME="P_WineID"
  PROP_TYPE="ALPHA"
  REMOVE_PROP="FALSE"
  CUTOFF="2000000000"/>
</UPDATE_ADAPTER>
```

## ROLLOVER element for deterministic distribution

In a deterministic distribution pipeline, the are five recommended settings for the `ROLLOVER` element.

| Option | Description |
|---|---|
| `NUM_IDX` | Leave blank or use any number, as it is recommended you use the Forge `--numPartitions` flag for the Forge component to actually set the number of Agraph partitions. |
| `PROP_NAME` | Set to the property (`X_PartitionNum`, for example) created by the Perl expression in the record manipulator. |
| `PROP_TYPE` | Set to the `INTEGER` (because it will hold the partition number of the `ADD` record). |
| `REMOVE_PROP` | Set to `TRUE` (because the `PROP_NAME` property should not be in the output). |
| `CUTOFF` | Set to the default value of `2000000000`. |

The following is an example of an update adapter using the above settings:

```
<UPDATE_ADAPTER NAME="UpdateAdapter"
  OUTPUT_URL="../partition0/dgraph_input/updates"
  OUTPUT_PREFIX=""
  MULTI="TRUE"
  MULTI_PROP_NAME="FILENAME">
 <RECORD_SOURCE>UpdateManipulator</RECORD_SOURCE>
 <ROLLOVER NAME="RECORD"
  NUM_IDX=""
  PROP_NAME="X_PartitionNum"
  PROP_TYPE="INTEGER"
  REMOVE_PROP="TRUE"
  CUTOFF="2000000000"/>
</UPDATE_ADAPTER>
```

Chapter 6

# Backing Up Baseline and Partial Updates

Endeca recommends that you back up your MDEX Engine index files periodically. This lets you revert to a specific partial or baseline update. This section describes types of backups that you can perform for the MDEX Engine index files, lists backup recommendations, and describes how to recover the index by reverting to a previous state of the MDEX Engine index.

## Types of backups

In your implementation, you can run only baseline, or baseline and delta, updates without having to run partial updates. You can also run frequent partial updates along with periodic baseline updates. In each case, you need to back up your MDEX Engine index files periodically.

Typical backup scenarios fall into three categories:

- Baseline backup
- Snapshot backup
- Incremental backup

### About baseline backups

A baseline backup is a periodic backup of baseline updates only.

Baseline backups are always useful. In particular, they are useful when your baseline updates are so fast that you can recover from failures by rerunning baseline updates.

Performing baseline backups works well in implementations in which you run baseline or delta updates only, without having to run partial updates. In these cases, it is sufficient to back up baseline update files so that you can recover the index by restarting the MDEX Engine with the `dgraph_input` directory reconstructed from a baseline backup.

### About snapshot backups

A snapshot backup is a periodic backup of the `dgraph_input` directory after stopping the MDEX Engine. Snapshot backups are useful if your baseline updates are relatively infrequent.

Snapshot backups are useful when you run baseline or delta updates with periodic partial updates in between, and can afford to periodically stop and restart the MDEX Engine, which lets you back up the `dgraph_input` directory. For example, you may run a baseline update daily, partial update hourly, and stop and restart the MDEX Engine nightly to back up `dgraph_input`.

### About incremental backups

An incremental backup includes a backup of partial updates that have occurred since the last baseline backup or snapshot backup. Incremental backups allow you to revert to a more granular state of the MDEX Engine index.

In this scenario, you run baseline or delta updates with periodic partial updates in between, and back up partial update files, so that you can recover the index to its specific state at a particular date and time. In addition, you also create baseline or snapshot backups.

## Backup recommendations

Use the recommendations provided in this section to back up the Endeca index files used by the MDEX Engine.

- Back up the Dgraph input directory after each baseline update. Periodically, back up the `dgraph_input` directory and all its subdirectories to an alternate location. `dgraph_input` is the directory where the MDEX Engine index is stored. When backing it up, ensure that you use a naming scheme that will allow you to retrieve baseline update files based on date and time.

  Back up the `dgraph_input` directory only when the MDEX Engine is stopped. Do not back up this directory when the Dgraph is running. If you try to copy it while the Dgraph is running, you may capture files in an inconsistent state.
- Back up partial update files. Prior to running partial updates, ensure that the partial update files are saved automatically in another backup directory. You may need to modify your partial updates script so that this backup occurs automatically. Use a time stamp naming scheme to ensure that you can retrieve the update files if needed.

  The files in the `dgraph_input/updates` directory are deleted after a partial update completes successfully. Therefore, if you want to revert to a particular partial update, back up all files and subdirectories in this directory.
- Periodically delete previous backups of partial update files. In other words, you do not have to retain all incremental backups—only retain those incremental backups that occurred since the most recent baseline or snapshot backup. When you restore the index, you only need to use the partial update files since the baseline backup or snapshot backup of the `dgraph_input` directory (this is the copy of the directory on which you will restart the MDEX Engine).

  Once you do a baseline or snapshot backup, you can delete all backups of partial updates that took place before the baseline or snapshot backups, if you choose to do so.

## Recovering the Endeca index

You can recover the Endeca index by reverting to a particular baseline or partial update.

You can revert the MDEX Engine to the Endeca index representing the state of data after a specific baseline or partial update. To do so, restart the MDEX Engine on the index files that were backed up after this baseline update, and point the MDEX Engine at the `dgraph_input/updates` directory that contains partial update files that occurred since this baseline update.

To revert to a previously applied baseline or partial update:

1. Stop the MDEX Engine.
2. Clean up the active `dgraph_input` and `dgraph_input/updates` directories.
3. In the backup directory for baseline update files, locate the files from the last successful baseline update.
4. Copy the backed-up baseline update files into `dgraph_input`.
5. In the backup directory for partial update XML files, locate the files since the baseline update you are reverting to.
6. Copy all backed-up partial update XML configuration files from all partial updates that occurred since the baseline update to which you want to revert into the `dgraph_input/updates` directory.

   If you are using the Deployment Template, the *`app-dir`*`/data/partials/cumulative_partials` directory is where all partial updates since the last baseline are typically stored. You can reapply them by copying the necessary files to the Dgraph `updates` directory and apply the updates afterwards.
7. Restart the MDEX Engine. The MDEX Engine reads the files in `dgraph_input` and in `dgraph_input/updates`.

If you are using the Deployment Template, the `numPartialsBackups` setting (in the PartialForge module of the AppConfig.xml configuration file) determines how many cumulative partials to store. If you do not define this number high enough, you may not have all the previous update files to even restore the state of the index. If this is the case, you will have to run a full baseline.

Chapter 7

# Troubleshooting Partial Updates

This section contains recommendations for troubleshooting partial updates, and describes how the MDEX Engine treats failed partial update operations.

## Pipeline troubleshooting recommendations

Setting up partial updates correctly involves making sure that the record adapter, record manipulator, update adapter, dimension server, and other pipeline components are configured properly.

Use the following recommendations to troubleshoot a partial update pipeline:

- For all incoming records, verify that your Endeca project has an Endeca property configured as the record specifier. Only one property in the project can have the `RECORD_SPEC` attribute set to `TRUE`.
- For the record adapter, verify that the information provided to it is in the right format. The format should correspond to the pattern that you specified in the **URL** field. The record adapter works properly if all of the files with the extension `*.txt.gz` are formatted in the same way. For example, a URL pattern of `incoming/updates/*.txt.gz` means that Forge reads any file that has the `txt.gz` suffix in the `incoming/updates` directory. Each file that matches the pattern is read in sequence.
- If you are supplying multiple update files, verify that the **Multi** file field is checked in the **Record Adapter** editor in Developer Studio.
- For the record manipulator, verify that you are correctly using the expressions used for removing and manipulating the data.
- For the update adapter, verify that the **Output URL** field in Developer Studio points to the directory in which completed updated records from the last partial update will be placed by Forge, for the consumption by the MDEX Engine.
- For the update adapter, verify that the partial update pipeline output prefix is identical to the one used for baseline updates.
- For the naming format of the update source data files, use a timestamp as the naming scheme. This ensures that Forge processes the files in the proper order of their creation.
- For the EAC partial update script that runs partial updates, verify that it is configured properly. The default script is created for you when you use the Endeca Deployment Templates to create the Endeca project; you can modify it to suit your needs.

**Related Links**

*Naming format of update source data files* on page 37

When Forge processes update source data files, it is important to keep two issues in mind concerning the names of the data files.

## Troubleshooting update operations that fail

Verify that the MDEX Engine processes the updates according to your configuration in the partial updates pipeline.

The MDEX Engine processes updates on a record-by-record basis. Updates fail or succeed entirely at the record level. This means that a record update that fails (for example, because it attempts to assign an unknown dimension value to the record), leaves the value of the record unchanged. Property value changes or dimension value changes in the failed record update have no effect.

In addition, if an error occurs during a record update, and subsequent update operations relate to the same record, these operations may also fail. However, in general, previous and future record updates and dimension updates are not affected by a specific record update failure.

For example, if a partial update operation fails, because you try to delete a record that does not exist, or add a child dimension value to a dimension that does not exist, the MDEX Engine does all of the following:

- Continues the partial update process.
- Logs a message to `stderr`, or to the file specified by `--out` on the command line.
- Writes a copy of the entire file containing the failed update record or dimension value to the `<`*`updatedir`*`>/failed_updates` directory, which the MDEX Engine automatically creates in your working directory.

**Important:** The `failed_updates` directory may fill up with failed update files. To prevent your system from running out of disk space, periodically clean this directory.

Update files that are due for processing are deleted after each partial update that has successfully processed them. They do not accumulate in the `dgraph_input/updates` directory.

The default directory the MDEX Engine uses for storing failed update files is `<`*`updatedir`*`>/failed_updates/`.

**Note:** You can use the `--failedupdatedir` flag for the Dgraph to specify the directory in which the MDEX Engine should store the failed update files.

Therefore, to troubleshoot failed update operations, provision enough disk space in your working directory for `/failed_updates`, and check this directory for failed update operations, if you notice any failed update errors in the log.

During development, use the `--updateverbose` flag to specify that the MDEX Engine should output verbose messages while processing updates. Do not use this flag on production systems, because it negatively impacts partial update performance.

Keep in mind that when the MDEX Engine starts up, it begins to process queries and updates in parallel. This means that if there are update files in the `updates` directory at startup time, the MDEX Engine (after opening its port) begins to process those updates at the same time that it begins to accept and service queries.

## UPDATE_RECORD errors

If the `UPDATE_RECORD` expression is not configured properly in the record manipulator (which is used for updating dimension values or property values), Forge issues errors.

The following expression errors cause Forge to generate errors:

- `ACTION` is not one of `ADD`, `ADD_OR_REPLACE`, `REPLACE`, `DELETE`, `DELETE_OR_IGNORE`, or `UP¬ DATE`.
- `ACTION` is `ADD` and a record with that specification already exists. In this case, the record to be added is skipped instead of replacing the existing record. Use an `ACTION` of `ADD_OR_REPLACE` to add a record if it does not exist or replace it if it does.
- `ACTION` is `UPDATE` and a record with that specification does not exist. In this case, the record to be updated is skipped.
- `ACTION` is `UPDATE` and a sub-action is not specified.
- `ACTION` is not `UPDATE` and a sub-action is specified.
- `ACTION` is `DELETE` and a record with that specification does not exist. In this case, the record to be deleted is skipped and an error message is generated. Use an `ACTION` of `DELETE_OR_IGNORE` to suppress the error message if the record does not exist.
- More than one sub-`ACTION` (such as `DVAL_ACTION`) is specified for a given property, dimension, or dimension value.

## The Dgraph checks permissions on the index directories

Starting with the version 6.1.2, the Dgraph checks permissions on the index directories before applying partial updates.

If the required read/write permissions are missing, the Dgraph fails to apply the update and issues an error in the standard error log. It also logs the path to the index directories to which it does not have read/write permissions.

The Dgraph checks permissions on these directories in the `Endeca/myApp/dgidx_out¬ put/myApp_indexes`:

- `/committed`
- `/generations`

(The filepaths assume that the Deployment Template scripts are used to set up the application.)

Both of these directories should have read and write permissions to allow accessing them by the Dgraph. However, due to file system issues or hardware maintenance issues combined with the Endeca implementation's topology, it is possible that under some conditions these permissions are reset. This may make these directories unaccessible by the Dgraph.

## Performance impact of partial updates

This section provides a reference list of performance gains of partial updates.

- Partial updates do not require periodically running baselines for performance improvements. Update operations done through multiple partial updates do not require running a periodic baseline update due to performance concerns or memory-use constraints.

After you run the first few partial updates, MDEX Engine query performance decreases slightly. After this initial performance decrease, query performance stabilizes.

- Startup performance after partial updates decreases slightly and stabilizes afterwards. Overall, startup time is roughly proportional only to the total size of the MDEX Engine index, regardless of how many updates played a role in its state.
- Partial updates with high turnover and high frequency perform fast. High turnover means that a large portion of the data is being updated or deleted. Any mix of add, delete, and update operations on a large number of records is handled gracefully during partial updates.

  In addition, you can combine record updates into larger batches. Running such large-batch partial updates results in better overall throughput for the MDEX Engine. (The overall downtime for running one specific partial update with high data turnover may be longer, but in total, the time it takes to run one large-batch partial update is shorter compared with running many smaller scale partial updates in previous releases.)
- The MDEX Engine is stable in the face of hardware crashes. A power failure of the MDEX Engine server does not affect the state of indexed data. It leaves indexed data on disk in a consistent state no matter at which point in time a crash occurs. If a crash occurs during a partial update, the files from the `dgraph_input/updates` directory are not deleted. After a restart, the MDEX Engine checks the `dgraph_input/updates` directory for the presence of any files that were not applied and applies them.

# Index