Endeca® Latitude

LDI MDEX Engine Components Guide

Version 2.2.2 • December 2011

Contents

Preface....9
About this guide....9
Who should use this guide....9
Conventions used in this guide....9
Contacting Endeca Customer Support....10

Chapter 1: Latitude Data Integrator Overview....11
LDI Designer....11
List of Latitude connectors....12
LDI Server....14

Chapter 2: Before You Begin....17
Data loading strategies and concepts....17
- Which updates to run....17
- When to use transactions....18

Configuration tips....19
- Setting up the workspace.prm file....19
- Recommended order of loading data....20
- Creating mdexType Custom properties....20
- Specifying multiple record delimiters....23

Supported data types....25
Latitude-specific parameters in workspace.prm....25
Default values for new attributes....27
SSL support....28
Additional documentation....29

Chapter 3: Working with Transaction Graphs....31
About transactions....31
Requirements for running graphs within a transaction....31
Wrapping existing graphs in a transaction....32
Transaction graphs in the Latitude Quick Start project....33
Creating a Transaction RunGraph graph....34
- Format of the steps input file....35
- Adding components to the transaction graph....35
- Configuring the Reader for the transaction input file....36
- Configuring the Edge for Reader component....36
- Configuring the Transaction RunGraph connector....37
- Running the transaction graph....38

Committing an outer transaction....39
Performance impact of transactions....40

Chapter 4: Full Initial Index Load of Records....41
Overview of the full initial index load....41
Creating a project....42
Source data format....43
- Adding the source data to the project....44

Creating a graph....45
Adding components to the graph....46
Configuring the components....47
- Configuring the Reader component....47
- Configuring the Reformat component....50
- Configuring the Bulk Add/Replace Records connector....53

Running the graph to load records....54

Chapter 5: Incremental Updates....55

Overview of incremental updates....55
Adding components to the incremental updates graph....56
Configuring the Reader and the Edge for incremental updates....56
Configuring the Add/Update Records connector....57
Running the incremental updates graph....58

Chapter 6: Loading the Attribute Schema....61
About attribute schema files....61
Loading the standard attribute schema....61
  Format of the PDR input file....62
  Adding components to the standard attributes schema graph....63
  Configuring the Reader for the PDR input file....64
  Configuring the Reformat component for standard attributes....65
  Configuring the Denormalizer component....69
  Configuring the WebServiceClient component for standard attributes....72
Loading the managed attribute schema....75
  Format of the DDR input file....75
  Adding components to the managed attributes schema graph....76
  Configuring the Reader and the Edge for DDRs....77
  Configuring the Reformat component for managed attributes....77
  Configuring the Denormalizer and the Edge for DDRs....80
  Configuring the WebServiceClient component for managed attributes....81
Using a transaction graph to load the schemas....81

Chapter 7: Loading Configuration Files....83
Types of MDEX Engine configuration documents....83
  Global Configuration Record....84
  dimsearch_config document....85
  recsearch_config document....86
  relrank_strategies document....87
  stop_words document....88
  thesaurus document....89
Loading the configuration documents....90
  Creating a graph....91
  Adding components to the graph....91
  Configuring the Reader for the configuration document....92
  Configuring the FastSort component....94
  Configuring the first Denormalizer component....95
  Configuring the second Denormalizer component....97
  Configuring the WebServiceClient component....99
Loading the GCR....101

Chapter 8: Loading Precedence Rules....103
About precedence rules....103
Schema for precedence rules....103
Format of the precedence rules input file....105
Adding components to the precedence rules graph....106
Configuring the precedence rules Reader....106
  Configuring the Reader Edge....107
Configuring the Reformat component for precedence rules....107
  Configuring the precedence rules Reformat Edge....109
Configuring the precedence rules WebServiceClient component....111
Deleting precedence rules....113

Chapter 9: Adding Key-Value Pairs....115
About key-value pair data....115
Format of the KVP input file....115
Configuring the Reader for the KVP input file....116
Configuring the Add KVPs connector....117
Configuring KVP metadata....118
Running the KVPs graph....119

Chapter 10: Loading Taxonomies....121
Overview of loading a taxonomy....121
Format of the taxonomy input file....122
Creating a graph for the taxonomy....123
Adding components to the taxonomy graph....124
Configuring the Reader for the taxonomy input file....124
Configuring the Add Managed Values connector....125
Configuring taxonomy metadata....126
Running the taxonomy graph....127

Chapter 11: Importing and Exporting the Configuration....129
About importing and exporting....129
Exporting the configuration....130
    Adding components to the export graph....130
    Configuring the Export Config connector....131
    Configuring the UniversalDataWriter component....133
Importing the configuration....134
    Adding components to the import graph....134
    Configuring the Reader in the import graph....134
    Configuring the Import Config connector....137
Running the configuration graphs with a transaction graph....138

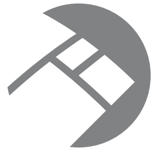
Chapter 12: Deleting Data....139
Format of the delete input file....139
Adding components to the delete data graph....140
Configuring the Reader for the delete input file....140
Configuring the metadata for data deletes....141
Configuring the Delete Data connector....142
Running the delete data graph....143

Chapter 13: Latitude Connector Reference....145
Bulk Add/Replace Records connector....145
Add/Update Records connector....148
Add KVPs connector....150
Add Managed Values connector....152
Delete Data connector....154
Export Config connector....156
Import Config connector....157
Reset MDEX connector....159
Transaction RunGraph connector....161
Visual and Common configuration properties....164
Connector output ports....167

Chapter 14: Troubleshooting Problems....169
OutOfMemory errors....169
BufferOverflow errors....170
Transaction-related errors....172
Connection errors....173
Multi-assign delimiter error....174

Appendix A: MDEX Engine Index Configuration Reference....175
XML elements....175
    COMMENT....175
    DIMNAME....176
    PROP....176
    PROPNAME....177
    PVAL....177
Dimsearch_config elements....177
    DIMSEARCH_CONFIG....178
Recsearch_config elements....178

RECSEARCH_CONFIG....178
Relrank_strategies elements....179
RELRANK_APPROXPHRASE....180
RELRANK_EXACT....180
RELRANK_FIELD....181
RELRANK_FIRST....181
RELRANK_FREQ....182
RELRANK_GLOM....182
RELRANK_INTERP....183
RELRANK_MAXFIELD....183
RELRANK_MODULE....184
RELRANK_NTERMS....184
RELRANK_NUMFIELDS....185
RELRANK_PHRASE....186
RELRANK_PROXIMITY....187
RELRANK_SPELL....187
RELRANK_STATIC....188
RELRANK_STRATEGIES....188
RELRANK_STRATEGY....189
RELRANK_WFREQ....191
Search_interface elements....192
MEMBER_NAME....192
PARTIAL_MATCH....193
SEARCH_INTERFACE....193
Stop_words elements....195
STOP_WORD....195
STOP_WORDS....196
Thesaurus elements....196
THESAURUS....197
THESAURUS_ENTRY....198
THESAURUS_ENTRY_ONEWAY....198
THESAURUS_FORM....199
THESAURUS_FORM_FROM....199
THESAURUS_FORM_TO....200



Copyright and disclaimer

Product specifications are subject to change without notice and do not represent a commitment on the part of Endeca Technologies, Inc. The software described in this document is furnished under a license agreement. The software may not be reverse engineered, decompiled, or otherwise manipulated for purposes of obtaining the source code. The software may be used or copied only in accordance with the terms of the license agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Endeca Technologies, Inc.

Copyright © 2003-2011 Endeca Technologies, Inc. All rights reserved. Printed in USA.

Portions of this document and the software are subject to third-party rights, including:

Corda PopChart® and Corda Builder™ Copyright © 1996-2005 Corda Technologies, Inc.

Outside In® Search Export Copyright © 2011 Oracle. All rights reserved.

Rosette® Linguistics Platform Copyright © 2000-2011 Basis Technology Corp. All rights reserved.

Teragram Language Identification Software Copyright © 1997-2005 Teragram Corporation. All rights reserved.

Trademarks

Endeca, the Endeca logo, Guided Navigation, MDEX Engine, Find/Analyze/Understand, Guided Summarization, Every Day Discovery, Find Analyze and Understand Information in Ways Never Before Possible, Endeca Latitude, Endeca InFront, Endeca Profind, Endeca Navigation Engine, Don't Stop at Search, and other Endeca product names referenced herein are registered trademarks or trademarks of Endeca Technologies, Inc. in the United States and other jurisdictions. All other product names, company names, marks, logos, and symbols are trademarks of their respective owners.

The software may be covered by one or more of the following patents: US Patent 7035864, US Patent 7062483, US Patent 7325201, US Patent 7428528, US Patent 7567957, US Patent 7617184, US Patent 7856454, US Patent 7912823, US Patent 8005643, US Patent 8019752, US Patent 8024327, US Patent 8051073, US Patent 8051084, Australian Standard Patent 2001268095, Republic of Korea Patent 0797232, Chinese Patent for Invention CN10461159C, Hong Kong Patent HK1072114, European Patent EP1459206, European Patent EP1502205B1, and other patents pending.

Preface

Endeca® Latitude applications guide people to better decisions by combining the ease of search with the analytic power of business intelligence. Users get self-service access to the data they need without needing to specify in advance the queries or views they need. At the same time, the user experience is data driven, continuously revealing the salient relationships in the underlying data for them to explore.

The heart of Endeca's technology is the MDEX Engine.™ The MDEX Engine is a hybrid between an analytical database and a search engine that makes possible a new kind of Agile BI. It provides guided exploration, search, and analysis on any kind of information: structured or unstructured, inside the firm or from external sources.

Endeca Latitude includes data integration and content enrichment tools to load both structured and unstructured data. It also includes Latitude Studio, a set of tools to configure user experience features including search, analytics, and visualizations. This enables IT to partner with the business to gather requirements and rapidly iterate a solution.

About this guide

This guide describes the components in the Endeca Latitude Data Integrator Designer that are used to ingest data into the MDEX Engine.

The Latitude Data Integrator Designer is used to load records, taxonomies, and configuration documents into the MDEX Engine.

The guide assumes that you are familiar with Endeca concepts and Endeca application development, as well as the interface of the Data Ingest Web Service.

Who should use this guide

This guide is intended for developers who are responsible for loading source data into the MDEX Engine.

Conventions used in this guide

This guide uses the following typographical conventions:

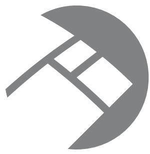
Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

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 Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.



Chapter 1

Latitude Data Integrator Overview

The Latitude Data Integrator (LDI) is a high-performance platform that lets you extract source records from a variety of source types, and load them into the MDEX Engine. The LDI consists of the LDI Designer, the Endeca Latitude connectors, and an LDI Server.

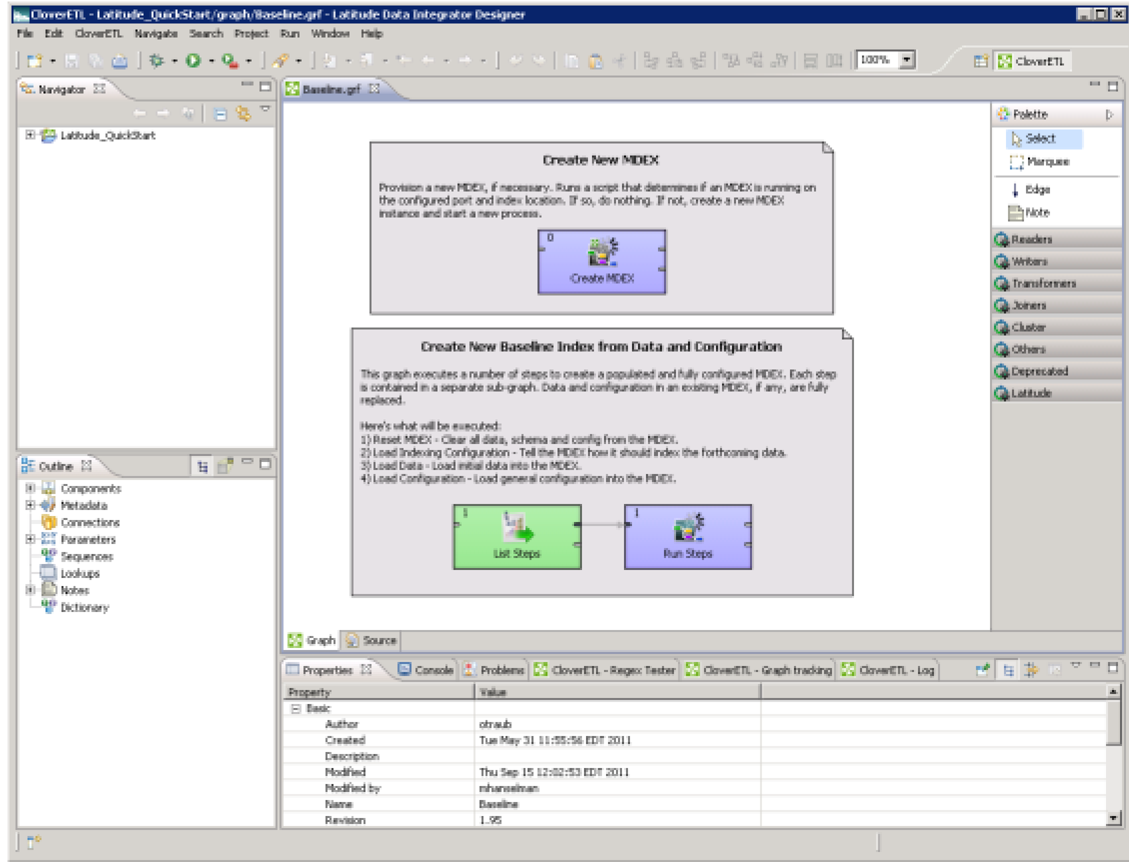
LDI Designer

Use the LDI Designer to create the graphs for loading and updating your data.

A graph is essentially a pipeline of components that processes the data. The simplest graph has one Reader component to read in the source data and one of the Endeca components to write (send) the data to the MDEX Engine. More complex graphs will use additional components, such as Transformer and Joiner components.

The Designer, with its powerful graphical interface, provides an easy way to graphically lay out even complex graphs. You drag and drop the components from the Palette and then configure them by clicking on the component icon.

The Designer perspective consists of four panes and the Palette tool, as shown in this example:



These panes are:

- The **Navigator** pane lists your projects, their folders (including the graph folders), and files.
- The **Outline** pane lists all the components of the selected graph.
- The **Tab** pane consists of a series of tabs (such as the Properties tab and the Console tab) that provide information about the components and the results of graph executions. The illustration shows the Log tab listing the output of a successful record loading operation.
- The **Graph Editor** pane lets you create a graph and configure its components.
- The **Palette** lets you select a component and drag it to the Graph Editor.

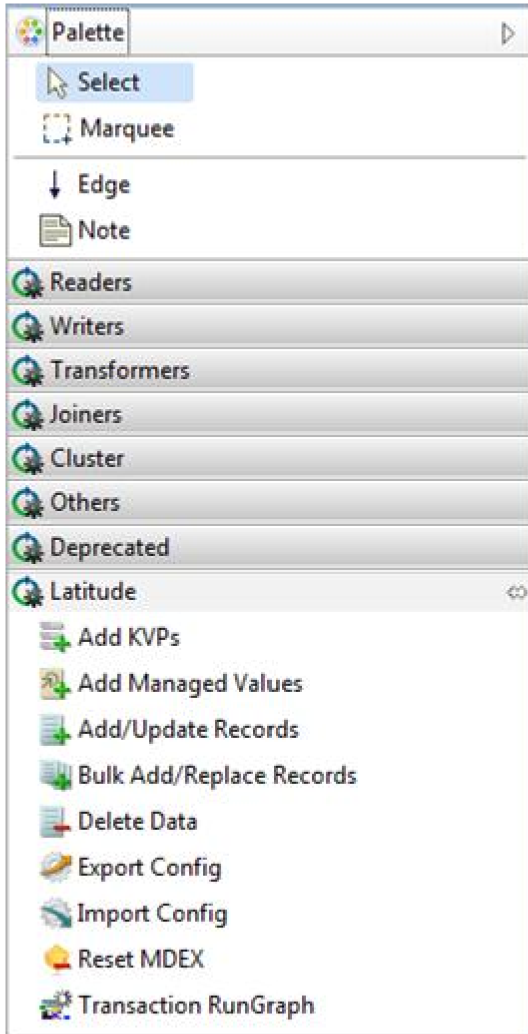
For more information on the Designer user interface, see the *Latitude Data Integrator Designer Guide*.

List of Latitude connectors

The Endeca Latitude connectors are used to load records into the MDEX Engine, delete records, export and import the MDEX Engine index and configuration, and start a transaction.

The Endeca-developed Latitude connectors a specifically designed to work with the records and configuration stored in the MDEX Engine. They utilize the MDEX Engine web services and the Bulk Load Interface.

Latitude connectors are grouped in the Designer Palette under the Latitude section:



To use a specific Latitude connector, select it from the palette and drag it into your graph.

The following table provides a brief overview of all Latitude connectors:

Latitude Connector	Description
Add/Update Records	Adds new records to a running MDEX Engine. You can add records to an empty MDEX Engine index (this operation is called a full index initial load), or to one that already contains records. You can also use this connector to load the records schema, by loading the PDRs (Property Description Records) and DDRs (Dimension Description Records). If an Endeca standard attribute to be added does not exist, it is created automatically.

Latitude Connector	Description
Bulk Add/Replace Records	Adds new records to a running MDEX Engine. You can add records to an empty MDEX Engine index (this operation is called a full index initial load), or to one that already contains records. If an Endeca standard attribute to be added does not exist, it is created automatically.
Add KVPs	Updates existing records by adding new key-value pair (KVP) assignments to those records. The connector can also create new records for the key-value pairs, as well as creating new standard attributes for KVP assignments for non-existent standard attributes.
Add Managed Values	Adds a taxonomy (managed values) to a running MDEX Engine. If the managed values belong to a managed attribute that currently does not exist in the MDEX Engine index, the managed attribute is created automatically.
Delete Data	Removes KVP assignments from records in the MDEX Engine index, or deletes entire records (that you specify for deletion).
Export Config	Exports the schema and configuration stored in the MDEX Engine.
Import Config	Imports the schema and configuration into the MDEX Engine.
Reset MDEX	Resets the MDEX Engine index back to the empty state by removing all the records (including the schema) from the MDEX Engine, provisioning the MDEX Engine and updating the spelling dictionary.
Transaction RunGraph	Runs other LDI graphs within it, similar to the standard **RunGraph** component available with the LDI. Unlike the standard **RunGraph**, **Transaction RunGraph** starts the outer transaction and runs multiple sub-graphs within that transaction.

For a comprehensive reference of the connectors and their configuration properties, see the *Latitude Connector Reference* section in this guide.

For information on how to build LDI Designer graphs with the connectors, see the corresponding sections in this guide.

LDI Server

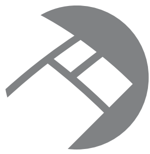
The LDI Server provides a runtime environment for the graphs.

The Latitude Data Integrator Server is not required in order to load data into the MDEX Engine. In other words, the Data Integrator Designer clients can run independently, and do not require the Server in order to do their work.

You use the Server only if you are running graphs in an enterprise-wide environment. In this environment, different users and user groups can access and run the graphs. In addition, you can schedule the graphs to run at designated times, and monitor their execution progress.

The Server runs on an enterprise application server, such as Apache Tomcat or IBM Websphere.

Because the Server is not a mandatory component for loading data into the MDEX Engine, it is not documented in this guide. For information on the setup and use of the Latitude Data Integrator Server, see the *Latitude Data Integrator Server Guide*.



Chapter 2

Before You Begin

This section provides configuration tips, lists data types supported in the LDI with the MDEX Engine, provides a reference for default values for newly added attributes, and discusses common ETL strategies you can employ for loading and updating data and configuration in the MDEX Engine index.

Data loading strategies and concepts

Endeca recommends the following common approaches and strategies for loading data and conducting other operational tasks.

Which updates to run

This topic discusses at a high-level which types of updates are typically run. This lets you decide which types of graphs you need to create in LDI for your update purposes.

Typical data update strategies for the Latitude application include the following:

Type of update	Description
Initial index load	This update is also known as initial baseline update. It's basic idea is simple loading of data, without the need to preserve any previously configured settings. It includes loading data into an empty MDEX Engine index. This update assumes that the MDEX Engine index is empty, and that the configuration and schema have only their default values acquired at the MDEX Engine provisioning stage. As an example, the Baseline graph from the Latitude Quick Start project performs an initial data load.
Baseline update	This update is also known as a subsequent baseline or a re-baseline. It's basic idea is to replace almost everything in the MDEX Engine index, and to avoid losing configuration changes that you may have already made interactively. This type of update is typically repeatable. It implies loading of the data into the MDEX Engine index that already contains previously loaded data. Such an index may also contain configuration that has been changed from its defaults. Similarly, the attributes schema may have been modified.

Type of update	Description
	For a re-baseline, a typical graph would contain an **Export Config** connector to export an existing configuration and schema, a **Reset MDEX** connector that removes all records and schema and provisions a new MDEX Engine, an **Import Config** connector that imports the previously exported configuration, and, finally, a set of connectors that load data. This set of sub-graphs may be run inside a **Transaction RunGraph**, in case you want to have control over the completeness of updates.
Partial update	This update includes adding new records and making changes to the records and configuration that already exist in the MDEX Engine index. For a partial update, a typical graph contains a **UniversalDataReader** and a **Add/Update Records** connector.

When to use transactions

This topic discusses transactions and provides recommendations for when it is useful to run your LDI graphs inside transactions as opposed to running graphs that do not utilize them.

Typically, LDI components load data and configuration into the MDEX Engine index by making web service requests or requests to the Bulk Ingest interface. Each web service request represents its own set of operations in the MDEX Engine, and succeeds or fails on its own. This means that, if some calls to the MDEX Engine succeed and others fail, the resulting MDEX Engine index may reflect only a partially updated data set (if, for example, some updates did not succeed).

In some cases, however, you may want to ensure that data changes from an entire data-updating graph either complete or fail as a unit, so that the resulting MDEX Engine index represents an entirely updated data set. You may also want to make sure that end users do not access intermediate states of the data in Latitude Studio, but instead can only have access to the pre-update state of the index (while the data-updating graph completes), and then seamlessly transition to the index after it has been fully updated.

To guarantee that your updates either completely succeed or fail, use a graph that runs an outer transaction.

An *outer transaction* (also known as transaction) is a set of operations performed in the MDEX Engine that is viewed as a single unit. If a transaction is committed, this means that all of the data and configuration changes made during the transaction have completed successfully and are committed to the MDEX Engine index.

To run a transaction, use the Latitude connector **Transaction RunGraph** in the LDI. This connector lets you create a graph that starts and commits an outer transaction in the MDEX Engine, utilizing calls to the Transaction Web Service. Using this connector, you can add sub-graphs and components that will run inside an outer transaction. Typically, a graph that runs a transaction is useful for running updates. Once such a graph completes, an update to your records is guaranteed to be fully committed to the MDEX Engine index.

Related Links

Working with Transaction Graphs on page 31
This chapter describes how to build an LDI transaction graph that can sub-graphs in a transaction environment. It also provides information about starting, committing, and rolling back transactions.

Wrapping existing graphs in a transaction on page 32

You can wrap any of your existing graphs in a graph that uses **Transaction RunGraph** connector.

Configuration tips

This section provides tips and general information for configuration tasks.

Setting up the workspace.prm file

If you start a new Latitude project and are planning to use transactions in it, ensure that the project's `workspace.prm` file lists the `MDEX_TRANSACTION_ID` parameter with an empty ID value.

The `MDEX_TRANSACTION_ID` parameter is specific to the MDEX Engine and is used to control transactions. The default `workspace.prm` file for a new project does not contain this parameter.

When you start a new Latitude project in LDI Designer, add the following line to your `workspace.prm` file:

```
MDEX_TRANSACTION_ID=
```

where the actual value of the ID is left blank.

This line ensures that in this project, you can run graphs with and without transactions.

- Graphs that use transactions. To run a graph that starts and commits a transaction, use the **Transaction RunGraph** connector. This connector automatically overrides the value of the ID (which is not specified) with the value `transaction` for the duration of the transaction.

  All Latitude components that you add to such a graph do not require any special configuration and are designed to accept this ID, if it is provided by **Transaction RunGraph**.

  Non-Latitude components that you add to this graph (those that use **WebServiceClient** or **HTTP Connector**) must have `"${MDEX_TRANSACTION_ID}"` attribute specified in their request structure. If such components are used within **Transaction RunGraph**, they automatically accept the transaction ID provided by this graph, for the duration of the transaction.

- Graphs that do not use transactions. To run a graph that does not use transactions, you add any components to it, and do not use **Transaction RunGraph** or any other graph associated with transactions.

  All Latitude components added to such a non-transaction graph accept the ID provided in `workspace.prm`. Since the value of this ID is empty, the transaction ID attribute is ignored, which allows these components to run outside of a transaction.

  Non-Latitude components that you add to this graph must have `"${MDEX_TRANSACTION_ID}"` attribute specified in their request structure. If such components are used within any graph that does not use transactions, they accept the transaction ID provided in `workspace.prm`. If this ID is empty, they ignore the attribute for the ID, which allows these components to run outside of a transaction.

Recommended order of loading data

This topic provides a recommended order for loading your configuration information and source data into the MDEX Engine.

Assuming that you are starting with an empty MDEX Engine (that is, only `mkmdex` has been run), the recommended order of loading your data is the following:

1. Global Configuration Record (GCR), which sets the global configuration settings for the MDEX Engine.
2. Attribute Schema Configuration, which creates the standard attributes and managed attributes, in this order:
   a. Standard attribute schema, which are the Property Description Records (PDRs)
   b. Managed attribute schema, which are the Dimension Description Records (DDRs)
   c. Managed attribute values (mvals)
3. Attribute Group Configuration, which consists of creating groups and adding attributes to them
4. Index Configuration, which consists of the index configuration documents in this order:
   a. `relrank_strategies` document (necessary if a relevance ranking strategy is referenced by the next two documents)
   b. `recsearch_config` document
   c. `dimsearch_config` document
   d. `stop_words` document
   e. `thesaurus` document
5. Application Source Records, which consist of the data on which user queries will be made.

You may alter the order to fit the needs of your Latitude application. For example, if you are satisfied with the default settings of the GCR, then there is no need to load the GCR. Or, to use another example, you do not need to load your attribute group configuration if you intend to create and manage attribute groups with Latitude Studio's Attribute Settings component.

Creating mdexType Custom properties

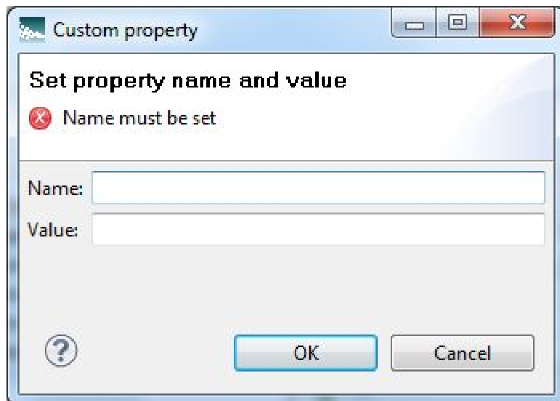
LDI Designer allows you to create an **mdexType** Custom property that you can use to explicitly specify the MDEX type to which a particular Endeca standard attribute should map.

The Custom Property feature can be used to specify MDEX types (such as `mdex:duration`, `mdex:time`, and `mdex:geocode`) that are not natively supported in the Designer. In this case, the ETL developer has to send a string through the Designer, making sure that the string value is formatted in the way that the MDEX Engine expects. The new **mdexType** Custom property, in other words, overrides the Designer native property type when the records are sent to the MDEX Engine.

This functionality is particularly useful for non-String multi-assign properties, because the Designer natively has to treat the property as a string since it has to include a delimiter. Thus, you can include delimiters in the multi-assign property (as though it were a String) but send the property to the MDEX Engine with `mdex:int` (for example) as the MDEX property type.

Important: Although the property will be designated as Designer type String, you must make sure that the string value is formatted according to the rules of the MDEX property type to which it will be mapped. For example, if it will be created as an `mdex:duration` attribute in the MDEX Engine, then the String value must use the `mdex:duration` format.

You add Custom properties by invoking the Custom property editor from the Fields pane in the Metadata Editor:



The Name field must be **mdexType** and the Value field must be one of the MDEX property types (such as `mdex:duration`). The Name and Value are used by the Latitude connector to specify (to the MDEX Engine) what MDEX property type should be used for when creating the standard attribute.

The source input file used as an example is a simple one:

```
ProductKey|ProductName|Duration|Location
95000|HL Mountain Rim|P429DT2M3.25S|42.365615 -71.075647
```

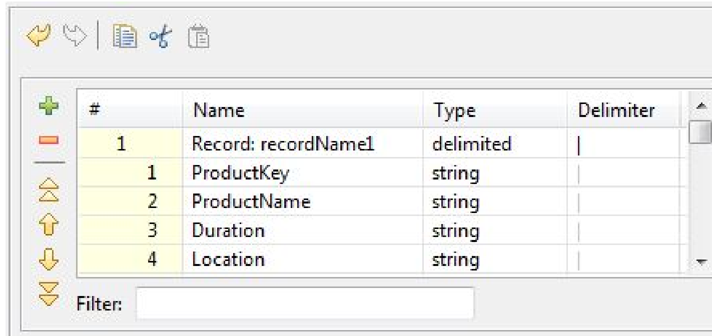
It creates only one record with four standard attributes:

- The ProductKey attribute is the primary key and is an Integer. Its value is 9500.
- The ProductName attribute is a String type with a value of "HL Mountain Rim".
- The Duration attribute will be a String property in the Designer metadata but will use a Custom property of `mdex:duration` in order to create a Duration standard attribute. Its value is "P429DT2M3.25S" (which specifies a duration of 429 days, 2 minutes, and 3.25 seconds).
- The Location attribute will be a String property in the Designer metadata but will use a Custom property of `mdex:geocode` in order to create a Geocode standard attribute. Its value is "42.365615 -71.075647" (which specifies a location at 42.365615 north latitude, 71.075647 west longitude).

To create a Custom property:

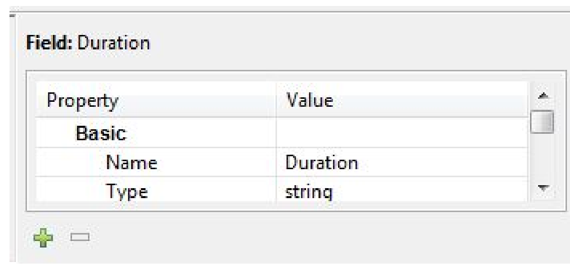
1. Create a graph with at least one reader, a Latitude connector (such as the **Add/Update Records** connector), and an Edge component.
2. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
3. In the Flat File dialog, select the input file and then click **Next** to display the Metadata editor.
4. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.

   At this point, the Record pane of the Metadata editor should look like this:



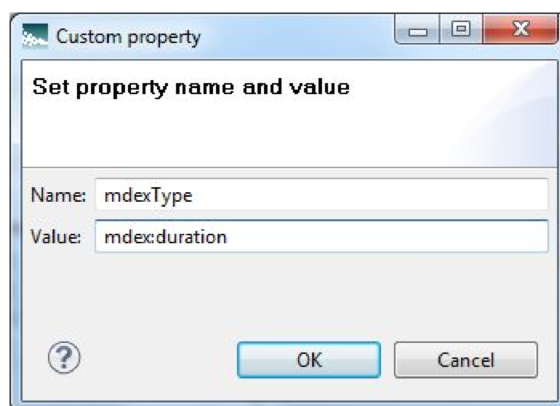
5. In the Record pane of the Metadata editor, make these changes:
   a) Click the **Record:recordName1** Name field and change the **recordName1** default value to a more descriptive name.
   b) Change the ProductKey Type to **integer**.
   c) Leave the ProductName Type as **string**.
6. To create a Custom property type for the Duration property:
   a) In the Record pane, click the Duration property to high-light it.

      The Duration property is displayed in the Field pane on the right, as in this example:



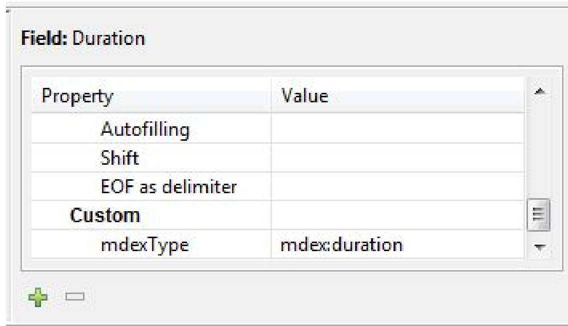
   b) In the Field pane, click the green **+** icon to bring up the Custom property editor.
   c) Enter **mdexType** in the Name field and **mdex:duration** in Value field.

      The Custom property editor should look like this:



   d) Click **OK** in the Custom property editor.

      As a result, a Custom section (with the new **mdexType** property) is added to the Duration property in the Field pane:



7. Repeat Step 6 if you want to create another **mdexType** Custom property type for another of your source properties.
   For example, for the Location attribute, you would create an **mdexType** Custom property with **mdex:geocode** in the Value field.
8. Click OK to apply your changes and close the Metadata editor.

As mentioned above, when the graph is run to add records, the MDEX Engine will use the **mdexType** Custom properties to create the standard attributes.

Keep in mind that you can create **mdexType** Custom properties for any of the MDEX property types, by setting the Value field to:

- `mdex:boolean` for Booleans
- `mdex:dateTime` to represent the date and time to a resolution of milliseconds
- `mdex:double` for floating-point values
- `mdex:duration` to represent a length of time with a resolution of milliseconds
- `mdex:geocode` to represent latitude and longitude pairs
- `mdex:int` for 32-bit signed integers
- `mdex:long` for 64-bit signed integers
- `mdex:string` for XML-valid character strings
- `mdex:time` for time-of-day values to a resolution of milliseconds

Specifying multiple record delimiters

By using an OR operator, you can specifying multiple record delimiters in the metadata.

In the Edge metadata, the default record delimiter for a file depends on which operating system the file was created. For example, the default record delimiter for a Windows file is **\r\n** while **\n** is typically used for Linux files.

However, you may have files that were created on different platforms (for example, if you have input files that you check out of a version control system, the files' line endings will vary according to the platform). In this case, you would want the record delimiter to be set to both values, so that you could use the same graph on Windows or Linux. You would then set the record delimiter to:

```
\r\n\\|\n
```

The | (pipe) character is an OR operator and the **\\|** syntax is a way to escape that OR operator in the LDI interface.

To specify multiple record delimiters in the metadata:

1. In the Record pane of the Metadata Editor, click the first row (the Record row).

In this example, you would click the Record:ProductCategory row.

#	Name	Type	Delimiter
1	Record: ProductCategory	delimited	,
1	ProductCategoryKey	integer	,
2	ProductCategoryAlternat...	integer	,
3	EnglishProductCategory...	string	,
4	SpanishProductCategory...	string	,
5	FrenchProductCategory...	string	\r\n

Filter:

2. In the Details pane (to the right of the Record pane), check the **Record delimiter** property to see the default setting.

   In this example, **\r\n** is set as the record delimiter.

Field: ProductCategory

Property	Value
Basic	
Name	ProductCategory
Type	delimited
Record delimiter	\r\n
Default delimiter	,
Skip source rows	1
Description	

3. Place the cursor in the Value field of the **Record delimiter** property and type in **\r\n\\|\n** as the value.

   The Details pane should now look like this:

Field: ProductCategory

Property	Value
Basic	
Name	ProductCategory
Type	delimited
Record delimiter	\r\n\\|\n
Default delimiter	,
Skip source rows	1
Description	

4. Click OK to save your changes made in the Metadata Editor.

Supported data types

This topic lists the Designer native data types and specifies which of them are supported in the MDEX Engine.

The table also shows how the Designer supported data types are mapped to MDEX Engine data types during an ingest operation. You will see the data types when you create the Metadata definition for the Edge component connector.

Designer Data Types in Metadata	Maps to MDEX Data Type
`boolean`	`mdex:boolean`
`byte`	Not supported
`cbyte`	Not supported
`date`	`mdex:dateTime`
`decimal`	`mdex:double`
`integer`	`mdex:int`
`long`	`mdex:long`
`number`	`mdex:double`
`string`	`mdex:string`
`string` with an `mdexType` Custom property set to `mdex:duration`	`mdex:duration`
`string` with an `mdexType` Custom property set to `mdex:geocode`	`mdex:geocode`

As the table notes, you can create an `mdexType` Custom property type for the input property's metadata and the MDEX Engine will use that type when creating the standard attribute's PDR. For details, see the "Creating mdexType Custom properties" topic in Chapter 10 of this guide.

Latitude-specific parameters in workspace.prm

The `workspace.prm` file contains parameters that define your project. Some parameters in this file are specific to Latitude, and in particular, to the MDEX Engine. This topic lists these Latitude-specific parameters.

The `workspace.prm` file is located under your project's directory, in the **Navigator** pane. Each project contains its own `workspace.prm` file. To view it, open it with Text Editor.

This file lists parameters that must be frequently referenced by components in your project, such as locations of the `DATA IN` and `DATA OUT` directories. Instead of referencing these values directly, you can specify them once in the `workspace.prm` and then reference the parameters when configuring your project's components.

A new project contains a `workspace.prm` file with default settings which you can modify to suit your needs.

The following table lists those parameters that affect the Latitude projects in which data is sent to the MDEX Engine. (For information about the non-Latitude parameters in this file, see the *LDI Designer Guide*.)

Note: While you can modify the parameter values, do not change the parameter names, because these names are reserved in LDI.

The `workspace.prm` file contains the following parameters specific to Latitude and the MDEX Engine:

Parameter	Description
`MDEX_HOST`	The host name of the server on which the MDEX Engine is running. This parameter is optional, but is recommended to include.
`MDEX_PORT`	The port of the server on which the MDEX Engine is running. This parameter is optional, but is recommended to include. For example, instead of specifying a specific port name, you can specify `${MDEX_PORT}` in the **MDEX port** field for any connector that requires it.
`MDEX_BULK_PORT`	The port for the Bulk Ingest Interface. This parameter is optional, but is recommended to include.
`MDEX_TRANSACTION_ID`	The ID of the outer transaction for the MDEX Engine. In a new project, this parameter is not specified, and you must add it as follows (with an empty value): `MDEX_TRANSACTION_ID=` This ensures that in your project, you can run components within graphs that either use or do not use transactions: • In a graph that uses transactions, Latitude-specific and non-Latitude components rely on the ID provided to them by the graph that runs a transaction (this graph overrides the ID in this file, for the duration of the transaction). Non-Latitude components must have `outerTransactionId="${MDEX_TRANSACTION_ID}"` specified in their request structure. • In a graph that does not use transactions, both Latitude-specific and non-Latitude components ignore this ID if it is empty, which allows them to run outside of a transaction.

Note: If you have any of the sample Latitude projects loaded in the LDI Designer, a few additional MDEX Engine-specific parameters may be listed in this file. These additional parameters are optional and are created in this file for the purposes of the sample projects.

Example: How to specify an outer transaction ID parameter

This example illustrates how to specify an outer transaction ID parameter in the component's configuration.

Note: Latitude-specific components automatically reference this ID, if it is specified in the `workspace.prm` file for your project with an empty value. However, you need to configure non-Latitude components that use MDEX Engine web services or bulk ingest interface to reference this ID, if you plan to use these components in graphs that run transactions, in addition to using them in graphs that do not run transactions.

For example, if you are using a **WebServiceClient** component for running any of the MDEX Engine web services, and plan to use this component inside an outer transaction, the **Request Structure** field for the component must include an attribute `outerTransactionId` with an ID of an outer transaction.

Note: If you do not use transactions, then this component should still contain the `outerTransactionID`, however, because it's value is empty in `workspace.prm`, it is ignored when this component runs outside of a transaction.

Specify the following request In the **Request Structure** field for your component:

```
<config-service:configTransaction
xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"
outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putGroups
xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
...
</config-service:putGroups>
</config-service:configTransaction>
```

where the string `outerTransactionId="${MDEX_TRANSACTION_ID}"` specifies the ID of the outer transaction listed in the `workspace.prm` file for your project.

Default values for new attributes

New standard and managed attributes created during an ingest are given a set of default values.

During any data ingest operation, if a non-existent Endeca standard attribute is specified for a record, the specified attribute is automatically created by the MDEX Engine. Likewise, non-existent Endeca managed attributes specified for a record are also automatically created. Note that you cannot disable this automatic creation of these attributes.

Standard attribute default values

The PDR for a standard attribute that is automatically created will use the system default settings, which (unless they have been changed by the data developer) are:

PDR property	Default setting
`mdex-property_Key`	Set to the standard attribute name specified in the request.

PDR property	Default setting
mdex-property_Type	Set to the standard attribute type specified in the request. If no type was specified, defaults to the mdex:string type.
mdex-property_IsPropertyValueSearchable	true (the standard attribute will be enabled for value search)
mdex-property_IsSingleAssign	false (a record may have multiple value assignments for the standard attribute)
mdex-property_IsTextSearchable	false (the standard attribute will be disabled for record search)
mdex-property_IsUnique	false (more than one record may have the same value of this standard attribute)
mdex-property_TextSearchAllowsWildcards	false (wildcard search is disabled for this standard attribute)
system-navigation_Select	single (allows selecting only one refinement from this standard attribute)
system-navigation_ShowRecordCounts	true (record counts will be shown for a refinement)
system-navigation_Sorting	record-count (refinements are sorted in descending order, by the number of records available for each refinement)

Managed attribute default values

A managed attribute that is automatically created will have both a PDR and a DDR created by the MDEX Engine. The default values for the PDR are the same as listed in the table above, except that `mdex-property_IsPropertyValueSearchable` will be `false` (i.e., the managed attribute will be disabled for value search).

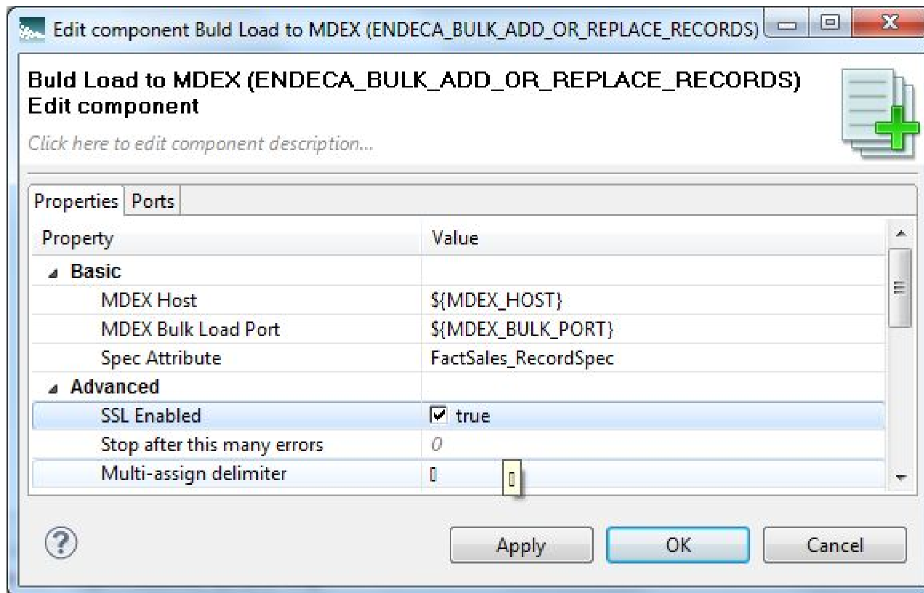
The DDR will use the system default settings, which (unless they have been changed by the data developer) are:

DDR property	Default setting
mdex-dimension_Key	Set to the managed attribute name specified in the request.
mdex-dimension_EnableRefinements	true (refinements will be displayed)
mdex-dimension_IsDimensionSearchHierarchical	false (hierarchical search is disabled during value searches)
mdex-dimension_IsRecordSearchHierarchical	false (hierarchical search is disabled during record searches)

SSL support

All Latitude connectors support SSL connections to an SSL-enabled MDEX Engine. You configure SSL connections in the LDI Designer **Edit component** dialog.

For example, this diagram shows the **SSL Enabled** field that you can configure for the **Bulk Add/Replace Records** connector:



Additional documentation

Additional LDI Designer documentation is available online and as part of the Latitude documentation set.

Latitude documentation set

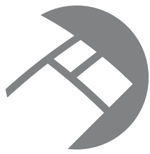
The following PDF documents are shipped as part of the Latitude documentation set:

- *Latitude Data Integrator Getting Started Guide* – a guide for ETL developers and data architects who want to explore the basics of the Designer.
- *Latitude Data Integrator Designer Guide* – a comprehensive user's guide for the Designer.
- *Latitude Data Integrator Server Guide* – a comprehensive user's guide for the LDI Server.

Documentation online

You can access online documentation from within the Designer by clicking **Help Contents** from the **Help** menu. Doing so brings up three documents:

- *CloverETL Designer User's Guide* – the online version of the *Latitude Data Integrator Designer Guide*.
- *Workbench User Guide* – describes the Eclipse Workbench development environment.
- *Java Development User Guide* – describes how to use the Java development tools.



Chapter 3

Working with Transaction Graphs

This chapter describes how to build an LDI transaction graph that can sub-graphs in a transaction environment. It also provides information about starting, committing, and rolling back transactions.

About transactions

An *outer transaction* (also known as transaction) is a set of operations performed in the MDEX Engine that is viewed as a single unit.

If a transaction is committed, this means that all of the data and configuration changes made during the transaction have completed successfully and are committed to the MDEX Engine index.

If any of the changes made within a transaction fail to complete successfully, the transaction fails to commit. In this case, you can choose to roll back the entire transaction, and the changes to the MDEX Engine index do not occur.

In general, the best practice is to set up operations so that successful updates are automatically committed (this is the default), but failed updates can be rolled back either automatically or manually.

The MDEX Engine Transaction Web Service is used for controlling outer transactions.

Requirements for running graphs within a transaction

If you would like to use outer transactions in your graphs, consider these requirements.

- Do not start more than one outer transaction at a time. If you have a graph that starts an outer transaction, such as a graph built with the **Transaction RunGraph** connector, it is important not to start another graph that attempts to start another outer transaction, otherwise, the MDEX Engine issues a transaction fault error.
- You can run all Latitude components inside a graph that starts an outer transaction. In other words, all Latitude components in LDI are transaction-friendly. When any Latitude component is run within such a graph, the underlying update operations from the web services or Bulk Ingest interface will reference the outer transaction ID in their calls to the MDEX Engine. This ID is provided to these components by the **Transaction RunGraph** connector. For the duration of the transaction, the connector sets the ID to `transaction`. In addition, the ID must be specified as `MDEX_TRANSAC¬ TION_ID=` in the `worskpace.prm` file for your project (notice the empty value). This allows the same components to be used in graphs that do not use transactions, without having to modify `worskpace.prm`.

Note: All Latitude components can also run in graphs that don't start a transaction. If you have a simple implementation, or if a graph that you are creating is light-weight and is not intended for heavy-duty data loading or configuration updates, it can run on its own and does not necessarily need to be run inside an outer transaction.

- If you are using a **WebServiceClient** component in LDI that is configured to run any of the MDEX Engine web services, the **Request Structure** field for the component must include an attribute `outerTransactionId` with a value of an outer transaction. For example, the following request specified in the **Request Structure** references the outer transaction ID as a parameter:

```
<config-service:configTransaction
xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

outerTransactionId="${MDEX_TRANSACTION_ID}">
...
</config-service:configTransaction>
```

  In this example, the string `outerTransactionId="${MDEX_TRANSACTION_ID}"` references the ID of the outer transaction listed in the `workspace.prm` file for your project.
- Consider creating all your data-updating graphs inside a graph that starts and commits an outer transaction:
  - In a clustered environment, the best practice is to use an outer transaction.

    When an outer transaction is started, it locks out all queries on the leader node for its duration. Latitude Studio cannot send queries to the MDEX Engine node that is processing an outer transaction.

    In a cluster, the LDI graph sends updates to the leader node only. During an outer transaction, the leader node responds to any queries, including the `admin?op=ping` command with an HTTP 403 response code. This way, the load balancer can be configured to automatically detect whether a transaction is in progress and remove the leader node from answering queries, while other nodes in the cluster continue to respond to user requests in Latitude Studio.
  - In a non-clustered environment (a single MDEX Engine node without the Cluster Coordinator), you may still use an outer transaction. Such a graph lets you group a set of operations into a single atomic unit that either succeeds or fails as a whole. Be aware, however, that while an outer transaction is running, the node does not serve queries. That is, if you run a graph within a transaction on a single MDEX Engine server, you benefit from an all-or-nothing data update operation, but while the transaction is running, the MDEX Engine does not respond to queries.

Related Links

Transaction-related errors on page 172

You may receive various transaction-related errors if you attempt to overlap running graphs wrapped in transactions with graphs that do not start an outer transaction.

Wrapping existing graphs in a transaction

You can wrap any of your existing graphs in a graph that uses **Transaction RunGraph** connector.

To wrap your existing graphs in a transaction:

1. Create a parameter in your workspace parameters file `workspace.prm` with this line:

   ```
   MDEX_TRANSACTION_ID=
   ```

   where the value is empty.
2. Add the following line to the outermost request element to any standard component, (such as **WebServiceClient** or **HTTPConnector**), that could be called from within a transaction:

   ```
   outerTransactionId="${MDEX_TRANSACTION_ID}"
   ```

   This ensures that the component behaves like a Latitude-specific component: when this value is non-empty, the component will run within a transaction; when this value is empty, it will be ignored and the component will run outside a transaction.
3. Modify any standard **RunGraph** components that may possibly run within a transaction by specifying `MDEX_TRANSACTION_ID` in the **Graph parameters to pass** field (this field accepts a semicolon-delimited list).
4. Finally, configure a **Transaction RunGraph** connector to reference one or more graphs, and run it.

   LDI will start a transaction named `transaction` (lowercase, case-sensitive) and run all sub-graphs within this transaction.

Related Links

When to use transactions on page 18

This topic discusses transactions and provides recommendations for when it is useful to run your LDI graphs inside transactions as opposed to running graphs that do not utilize them.

Transaction graphs in the Latitude Quick Start project

The Latitude Quick Start project provides three transaction graphs that you can use in your projects.

The Begin Transaction, Commit Transaction and Rollback Transaction graphs reference the string `transaction` as the value of the transaction ID.

Begin Transaction graph

The Begin Transaction graph uses the Transaction Web Service's `startTransactionOperation` to begin a transaction. The request structure of the graph's **WebServiceClient** component has the following request:

```
<ns:request xmlns:ns="http://www.endeca.com/MDEX/transaction/2011">
  <ns:startTransactionOperation id="${MDEX_TRANSACTION_ID}"/>
</ns:request>
```

If the operation succeeds, then the MDEX Engine enters transaction mode. In transaction mode, queries and updates that do not have the transaction ID are rejected and updates applied within the transaction do not propagate across an MDEX Engine cluster.

Keep in mind that the transaction opened by the Begin Transaction graph must eventually be ended by the Commit Transaction graph, or the Rollback Transaction graph.

Commit Transaction graph

The Commit Transaction graph uses the Transaction Web Service's `commitTransactionOperation` to end a transaction. The request structure of the graph's **WebServiceClient** component has the following request:

```
<ns:request xmlns:ns="http://www.endeca.com/MDEX/transaction/2011">
  <ns:commitTransactionOperation id="${MDEX_TRANSACTION_ID}"/>
</ns:request>
```

If a transaction of the given ID is in progress and if the operation succeeds, the MDEX Engine will exit transaction mode. The MDEX Engine will once again accept unqualified queries and any updates applied during the transaction will be pushed out across the MDEX Engine cluster.

Rollback Transaction graph

The Rollback Transaction graph uses the MDEX Engine `admin?op=rollback` operation to roll back a transaction. In the event that a running transaction fails, this operation lets you roll back to the previously-committed version of the MDEX Engine index and stop the transaction.

The **URL** field of the **HTTP connector** component has this value:

```
http://${MDEX_HOST}:${MDEX_PORT}/admin?op=rollback&outerTransaction¬
Id=${MDEX_TRANSACTION_ID}
```

The results of the `admin?op=rollback` operation are logged in the `stdout/stderr` log of the MDEX Engine.

Creating a Transaction RunGraph graph

This section describes how to build an LDI graph that uses the **Transaction RunGraph** connector to run a series of graphs within a single atomic transaction.

To run one ore more graphs within a transaction, create a master graph using the **Transaction RunGraph** connector.

The **Transaction RunGraph** connector works as follows:

1. It starts an outer transaction using the Transaction Web Service.
2. It runs a series of defined sub-graphs within that transaction.
3. It commits the transaction when all the graphs have successfully finished.

For the duration of the transaction, **Transaction RunGraph** is designed to override the transaction ID specified in `workspace.prm` with the string `transaction`. Components that run within this graph automatically pick up this ID.

In addition, you can configure the **Transaction RunGraph** connector to react to unsuccessful runs, such as it can roll back the transaction.

In this section, a sample **Transaction RunGraph** graph will be built to run the two graphs that load the standard attribute and managed attribute schemas into the MDEX Engine.

Related Links

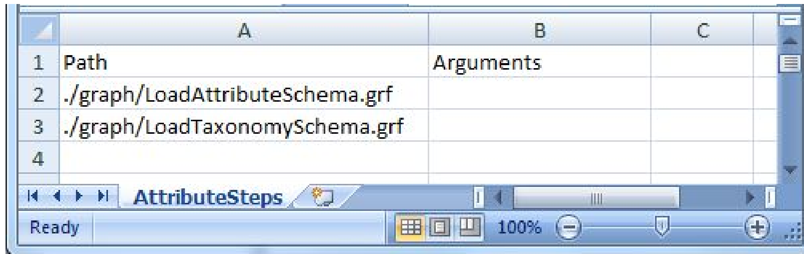
Transaction RunGraph connector on page 161

Use this connector to run LDI graphs, similar to the standard **RunGraph** component available with the LDI. Unlike the standard **RunGraph**, **Transaction RunGraph** starts the outer transaction and runs multiple sub-graphs within that transaction.

Format of the steps input file

The input file for the **Transaction RunGraph** connector defines which graphs will be run by it.

The `AttributeSteps.csv` sample input file used to list the graphs looks like this:



	A	B	C
1	Path	Arguments	
2	./graph/LoadAttributeSchema.grf		
3	./graph/LoadTaxonomySchema.grf		
4			

The first line (the header row) of the sample file has two header properties:

```
Path,Argument
```

The actual names of the header properties can be different from the names used here. The properties are delimited (for example, by the comma in the sample CSV file). After the header row, the second and following rows in the input file contain the input values:

- The `Path` column lists the path names of the graphs to be run by the **Transaction RunGraph** component. The order in which the graphs are listed is the order in which they are run.
- The `Arguments` column specifies any graph command-line arguments. No arguments are specified in our example input file.

After creating the file, copy it into the **data-in** folder.

Transaction ID in the workspace.prm file

When running graphs in a transaction environment, specify an outer transaction ID in your `workspace.prm` file by setting it in the `MDEX_TRANSACTION_ID` variable, as in this example:

```
MDEX_TRANSACTION_ID=
```

Leaving the value empty is important. It allows the sub-graphs to be run outside of transactions if needed, as well as within a transaction. When the sub-graphs are run within a **Transaction RunGraph**, the master graph overrides the ID with the string `transaction` for the duration of the transaction. When the sub-graphs are run independently of the master graph and outside of a transaction, the empty value from `workspace.prm` is used, which enables the graphs to ignore the ID attribute in the request to the MDEX Engine.

Adding components to the transaction graph

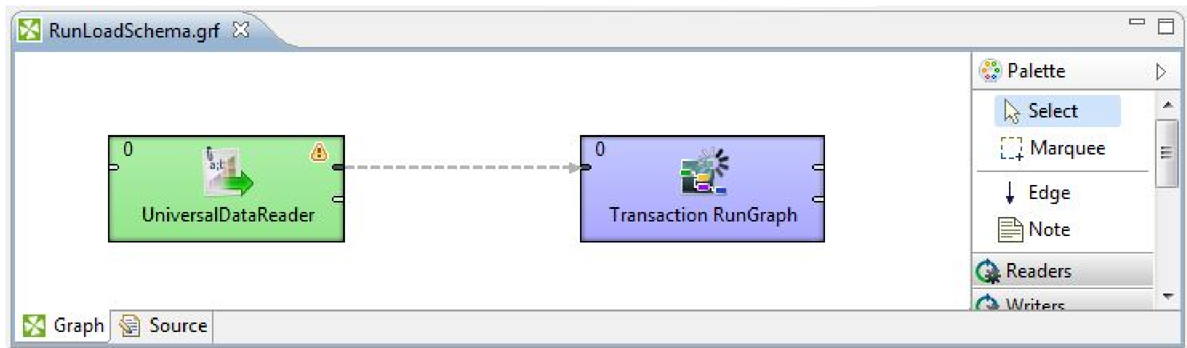
This topic describes the two LDI components that must be added to the transaction graph.

This procedure assumes that you have created an empty graph (named RunLoadSchema in our example).

To add components to the transaction graph:

1. In the Palette pane, drag the **UniversalDataReader** component from the **Readers** section.
2. In the Palette pane, drag the **Transaction RunGraph** component from the **Latitude** section.
3. In the Palette pane, click **Edge** and use it to connect the components.
4. Save the graph.

At this point, the Graph Editor with the connected components should look like this:



Configuring the Reader for the transaction input file

This task describes how to configure the **UniversalDataReader** component to read in the file that lists the graphs to be run.

This procedure assumes that you have created the RunLoadSchema graph and added the **UniversalDataReader** component. It also assumes that you have added the `AttributeSteps.csv` input file to the project's **data-in** folder.

To configure the Reader component for the run-graphs input file:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the transaction input file (`AttributeSteps.csv` in our example) and click **OK**.
3. Change the **Number of skipped records per source** field set to 1.

   The reason is that we do not want the first row (the header property row) to be read in as data.
4. Optionally, use the **Component name** field to provide your own name for the component.
5. Click **OK** to apply your configuration changes to the Reader component.
6. Save the graph.

The next step is to configure the Reader's Edge metadata.

Configuring the Edge for Reader component

The Edge for the Reader component must be configured with a Metadata definition.

This Metadata definition task will use the Metadata Editor. In the procedure, the column names will be extracted from the input file via a reparsing operation.

To configure the Metadata definition for the Reader Edge in the transaction graph:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.

3. In the URL Dialog:, browse for the PDR input file, select it, and click **OK**.
   a) Double-click the **data-in** folder.
   b) Select the transaction input file and click **OK**.

   You are returned to the Flat File dialog.
4. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**. The input data is loaded into the Metadata Editor, with the properties named Field1, Field2, and so forth.
5. In the middle pane of the Metadata Editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.

   The correct property names are now displayed in the upper and middle panes of the Metadata Editor, which should look like this:

#	Name	Type	Delimiter
1	Record: AttributeSteps_csv	delimited	,
1	Path	string	,
2	Arguments	string	\r\n

Filter:

6. In the upper pane of the Metadata Editor:
   a) Optionally, click the **Record** Name field and change the name of the metadata to a more descriptive name.
   b) Make sure that the **Type** field of all the properties is set to type **string**.
   c) Verify that the fields have the correct delimiter character set (which is the comma for our example).
7. When you have input all your changes, click **Finish**.
8. Save the graph.

The next step is to configure the **Transaction RunGraph** connector.

Configuring the Transaction RunGraph connector

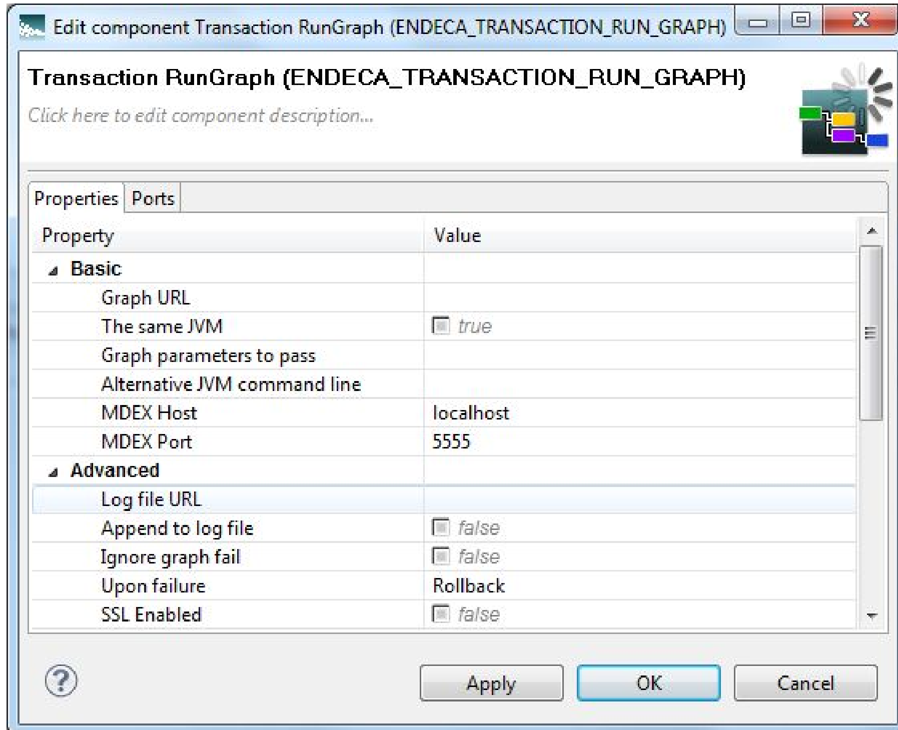
This topic describes how to configure the **Transaction RunGraph** connector to run the graphs.

This procedure assumes that you have created a graph and added the **Transaction RunGraph** connector.

To configure the **Transaction RunGraph** connector:

1. In the Graph editor, double-click the **Transaction RunGraph** component.

   The Edit Component dialog is displayed.



2. In the Edit Component dialog, make these settings:
   - **MDEX Host**: Enter the host name of the machine on which the MDEX Engine is running. `lo¬calhost` can be used as the name.
   - **MDEX Port**: Enter the number of the port on which the MDEX Engine is listening.
   - **Upon failure**: Select the action that the component should take upon a transaction failure: **Rollback** (roll back to the state before the transaction had started, and commit the transaction), **Commit** (commit those changes that have been made successfully before the failure occurred, and commit the transaction), or **Do nothing** (nothing is done, which means you may need to manually stop the outer transaction).
   - **SSL Enabled**: Toggle this field to `true` only if the MDEX Engine is SSL enabled.

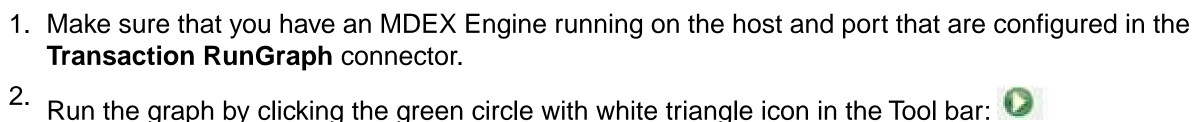
   You can leave the other settings at their defaults.
3. When you have input all your changes, click **OK**.
4. Save the graph.

The final step is to run the transaction graph.

Running the transaction graph

After creating the transaction graph and configuring its components, you can run the graph to run its sub-graphs in a transaction environment.

To run the transaction graph:

1. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Transaction RunGraph** connector.
2. Run the graph by clicking the green circle with white triangle icon in the Tool bar: 

As the graph runs, the process of the graph execution is listed in the Console Tab. The execution is completed successfully when you see this final output message:

```
INFO  [main] - Execution of graph successful !
```

Committing an outer transaction

To manually commit an outer transaction that failed to commit successfully, run the **RollBackTransaction** or **CommitTransaction** graph from the Latitude Quick Start project, or issue an `/admin?op=rollback&outerTransactionId="ID"` command on the MDEX Engine server, specifying the transaction ID.

In some instances, you may have a graph that starts an outer transaction but fails to commit it. This may happen, for example, when you are creating a new graph and troubleshooting its sub-graphs. If any of the sub-graphs fail, the entire graph running a transaction may fail also.

Since only one outer transaction can be in progress at a time, if the graph running an outer transaction fails, you cannot run any other graphs that start transactions until the transaction that is in progress is committed. In such cases, it is useful to know how to commit an outer transaction manually.

Typically, you may need to close an already running transaction after you receive a transaction-related error, when trying to run one of your graphs. To identify whether an outer transaction is currently running, issue an `http://mdex/admin?op=ping` request. An HTTP code 403 means that a transaction is open.

To commit an outer transaction:

Do one of the following:

- Run the **RollBackTransaction** graph that is included as part of the Latitude Quick Start project. This graph rolls back all the changes from this transaction and commits the transaction. This graph uses the transaction ID string `transaction`. If the ID of your transaction is different, change the ID in the `workspace.prm` file for your project. Running this graph is equivalent to running the `/admin?op=rollback&outerTransactionId="myID"` command. Note that `myID` defaults to the `transaction` string when run within a **Transaction RunGraph**.
- Run the **CommitTransaction** graph that is included as part of the Latitude Quick Start project. This graph commits those changes that succeeded within a transaction, and ignores the rest, and then commits the transaction. If you choose to run this graph, examine which changes have been applied and which have failed. This graph uses the ID string `transaction`. If the ID of your transaction is different, change the ID in the `workspace.prm` file for your project.
- Issue an `/admin?op=rollback&outerTransactionId="myID"` command on the MDEX Engine server that has a transaction open, where *`myID`* is the ID of the transaction.

Running any of these options allows you to commit a transaction that has failed to commit successfully.

Related Links

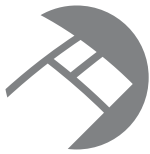
Transaction-related errors on page 172

You may receive various transaction-related errors if you attempt to overlap running graphs wrapped in transactions with graphs that do not start an outer transaction.

Performance impact of transactions

Running an outer transaction does not affect performance of the MDEX Engine.

However, be aware that a transaction that is in progress (especially if it is running update operations on a large amount of data), will increase the disk usage resulting in higher disk high-water mark values (Linux).



Chapter 4

Full Initial Index Load of Records

This section describes how to create an LDI project and a graph that will perform a full initial index load of records into the MDEX Engine.

Overview of the full initial index load

This section walks you through the various tasks involved in creating a graph that can perform initial load of source records into the MDEX Engine.

The task that this section covers is how to perform an initial ***full index load*** of your source records into the MDEX Engine. (Full index loads are also known as ***baseline updates*** .) As the source records are ingested, they are converted into Endeca records and are indexed by the MDEX Engine.

This process assumes that:

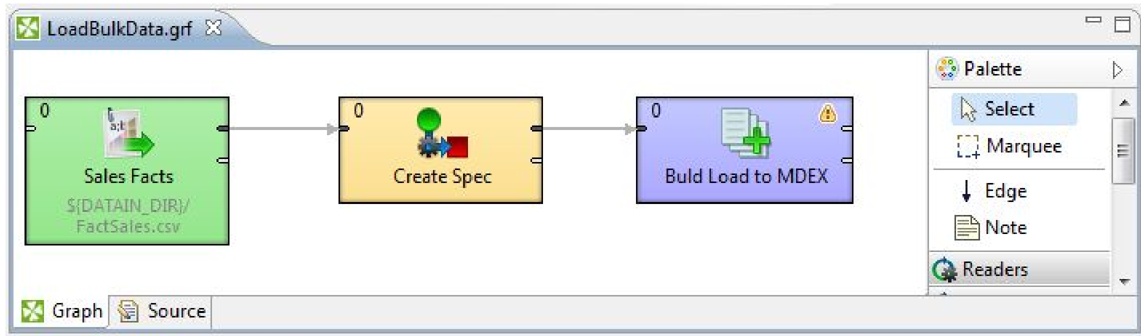
- The MDEX Engine is empty of user source data.
- You have already loaded your attribute schema (PDRs and DDRs) into the MDEX Engine. The load-schema procedure is documented in the section *Loading the Attribute Schema* in this guide.

Note: You can also initially your data without having to first load your attribute schema. However, if you do so, you will not have control over the default values for the standard attributes that are created. For this reason, it is recommended that you first load your attribute schema data before loading your user source data.

Sample full index load graph

The Latitude Sample Application has an extensive graph (named LoadData) that uses the **Bulk Add/Replace Records** connector. The LoadData graph inputs ten data source files and uses **ExtHashJoin** joiner components to join all the source data.

In order to simplify the description of a bulk load graph, a subset of the LoadData graph is used in this chapter. This sample subset graph (named LoadBulkData) uses three components:



The three components are:

- The **Sales Facts** component is a **UniversalDataReader** that reads in sales transaction records from one source data file.
- The **Create Spec** component is a **Reformat** component that creates the primary-key attribute for the records.
- The **Bulk Load to MDEX** component is a **Bulk Add/Replace Records** connector. This Endeca Latitude connector sends the records to the Bulk Load Interface of the MDEX Engine.

The source data is sales transaction information stored in a CSV file, with each source record having multiple columns that are delimited by the comma character. The format of the source data is explained in a following topic. You can, of course, use other source formats, including reading from a database. These other input formats may require other types of readers, such as the **DBInputTable** reader.

Creating a project

You must create an LDI project in which you will build your graph.

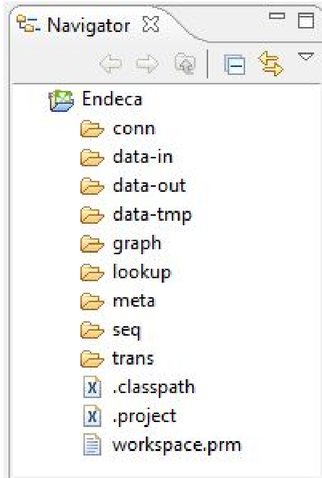
If you already have a project, you can re-use it for your graph. In other words, a project can have multiple graphs configured in it.

To create a new LDI project:

1. From the File menu, select **New** > **CloverETL Project**.
2. In the **New CloverETL project** dialog, enter a name for the project in the **Project name** field.

   You can leave the **Use default location** box checked.
3. Click **Next** and then click **Finish**.

Your new project is displayed in the Navigator pane, as in this example that shows the Endeca1 project in an expanded format:



Note that the Outline pane is empty, as is the Graph Editor.

Source data format

You can load source data from a variety of formats.

Your Endeca applications will most often read data directly from one or more database systems, or from database extracts. Input components load records in a variety of formats including delimited, JDBC, and XML. Each input component has its own set of configuration properties. One of the most commonly used type of input component loads data stored in delimited format.

The format used as an example in this chapter is a two-dimensional format similar to the tables found in database management systems. Database tables are organized into rows of records, with columns that represent the source properties and property values for each record. (This type of format is often called a rectangular data format.) The source records are stored in a CSV file named `FactSales.csv` (the file is in the `data-in` folder of the Latitude Sample Application).

The following image, which shows the beginning lines of the `FactSales.csv` input file, illustrates how the source data is organized in a two-dimensional format:

	A	B	C	D	E
1	FactSales_ProductKey	FactSales_OrderDateKey	FactSales_DueDateKey	FactSales_ShipDateKey	FactSales_ResellerKey
2	349	20050701	20050713	20050708	676
3	350	20050701	20050713	20050708	676
4	351	20050701	20050713	20050708	676
5	344	20050701	20050713	20050708	676
6	345	20050701	20050713	20050708	676
7	346	20050701	20050713	20050708	676
8	347	20050701	20050713	20050708	676
9	229	20050701	20050713	20050708	676
10	235	20050701	20050713	20050708	676
11	218	20050701	20050713	20050708	676

FactSales

You specify the location and format of the source data to be loaded in the LDI reader component in the graph. The reader component passes the data to the Endeca connector, which is configured to connect to either the Data Ingest Web Service (DIWS) or the Bulk Load Interface, both of which reside

on the MDEX Engine. The records are then loaded into the MDEX Engine in batches of a pre-configured size. During the ingest operation, each source row is transformed into an Endeca record with a key-value pair for each non-null source column. The MDEX Engine then indexes the records for use during search queries.

Primary key attribute

You will be using one of the Endeca standard attributes as the primary-key attribute for the records. (The primary-key property is also known as the record spec property.) The primary-key property must be a unique, single-assign property. For more information on primary keys, see the *Data Ingest API Guide*.

In our sample graph, the `FactSales.csv` input file does not have a field that contains unique values. Therefore, the **Create Spec** component creates the FactSales_RecordSpec primary-key attribute by concatenating two attributes. The name of the primary-key attribute will be specified in the Metadata definition for the Edge component.

Use of hyphens in input property names

Although the MDEX Engine will accept attribute names with hyphens (because hyphens are valid NCName characters), the Designer will not accept source property names with hyphens as metadata. Therefore, if you have a source property name such as "Ship-Date", make sure you remove the hyphen from the name.

Using multi-assign data

Your source data may have multi-assign properties, that is, a property that has more than one value. For example, instead of having two properties (say, Color1 and Color2) in which each property has only one value, you can instead have one property (say, Color) with multiple values, as in this simple example:

```
ComponentID|Color|Size
123|Blue|Medium
456|Blue;Red|Small
789|Red;Black;Silver|Large
```

In the example, the pipe character (|) is the delimiter between the properties, while the semi-colon (;) is the delimiter between multiple values in a given property. For example, the Color property for record 789 has values of "Red", "Black", and "Silver".

When configuring the Writer component, you can then specify (in the Writer Edit Component dialog) that the semi-colon is to be used as the delimiter for multi-assign properties.

Keep in mind that an Endeca property that is multi-assign must have the `mdex-property_IsSingleAssign` property set to `false` in its PDR. The default value of the property is `false`, which means the property is enabled for multi-assign by default.

Adding the source data to the project

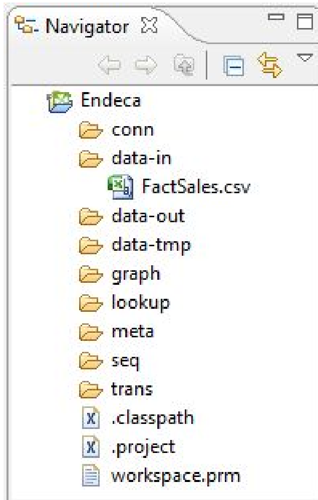
The easiest way to add your source data is to copy it into the project's `data-in` directory.

This procedure assumes that you are copying a CSV (comma-separated value) file named `FactSales.csv`, which contains the delimited records. The comma character is the delimiter. You can use other input file formats, such as a CSV (comma-separated value) file. The source records are stored in a CSV file named `FactSales.csv` (the file is in the `data-in` folder of the Latitude Sample Application).

To add the source data file to your Data Integrator project:

1. Locate the project's `data-in` directory.
   To find its location, right-click on **data-in** (in the Navigator pane) and select **Properties**.
2. Copy the source file into the `data-in` directory.
   You use the Designer GUI to paste the file into the **data-in** folder in the Navigation pane.
3. In the Navigator pane, right-click on **data-in** and select **Refresh**.

After refreshing the Navigator pane, it should look like this example:



As the example shows, the `FactSales.csv` is now available to the project's graphs.

Creating a graph

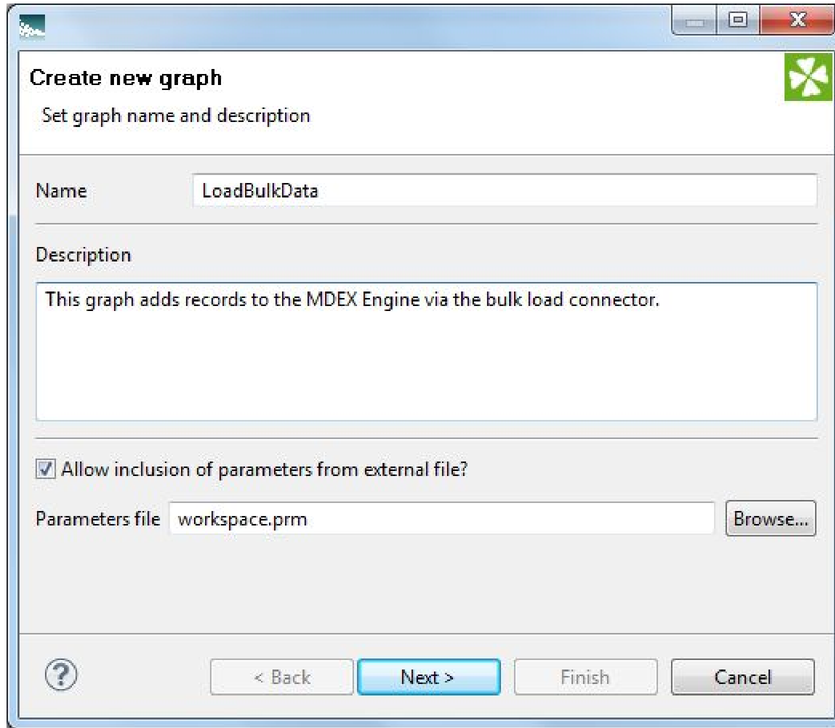
This task describes how to create an empty graph.

An empty graph is one that does not have any transformation components. The only prerequisite for this task is that you must have created a Data Integrator Designer project. A project can have multiple graphs, but only one graph will be created for the project in this chapter.

To create an empty graph:

1. In the Navigator pane, right-click the **graph** folder.
2. Select **New** > **ETL Graph**.
   The **Create new graph** dialog is displayed.
3. In the **Create new graph** dialog:
   a) Type in the name of the graph, such as **LoadBulkData**.
   b) Optionally, type in a description.
   c) Leave the **Allow inclusion of parameters from external file** box checked.
   d) Click **Next** when you finish.

   After this step, the **Create new graph** dialog should look like this example:



4. In the **Output** dialog, click **Finish**.

As a result of creating the graph, the following changes appear in the perspective:

- The Graph window will have an empty graph, with the graph name as the name of the window.
- The Properties window (below the Graph window) will show the graph properties.
- The Outline pane will show a list of items (most of them are empty).
- The Palette pane will list the available graph components, including the Endeca components.

The next task is to add components to the graph.

Adding components to the graph

You need to add components to the empty graph in order for it to process the input source data and output it to the MDEX Engine.

The components to be added are:

- A *Reader* is a graph component that reads in source data. In our example, the **UniversalDataReader** component is used because it can read in data from CSV files.
- A *Transformer* component can transform the incoming data before it is sent to the next component. In our example, the **Reformat** component is used to create a new primary-key attribute from two existing attributes. Note that this component would not be necessary if you were using an existing attribute as the primary-key attribute.
- A *Writer* is a graph component that is responsible for outputting data from the Transformation. The **Bulk Add/Replace Records** connector is used because we are doing a bulk load of the data.

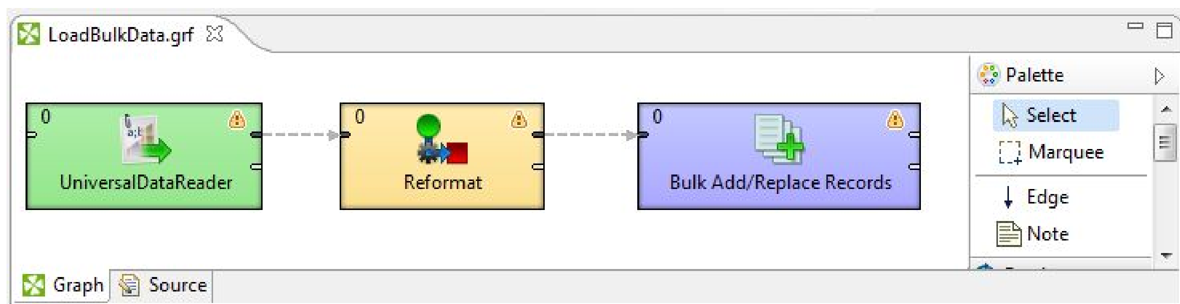
In addition, an *Edge* component will be added to connect the components. The configurations for all components are covered in this chapter.

To add components to the graph:

1. In the Palette pane, open the **Readers** section and drag the **UniversalDataReader** component into the Graph Editor.
2. In the Palette pane, open the **Transformers** section and drag the **Reformat** component into the Graph Editor.
3. In the Palette pane, open the **Latitude** section and drag the **Bulk Add/Replace Records** component into the Graph Editor.
4. In the Palette pane, click **Edge** and use it to connect the components.

   After connecting the components, you can get out of Edge selection mode by hitting **Escape** on your keyboard or clicking on **Select** in the Palette.
5. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the connected components should look like this:



The next tasks are to configure these components for the source data and for a connection to the MDEX Engine.

Configuring the components

This section describes how to configure the **UniversalDataReader**, **Reformat**, and **Bulk Add/Replace Records** components, as well as the metadata for the two Edge components.

The components will be configured in this order:

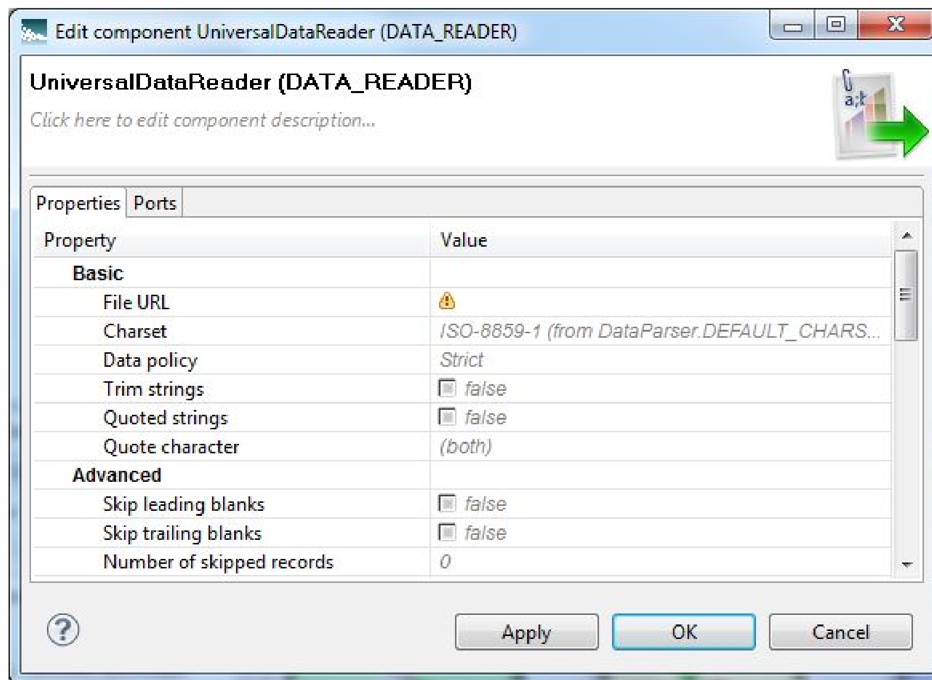
1. The **UniversalDataReader** component.
2. The metadata for the Edge component between the **UniversalDataReader** and **Reformat** components.
3. The metadata for the Edge component between the **Reformat** and **Bulk Add/Replace Records** components.
4. The **Reformat** component.
5. The **Bulk Add/Replace Records** component.

Configuring the Reader component

This task describes how to configure the Reader component to read in the source data.

This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the data source file to the project's **data-in** folder.

The Reader Edit Component dialog is where you configure the Reader as to how it should handle the source data:



To configure the Reader component:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the source data file and click **OK**.
3. Check the **Quoted strings** box so that its value changes to `true`.

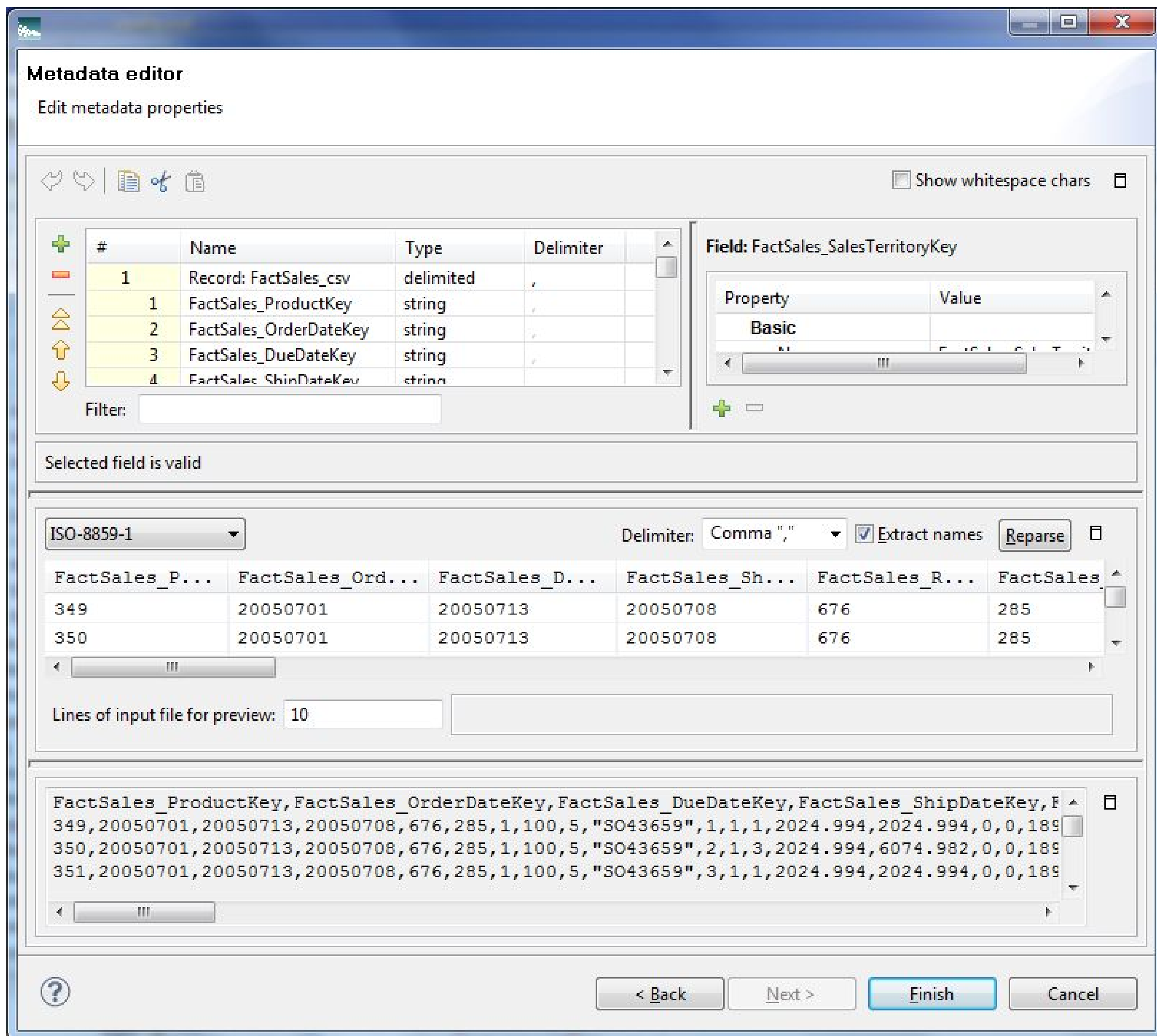
   If set to `true`, delimiter characters inside the quoted strings are ignored (not treated as delimiters) and the quotes are removed.
4. Leave the **Number of skipped records** field as 0.
5. Optionally, you can use the **Component name** field to provide a customized name (such as "Sales Facts") for this component.
6. Click **OK** to apply your configuration changes to the Reader component.
7. Save the graph.

Configuring metadata for the Reader Edge

The Edge component (between the Reader and Reformat components) has to be associated with a Metadata definition so it knows what fields of data are being passed from the Reader component to the Reformat component.

By setting the Metadata definition, you are actually defining the properties that will be tagged on the records.

Most of the metadata configuration will be done in the Metadata editor:

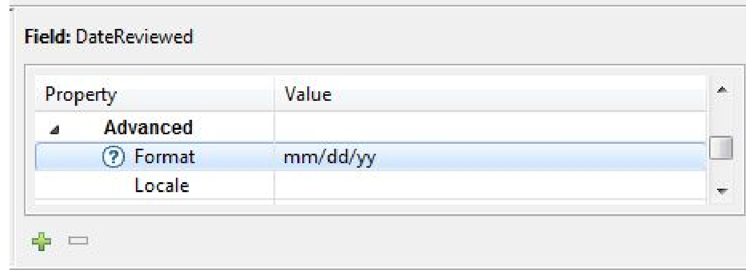


You will be using this editor in Steps 6 and 7 of this procedure.

To configure the Metadata definition for the Reader Edge:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
2. Select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
3. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
4. In the URL Dialog:, double-click the **data-in** folder, select the `FactSales.csv` data source file, and click **OK**.
   As a result, the Flat File dialog is populated with source data from the data file.
5. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.
   The Metadata Editor is displayed, as in the example above.
6. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.

7. In the Record pane of the Metadata editor, make these changes:
   a) Click the **Record** Name field and change the default value to a name that is appropriate for your data, such as **FactSales** for the sales transactions data set. In this example, **Record:FactSales** will be the resulting Name value.
   b) If your source data has date properties, you should set their type to **date** (the type may be set to **string** by the Designer). Then use the **Format** field (in the Field pane on the right) to set the appropriate value, as in this example:



   c) Verify that the other properties have their property type set correctly.

      For example, change the FactSales_UnitPriceDiscountPct and FactSales_DiscountAmount property types from `integer` to `number` in our sample metadata.
   d) Verify that all properties have the correct delimiter character set (which is the comma character in our source data).
   e) When you have input all your changes, click **Finish**.
8. Save the graph.

The Metadata definition for the Reader Edge component is now set.

Configuring the Reformat component

Reformat components are used to transform incoming records and send them to the specified port.

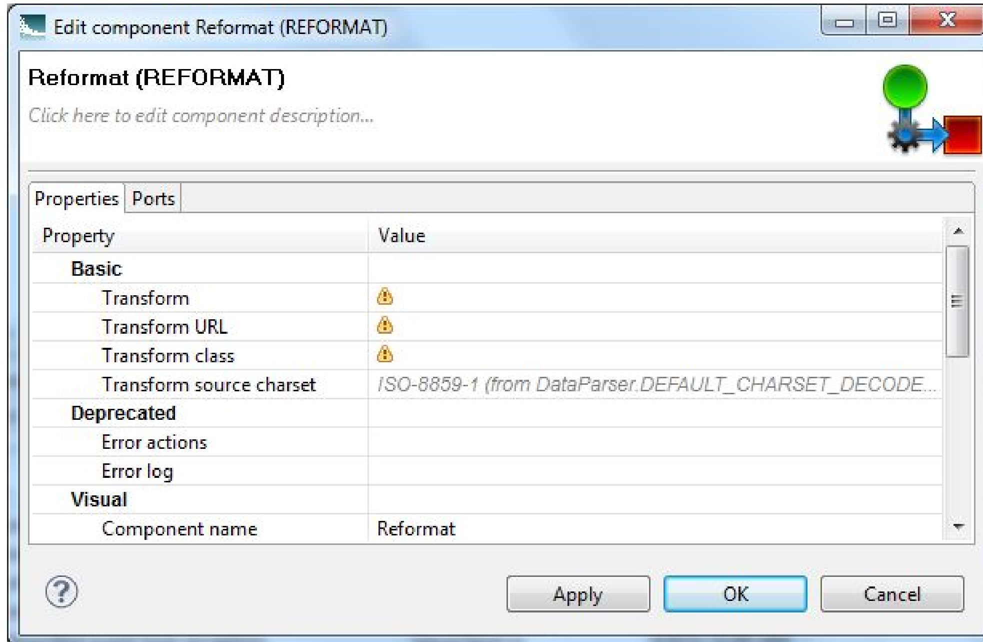
The transformation is done by the CTL function in the **Reformat** component. Return values of the transformation are the numbers of output port(s) to which data record will be sent.

As mentioned earlier, the `FactSales.csv` input file for the graph does not have a field that can be used as the primary-key attribute. Therefore, this **Reformat** component creates a new attribute (named `FactSales_RecordSpec`) by concatenating two existing attributes. After creation, the `FactSales_RecordSpec` attribute is used in the Spec Attribute property of the **Bulk Add/Replace Records** connector.

To configure the **Reformat** component:

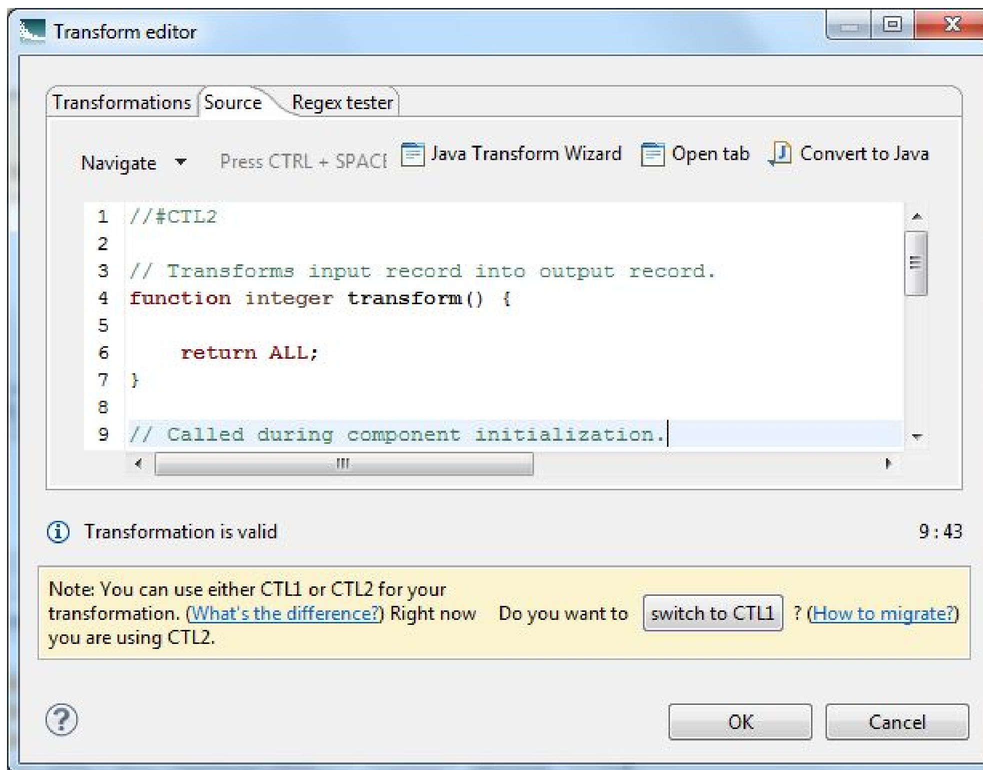
1. In the Graph window, double-click the **Reformat** component.

   The Reformat Edit Component dialog is displayed.



2. Single-click in the **Transform** field and then click the **...** button.
   The Transform editor is displayed.
3. Click the **Source** tab in the editor.

   The CTL template for the transform function is shown.



4. Modify the CTL script so that it looks like the following example. Note that the final line of the CTL transformation (just before the `return ALL` line) creates the `FactSales_RecordSpec` attribute.

```
//#CTL2
// Transforms input record into output record.
function integer transform() {
 $0.* = $0.*;
 $0.FactSales_RecordSpec = $0.FactSales_SalesOrderNumber+"-"+$0.Fact¬
Sales_SalesOrderLineNumber;

 return ALL;
}
```

   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.

5. When you have finished your edits, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
6. Optionally, you can use the **Component name** field to provide a customized name (such as "Create Spec") for this component.
7. Click **OK** to apply your configuration changes to the component.
8. Save the graph.

The two messages listed above should disappear once you configure the **Reformat** component Edge metadata.

Configuring metadata for the Reformat Edge

The metadata for the Edge component (between the Writer and Reformat components) also has to configured.

This task will use the same data source file (named `FactSales_RecordSpec`) as the Reader Edge. The main difference is that an additional property will be manually added to the metadata.

To configure the Metadata definition for the **Edge** component:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the `FactSales.csv` data source file and click **OK**.
4. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.
5. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.
6. In the Record pane of the Metadata editor, make these settings:
   a) Click the **Record** Name field and change the default value to a name that is appropriate for your data, such as **FactSalesRecSpec**.

b) Make sure that the **Type** fields of these properties match those of the Reader Edge metadata properties.

c) Create a new property (for the primary-key attribute) clicking the + button (which creates a new field with a default name such as `field25`), typing the `FactSales_RecordSpec` name of the new property, and leaving the **Type** field as `string`.

d) Verify that all properties have the correct delimiter character set (which is the comma character for this source data).

e) When you have input all your changes, click **Finish**.

7. Save the graph.

Now the metadata for the two Edge components is set, the next step is to configure the **Bulk Add/Replace Records** component.

Configuring the Bulk Add/Replace Records connector

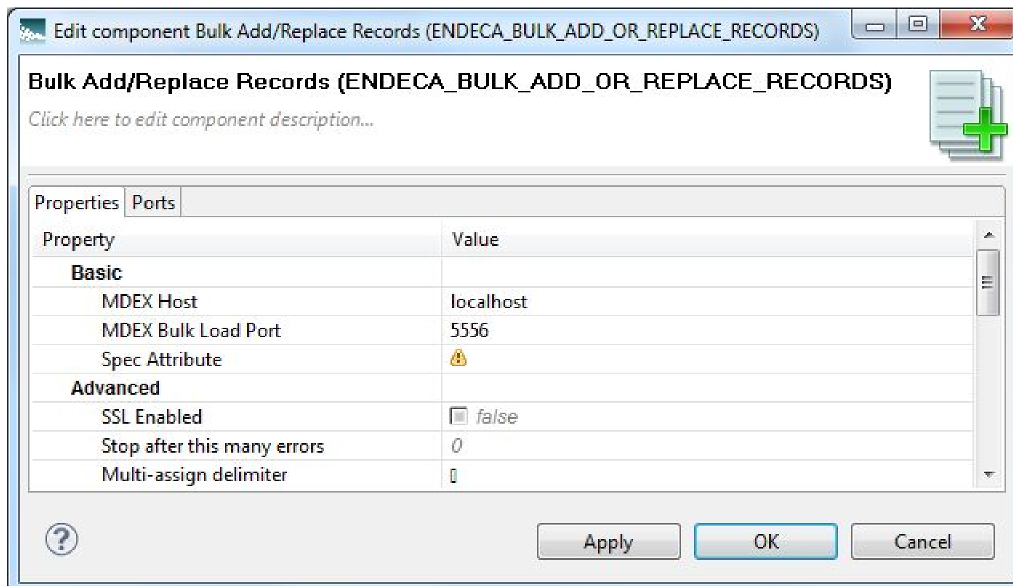
This topic describes how to configure the **Bulk Add/Replace Records** connector for the bulk loading of records.

This procedure assumes that you have created a graph and added the **Bulk Add/Replace Records** connector.

Note: When using the **Bulk Add/Replace Records** connector, it is a good idea to use the Dgraph `--bulk_load_port` flag when starting the MDEX Engine.

To configure the **Bulk Add/Replace Records** connector:

1. In the Graph editor, double-click the **Bulk Add/Replace Records** component.
   The Edit Component dialog is displayed.



2. In the Writer Edit Component dialog, enter these settings:

   - **MDEX Host**: Enter the host name of the machine on which the MDEX Engine is running. You can specify **${MDEX_HOST}** if you have the `MDEX_HOST` variable defined in the `workspace.prm` file for your project.

- **MDEX Bulk Load Port**: Enter the bulk load port on which the MDEX Engine is listening. You can specify **${MDEX_BULK_PORT}** if you have the `MDEX_BULK_PORT` variable defined in the `workspace.prm` file for your project.

  Note that the MDEX Engine opens up the bulk load port on 5556 (when the Dgraph `--port` command flag is not used) or at `--port` + 1 (if the `--port` flag is used). Make sure that you do not specify the MDEX Engine's HTTP port (`--port`) because a network connection error will be thrown by the connector.

- **Spec Attribute**: Enter the name of the standard attribute that is the primary key (record spec) for the records.
- **SSL Enabled**: Toggle this field to `true` only if the MDEX Engine is SSL enabled.
- **Stop after this many errors**: Optionally, you can specify the maximum number of ingest errors that can occur before the load operation is terminated.
- **Multi-assign delimiter**: Optionally, you can specify the character that separates multi-assign values in an input property. Keep in mind that this delimiter is different from the delimiter that separates properties.

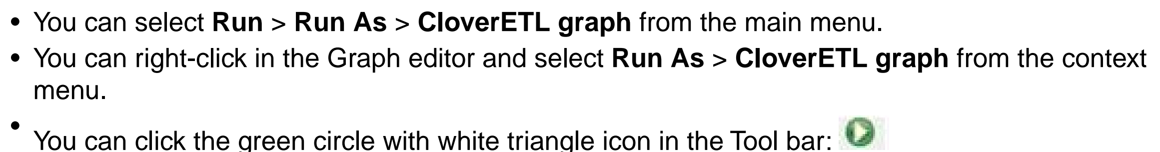
3. When you have input all your changes, click **OK**.
4. Save the graph.

Running the graph to load records

Endeca recommends that you use a transaction graph to run the bulk load graph.

A transaction graph uses a **Transaction RunGraph** connector to safely run one or more graphs within the transaction environment of the MDEX Engine. This connector can start an outer transaction, run the set of graphs so that they succeed or fail as a unit, and finally commit the transaction (or roll it back upon failure).

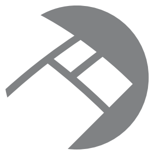
You can run a graph in one of three ways:

- You can select **Run** > **Run As** > **CloverETL graph** from the main menu.
- You can right-click in the Graph editor and select **Run As** > **CloverETL graph** from the context menu.
- You can click the green circle with white triangle icon in the Tool bar: 

To use a transaction graph to bulk load records:

1. Create a transaction graph as described in the section *Working with Transaction Graphs*.
2. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Bulk Add/Replace Records** connector.
3. Run the transaction graph using of the methods listed above.

As the graph runs, the process of the graph execution is listed in the Console Tab. The output lists the number of records that were read in by the **UniversalDataReader** component and the number of records that were sent to the MDEX Engine by the **Bulk Add/Replace Records** connector.



Chapter 5

Incremental Updates

This chapter describes how to create a Data Integrator graph that will perform an incremental update of records into the MDEX Engine.

Overview of incremental updates

You can incrementally update the data set in the MDEX Engine, including adding new records.

Using the **Add/Update Records** connector, you can perform these types of incremental updates:

- Add a brand-new record to the data set in the MDEX Engine.
- Update an existing record by adding key-value pairs.

Note that the **Add/Update Records** connector cannot load managed attribute values, nor can it delete records or record data.

Format of the incremental source input file

Because the assumption is that you are adding (or updating) records that are similar in format to what is already in the MDEX Engine, the format of the input will be very similar to the format of the input file for the full index load. For more information, see the topic titled "Source data format" in Chapter 2 ("Full Initial Index Load of Records") of this guide.

How updates are applied

The records to be added are considered totally additive. That is, if a record with the same primary key already exists in the MDEX Engine, the key-value pairs list of the added record will be merged into the existing record.

If an Endeca attribute with the same name already exists (but has a different assigned value), then the added key-value pair will be an additional value for the same property (multi-assign). For example, if the existing record has one standard attribute named **Color** with a value of "red" and the request adds a **Color** property with a value of "blue", then the resulting record will have two **Color** key-value pair assignments.

Keep in mind, however, that you cannot add a second value to a single-assign attribute. (That is, an attribute whose PDR has the `mdex-property_IsSingleAssign` set to `true`.) In the **Color** example, if **Color** were a single-assign attribute and the record already had one **Color** assignment, then an attempt to add a second **Color** assignment would fail.

When adding standard attributes, the operation works as follows for the new attribute:

- If the new attribute already exists in the MDEX Engine but with a different type, an error is thrown and the new attribute is not added.
- If the new attribute already exists in the MDEX Engine and is of the same type, no error is thrown and nothing is done.
- If the new attribute is supposed to be a primary-key attribute but a managed attribute already exists with the same name, an error is thrown and the new standard attribute is not added.

Note that updating a record can cause it to change place in the default order. That is, if you have records ordered A, B, C, D, and you update record B, records A, C, and D remain ordered. However, record B may move as a result of the update, which means the resulting order might end up as B,A,C,D or A,C,B,D or another order.

Adding components to the incremental updates graph

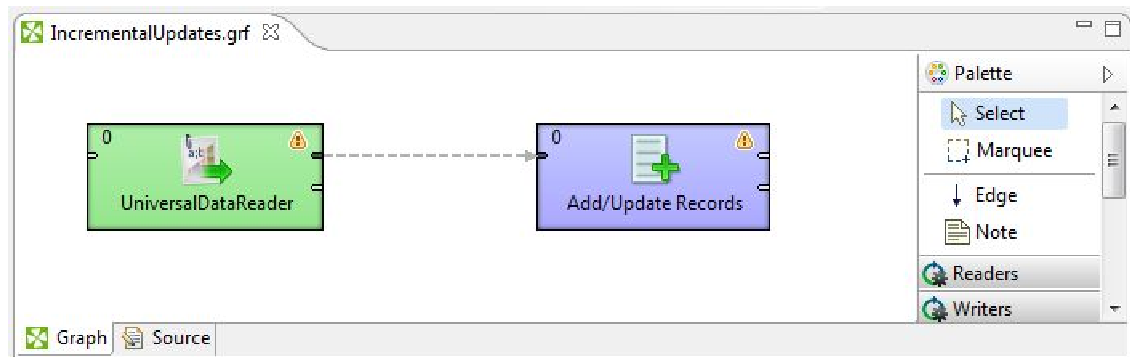
The graph for performing incremental updates requires a reader and the **Add/Update Records** connector.

This procedure assumes that you have created an empty graph.

To add components to a graph for incremental updates:

1. In the Palette pane, open the **Readers** section and drag the **UniversalDataReader** component into the Graph Editor.
2. In the Palette pane, open the **Latitude** section and drag the **Add/Update Records** connector into the Graph Editor.
3. In the Palette pane, click **Edge** and use it to connect the two components.
4. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the two connected components should look like this:



The next tasks are to configure the components.

Configuring the Reader and the Edge for incremental updates

The configuration of the incremental updates reader and edge components is almost identical to that of fresh index load graph.

This procedure assumes that you have added the incremental updates source file to the project's **data-in** folder.

To configure the **UniversalDataReader** and Edge components for incremental updates:

1. To configure the **UniversalDataReader** component for the incremental updates input file, use the same procedure as described in the topic titled "Configuring the Reader component" in Chapter 4 ("Full Initial Index Loads of Records") of this guide.

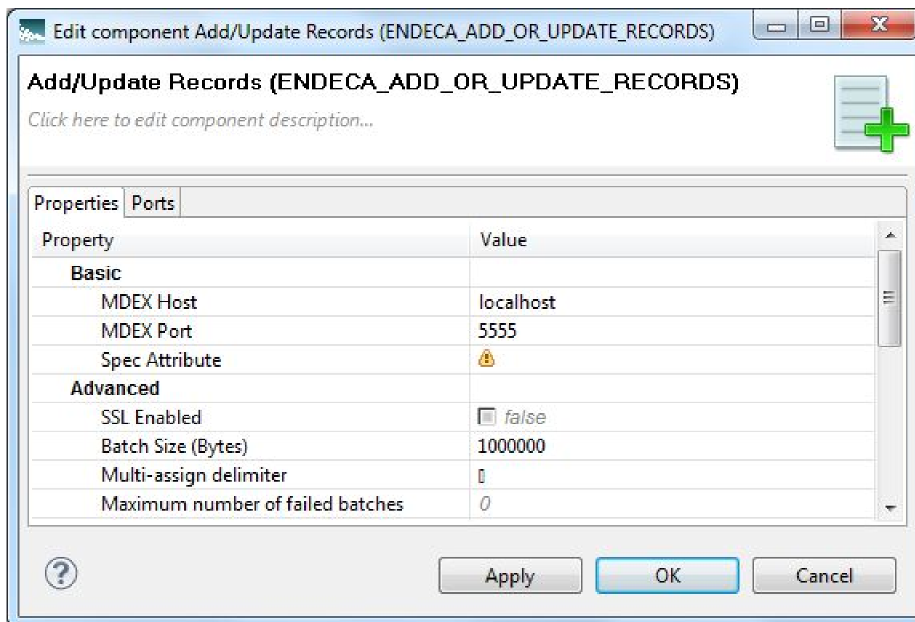
   The only difference is that you will be using your incremental updates file as the input file.
2. To configure the Edge component, use the same procedure as described in the topic titled "Configuring metadata for the Reader Edge" in Chapter 4 ("Full Initial Index Loads of Records") of this guide.
3. When you have finished your configuration, save the graph.

Configuring the Add/Update Records connector

You must configure the **Add/Update Records** connector with the location and port of the MDEX Engine, as well as the primary key for the records.

This procedure assumes that you have created a graph and added the **Add/Update Records** connector.

The Writer Edit Component dialog is where you configure the **Add/Update Records** connector:



To configure the **Add/Update Records** connector:

1. In the Graph window, double-click the **Add/Update Records** component.
   The Writer Edit Component dialog is displayed.
2. In the Writer Edit Component dialog, enter these mandatory settings in the Basic section:
   - **MDEX Host**: Enter the host name of the machine on which the MDEX Engine is running. You can specify **${MDEX_HOST}** if you have the `MDEX_HOST` variable defined in the `workspace.prm` file for your project.

- **MDEX Port**: Enter the port on which the MDEX Engine is listening for requests. You can specify **${MDEX_PORT}** if you have the `MDEX_PORT` variable defined in the `workspace.prm` file for your project.
- **Spec Attribute**: Enter the name of the property that is the primary key (record spec) for the records.

3. Still in the Writer Edit Component dialog, you can make these optional settings in the Advanced section:

    - **SSL Enabled**: Toggle this field to `true` if the MDEX Engine is SSL-enabled.
    - **Batch Size (Bytes)** : To change the default batch size (which is in bytes), enter a positive integer. Specifying 0 or a negative number will disable batching.
    - **Multi-assign delimiter**: Specify the character that separates multi-assign values in an input property. Keep in mind that this delimiter is different from the delimiter that separates properties.
    - **Maximum number of failed batches**: Enter a positive integer that sets the maximum number of batches that can fail before the ingest operation is ended. Entering 0 allows no failed batches.

4. When you have input all your changes, click **OK**.
5. Save the graph.

Running the incremental updates graph

After creating the graph and configuring the components, you can run the graph to load the incremental update records into the MDEX Engine.

To run the graph to load incremental updates:

1. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Add/Update Records** connector.
2. Run the graph using one of the run methods.

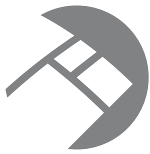
    For example, you can click the green circle with white triangle icon in the Tool bar:

As the graph runs, the process of the graph execution is listed in the Console Tab. The execution is completed successfully when you see final output similar to this example of adding five new records:

```
INFO  [WatchDog] - ----------------------** Final tracking Log for phase
[0] **---------------------
INFO  [WatchDog] - Time: 06/06/11 10:53:21
INFO  [WatchDog] - Node                   ID          Port      #Records
      #KB aRec/s   aKB/s
INFO  [WatchDog] - -------------------------------------------------------
-------------------------
INFO  [WatchDog] - UniversalDataReader    DATA_READER0
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                           Out:0           5
        3      5       3
INFO  [WatchDog] - Incrementals           ENDECA_ADD_OR_UPDATE_RECORDS0
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                            In:0           5
        3      5       3
INFO  [WatchDog] - ---------------------------------** End of Log **------
-------------------------
INFO  [WatchDog] - Execution of phase [0] successfully finished - elapsed
time(sec): 1
```

```
INFO  [WatchDog] - -----------------------** Summary of Phases execution
**---------------------
INFO  [WatchDog] - Phase#            Finished Status         RunTime(sec)
   MemoryAllocation(KB)
INFO  [WatchDog] - 0                 FINISHED_OK                        1
             4927
INFO  [WatchDog] - ------------------------------** End of Summary **-----
----------------------
INFO  [WatchDog] - WatchDog thread finished - total execution time: 1 (sec)
INFO  [main] - Freeing graph resources.
INFO  [main] - Execution of graph successful !
```

As the example shows, the Final Tracking Log lists the number of records that were read in by the **UniversalDataReader** component and the number of records (5 in this example) that were sent to the MDEX Engine by the **Add/Update Records** connector.



Chapter 6

Loading the Attribute Schema

This chapter describes how to load your PDR and DDR configuration files into the MDEX Engine.

About attribute schema files

The attribute schema for your application is defined by the PDR and DDR files in the MDEX Engine.

Each Endeca standard attribute is defined by its PDR (Property Description Record). Each Endeca managed attribute is defined by its own PDR and also by a DDR (Dimension Description Record).

If you are loading your source records without first loading your attribute schema, the MDEX Engine will automatically create the PDRs for your standard attributes, using the system default settings.

However, it is recommended that you create your own PDR and DDR input records and then use the Latitude Data Integrator Designer to load that schema into the MDEX Engine. This process uses the **UniversalDataReader** component to read in the schema files and the **WebServiceClient** component to load them into the MDEX Engine via the Configuration Web Service.

Loading the standard attribute schema

This topic provides an overview of the PDR load process.

From a high-level view, the steps you will follow to load your PDR schema into the MDEX Engine are:

1. Create the PDR input file. (Described in this chapter in the "Creating the PDR input file" topic.)
2. Either create a new project or re-use an existing one. (Not described in this chapter, as we will use the same project that was created in the "Creating a project" topic in Chapter 2.)
3. Create a graph and add the **UniversalDataReader**, **Reformat**, **Denormalizer**, and the **WebServiceClient** components. (Described in this chapter.)
4. Configure the components. (Described in this chapter.)
5. Run the graph. (Not described in this chapter, as this procedure is the same as described in the "Running the graph" topic in Chapter 2.)

Keep in mind that if you are also loading DDR records, you should first load the PDRs that will be associated with the DDRs (unless the appropriate PDRs have already been loaded into the MDEX Engine).

Format of the PDR input file

The PDR input file defines one or more Endeca standard attributes, with the specific settings of some PDR properties.

The sample input file used in this chapter looks like this:

	A	B	C	D
1	Key	DisplayName	TextSearch	SortOrder
2	FactSales_ProductKey	Product Key	FALSE	lexical
3	FactSales_OrderDateKey	Order Date Key	FALSE	lexical
4	FactSales_DueDateKey	Due Date Key	FALSE	lexical
5	FactSales_ShipDateKey	Ship Date Key	FALSE	lexical
6	FactSales_ResellerKey	Reseller Key	FALSE	lexical
7	FactSales_EmployeeKey	Employee Key	FALSE	lexical
8	FactSales_PromotionKey	Promotion Key	FALSE	lexical

AttributeMetadata

The first line (the header row) of the sample file has these header properties:

```
Key,DisplayName,TextSearch,SortOrder
```

The actual names of the header properties in your input file can be different from the names used here (for example, you can use `AttrName` instead of `Key`). The properties are delimited (for example, by the comma in a CSV file or the pipe character in a text file).

After the header row, the second and following rows in the input file contain the values for the configuration properties.

The header properties map to these PDR properties:

Input Header Property	Maps to PDR Property
`Key`	`mdex-property_Key`
`DisplayName`	`mdex-property_DisplayName`
`TextSearch`	`mdex-property_IsTextSearchable`
`SortOrder`	`system-navigation_Sorting`

The **Reformat** component will take these header properties and values and construct PDRs for the standard attributes.

Keep in mind that you can add additional properties to the input file so that they can be set. As mentioned, any PDR property that is not specified is added with its default value. See Chapter 1 of this guide for the system default values used by the MDEX Engine when creating standard attributes.

Note: Standard attribute names also cannot use hyphens in their names. Although the MDEX Engine will accept standard attribute names with hyphens, the Designer will not. Therefore, if you have a standard attribute name such as "Sales-Type", make sure you remove the hyphen from the name.

updateProperties operation

You can use the Configuration Service's `updateProperties` operation to load the PDR files into the MDEX Engine. The operation creates the standard attributes or updates them if they already exist. The PDR properties are listed in the topic "Default values for new attributes" in Chapter 2 of this guide.

The following is an example of an `updateProperties` operation:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"
  outerTransactionId="${MDEX_TRANSACTION_ID}">
    <config-service:updateProperties
      xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
        $xmlString
    </config-service:updateProperties>
</config-service:configTransaction>
```

This sample operation uses two variables:

- The `MDEX_TRANSACTION_ID` variable specifies the outer transaction ID for the request. The variable and its value is stored in the `workspace.prm` file of the LDI Designer project.
- The `$xmlString` variable contains the various PDRs that have been constructed by a `Reformat` component in the graph.

The operation would be specified in the request structure of a **WebServiceClient** connector, which will then send the request to the Configuration Service on the MDEX Engine.

Adding components to the standard attributes schema graph

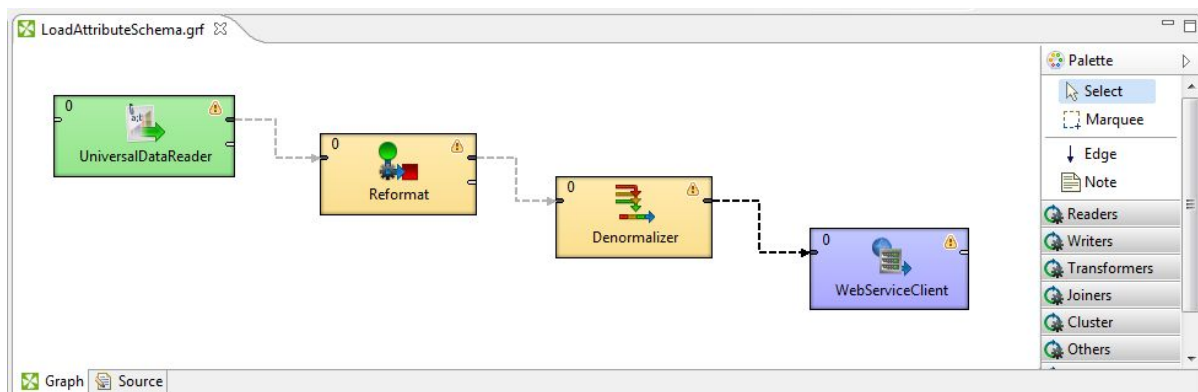
This topic describes the LDI components that must be added to the graph for your standard attributes schema.

This procedure assumes that you have created an empty graph (our example is named LoadAttributeSchema).

To add components to the graph that loads the standard attribute description files:

1. In the Palette pane, drag the following components into the Graph Editor:
   a) Drag the **UniversalDataReader** component from the **Readers** section.
   b) Drag the **Reformat** component from the **Transformers** section.
   c) Drag the **Denormalizer** component from the **Transformers** section.
   d) Drag the **WebServiceClient** component from the **Others** section.
2. In the Palette pane, click **Edge** and use it to connect the components.
3. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the connected components should look like this:



Configuring the Reader for the PDR input file

This task describes how to configure the **UniversalDataReader** component to read in the PDR source data.

This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the PDR source file to the project's **data-in** folder.

To configure the Reader component for the PDR input file:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the source data file and click **OK**.
3. Click **OK** to apply your configuration changes to the Reader component.
4. Save the graph.

Configuring the Reader Edge

The Edge for the Reader component must be configured with a Metadata definition.

This Metadata definition task will use the Metadata Editor. In the procedure, the column names will be extracted from the input file via a reparsing operation.

To configure the Metadata definition for the Reader Edge:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. In the Flat File dialog, browse for the PDR input file, select it, and click **OK**.
4. In the Flat File dialog, click **Next**.
   The Metadata Editor is displayed.
5. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.

   The PDR data is loaded into the Metadata Editor, with the properties named Field1, Field2, and so forth.
6. In the middle pane of the Metadata Editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.

   The correct property names are now displayed in the upper and middle panes of the Metadata Editor, which should look like this example:

#		Name	Type	Delimiter
1		Record: AttributeMetadata_csv	delimited	,
	1	Key	string	,
	2	DisplayName	string	,
	3	TextSearch	string	,
	4	SortOrder	string	\r\n

Filter:

7. In the upper pane of the Metadata Editor:
   a) Optionally, click the **Record** Name field and change the name of the metadata to a more descriptive name.
   b) Make sure that the **Type** field of all the properties is set to type **string**.
   c) Verify that the fields have the correct delimiter character set (which is the comma for our example).
8. When you have input all your changes, click **Finish**.
9. Save the graph.

Configuring the Reformat component for standard attributes

A **Reformat** component is used to transform incoming configuration data into a Standard Attribute Description Record.

The transformation is done by this CTL function in the **Reformat** component:

```
integer n = 1;
integer aggrKey = 0;

// Transforms input record into output record.
function integer transform() {
  string searchBool = "";
  string saRecord = "<mdex:record xmlns=\"\">";
  saRecord = saRecord + "<mdex-property_Key>" + $0.Key + "</mdex-proper¬
ty_Key>";
  saRecord = saRecord + "<mdex-property_DisplayName>" + $0.DisplayName +
"</mdex-property_DisplayName>";

  // Lower case the boolean in the CSV file
  searchBool = lowerCase($0.TextSearch);
  saRecord = saRecord + "<mdex-property_IsTextSearchable>" + searchBool +
"</mdex-property_IsTextSearchable>";

  saRecord = saRecord + "<system-navigation_Sorting>" + $0.SortOrder +
"</system-navigation_Sorting>";

  $0.xmlString = saRecord + "</mdex:record>";

  // Batch up the web service requests.
  $0.singleAggregationKey = aggrKey;
  n++;
  if (n % 15 == 0) {
    aggrKey++;
  }
```

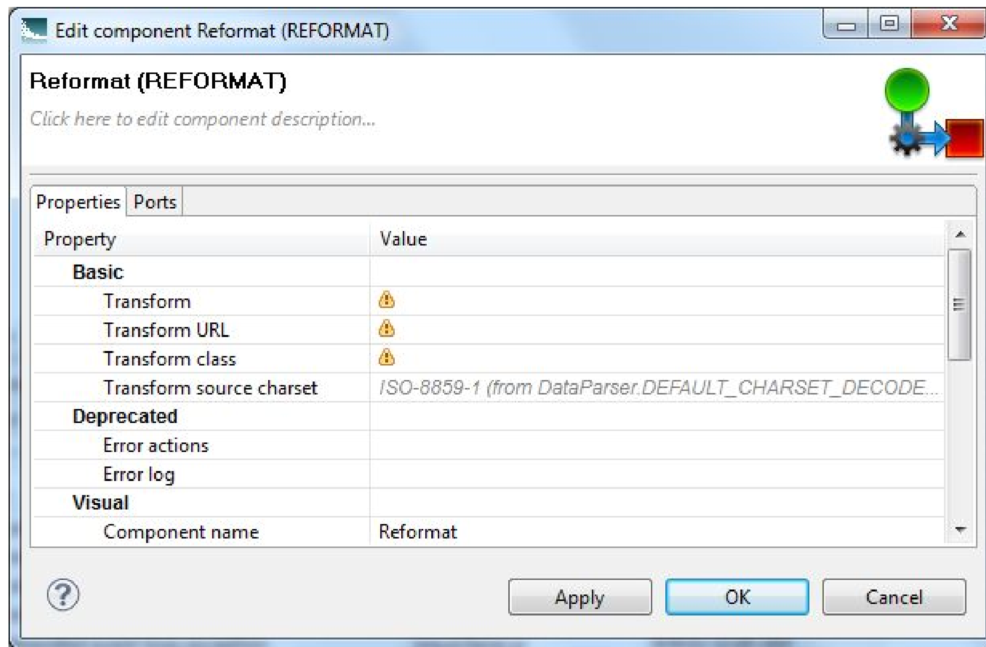
```
  return ALL;
}
```

The function builds each Standard Attribute Description Record (SADR) using the configuration data in the input CSV file. Keep in mind that, if you wish, you can add more SADR property definitions; if you do so, be sure to update the input file for the additional input values.

To configure the **Reformat** component in the standard attribute schema graph:

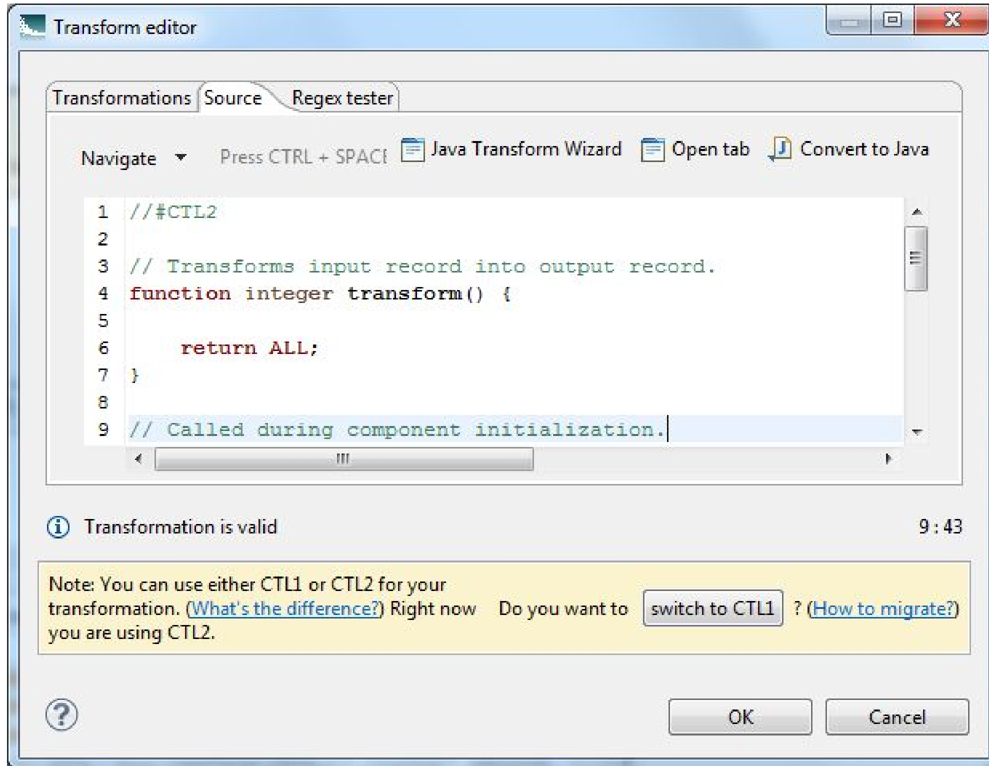
1. In the Graph window, double-click the **Reformat** component.

   The Reformat Edit Component dialog is displayed.



2. Single-click in the **Transform** field and then click the **...** button.
   The Transform editor is displayed.
3. Click the **Source** tab in the editor.

   The CTL template for the transform function is shown.



4. Modify the CTL script so that it looks like the CTL example above.
   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.
5. When you have finished your changes in the Transform editor, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
6. Optionally, you can change the **Component name** field to provide a customized name (such as "Transform Attribute Metadata") for this component.
7. Click **OK** to apply your configuration changes.
8. Save the graph.

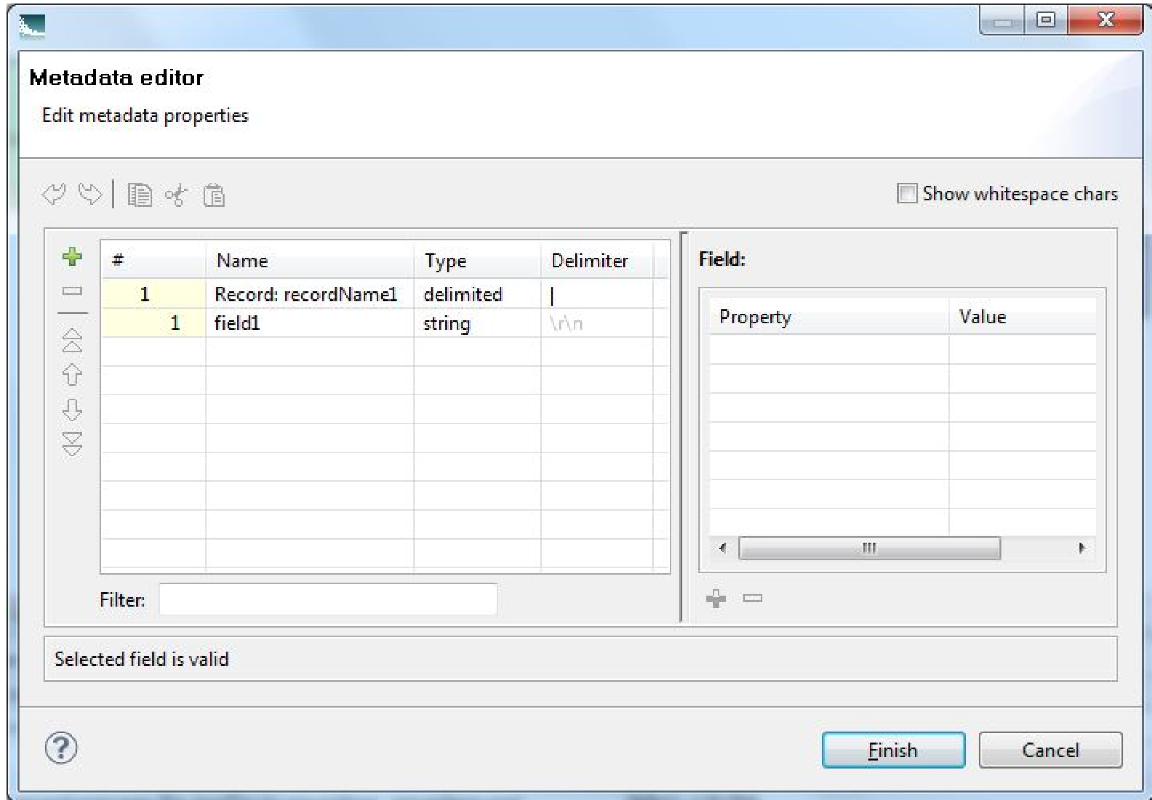
The two messages listed above should disappear once you configure the **Reformat** component Edge metadata.

Configuring the Reformat Edge

This task describes how to configure the Edge component that connects the **Reformat** and **Denormalizer** components.

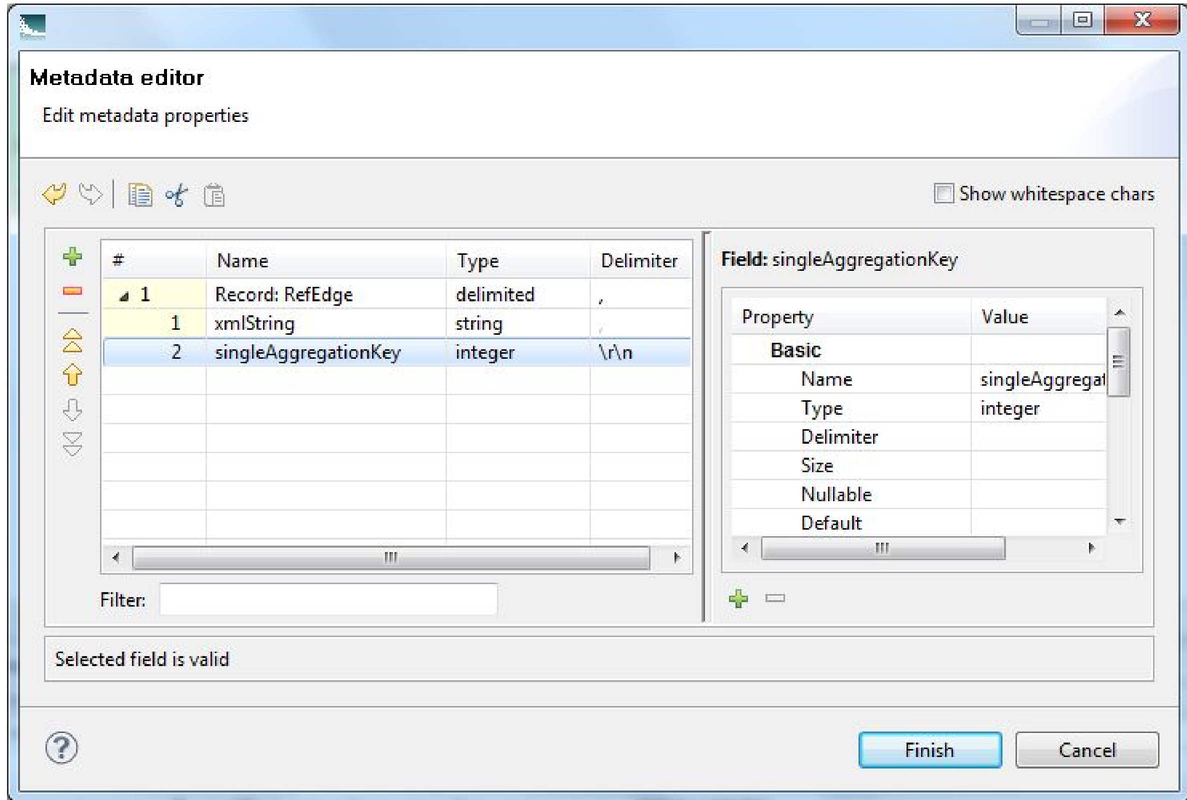
To configure the **Reformat** component's Edge in the attribute schema graph:

1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a name that is appropriate for your data.
   b) Leave the **Type** field as `delimited`.
   c) Set the **Delimiter** field to the delimiter character in your input file (which is the comma in our example).
3. For the other fields:
   a) Change the **field1** name to `xmlString` and leave its **Type** as `string`.
   b) Add a new field by using the + (plus sign control). Name the field `singleAggregationKey` and set its **Type** as `integer`.

   At this point, the Metadata editor should look like this:



4. When you have input all your changes in the Metadata editor, click **Finish**.
5. Save the graph.

Configuring the Denormalizer component

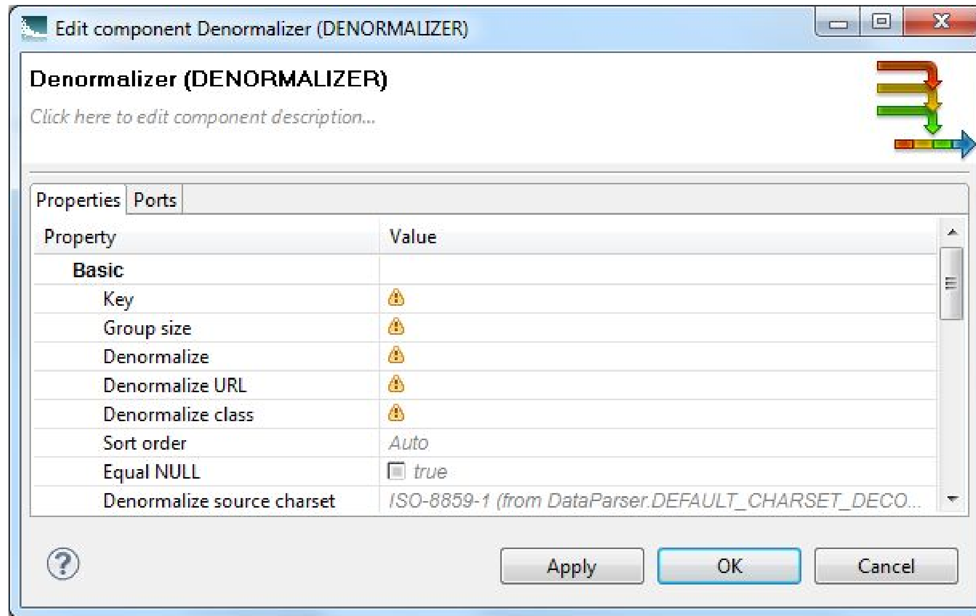
A **Denormalizer** component is used to create a single output record for a group of input records defined by the key.

The transformation is done by these CTL functions in the **Denormalizer** component:

```
integer n = 0;
string value = "";

function integer append() {
   value = value + $0.xmlString + "\n";
   n++;
   return n;
}

// This function is called once after the
// append() function was called for all records
// of a group of input records defined by the key.
// It creates a single output record for the whole group.
function integer transform() {
   $0.xmlString = value;
   value = "";
   return OK;
}
```

To configure the **Denormalizer** component in the attribute schema graph:

1. In the Graph window, double-click the **Denormalizer** component.
   The Denormalizer Edit Component dialog is displayed.



2. Single-click in the **Key** field and then click the **...** button.
   The Edit Key dialog is displayed.
3. In the **Fields** pane of the Edit Key dialog, select **singleAggregationKey** and move it to the **Key parts** pane by clicking the right-arrow button. Click **OK** to apply your change.
4. Single-click in the **Denormalize** field and then click the **...** button.
   The Transform editor is displayed.
5. In the **Source** tab of the editor, modify the CTL script so that it looks like the example above.
   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.
6. When you have finished your edits, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
7. Optionally, you can use the **Component name** field to provide a customized name (such as "Load Schema") for this component.
8. Click **OK** to apply your configuration changes.
9. Save the graph.

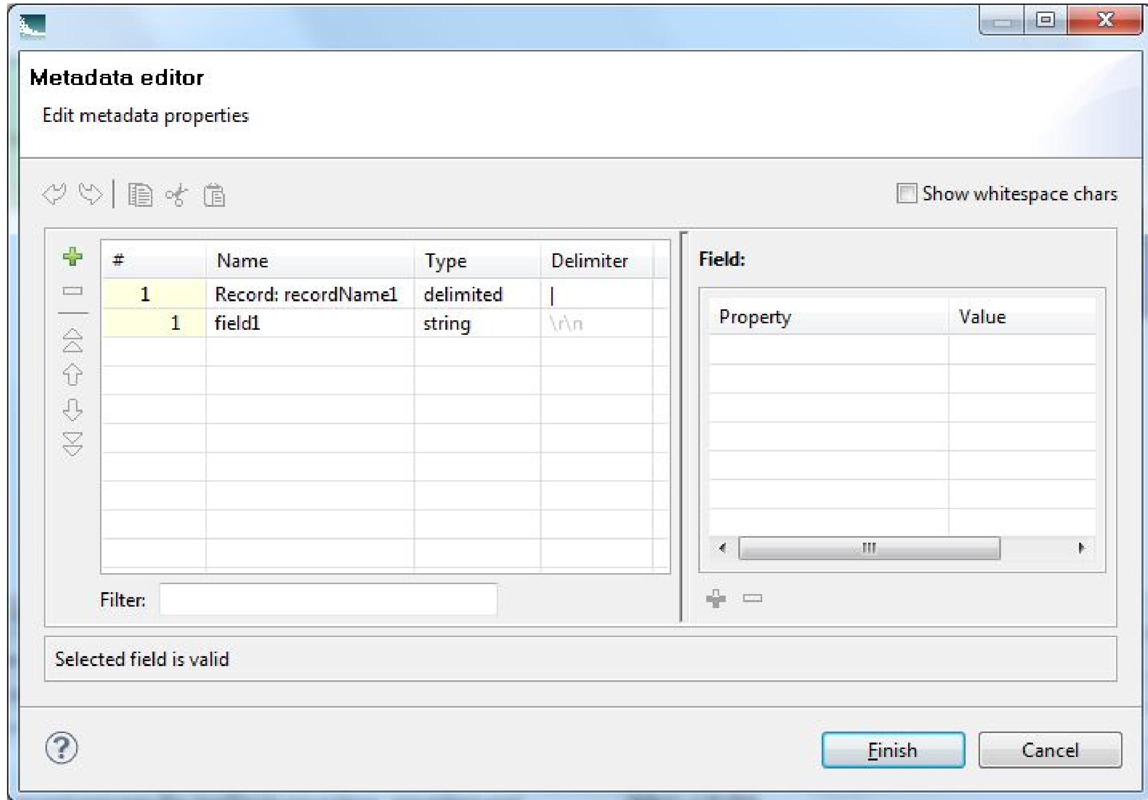
The two messages listed above should disappear once you configure the **Denormalizer** component Edge metadata.

Configuring the Denormalizer Edge

This task describes how to configure the Edge component that connects the **Denormalizer** and **WebServiceClient** components.

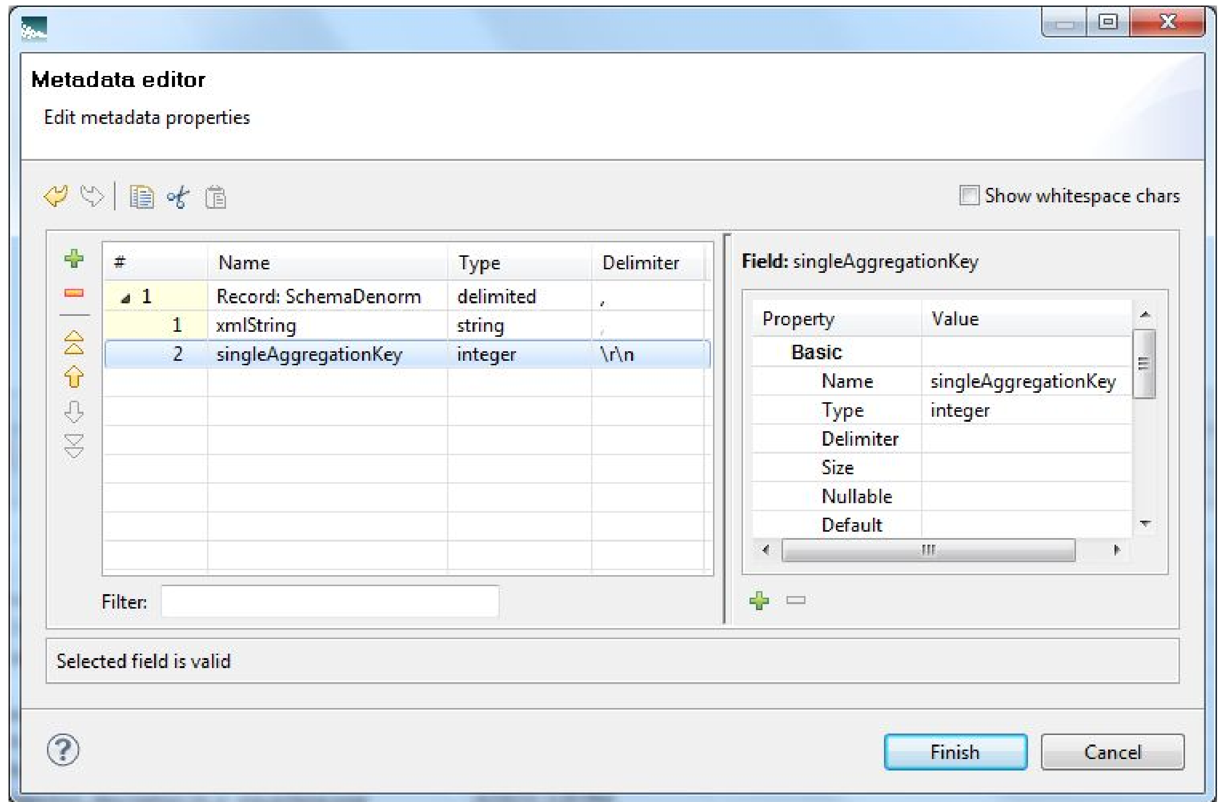
To configure the **Denormalizer** component's Edge in the attribute schema graph:

1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a name that is appropriate for your data.
   b) Leave the **Type** field as `delimited`.
   c) Set the **Delimiter** field to the delimiter character in your input file (which is the comma in our example).
3. For the other fields:
   a) Change the **field1** name to `xmlString` and leave its **Type** as `string`.
   b) Add a new field by using the + (plus sign control). Name the field `singleAggregationKey` and set its **Type** as `integer`.

   At this point, the Metadata editor should look like this:



4. When you have input all your changes in the Metadata editor, click **Finish**.
5. Save the graph.

Configuring the WebServiceClient component for standard attributes

This topic describes how to configure the **WebServiceClient** connector for loading standard attribute metadata.

This procedure assumes that you have created a graph and added the **WebServiceClient** component.

The procedure also assumes that you are specifying an outer transaction ID with the request and that your `workspace.prm` file has defined the ID in the `MDEX_TRANSACTION_ID` variable, as in this example:

```
MDEX_TRANSACTION_ID=
```

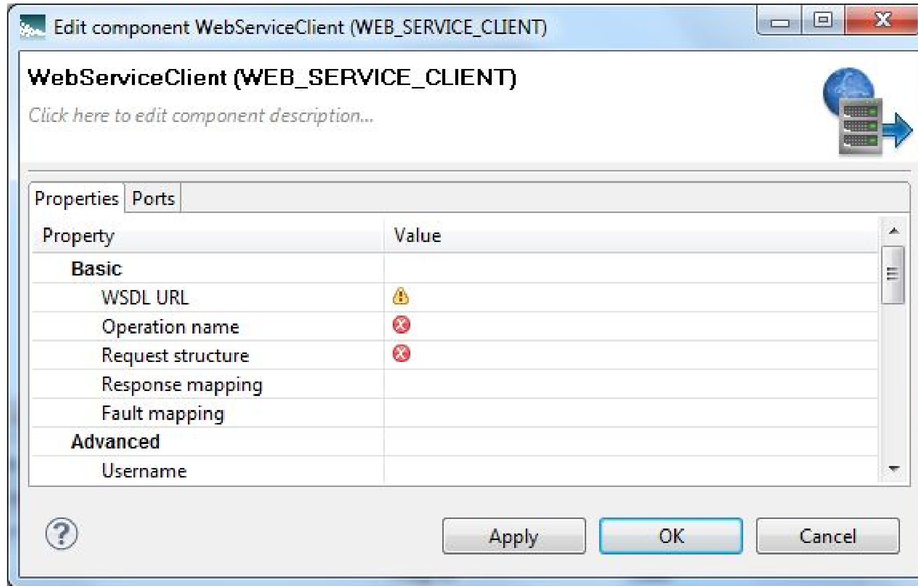
To configure the **WebServiceClient** connector for standard attribute metadata:

1. Make sure that the MDEX Engine is running and the Configuration Web service is available by issuing this URL command from your browser (be sure to use the correct port number for your MDEX Engine):

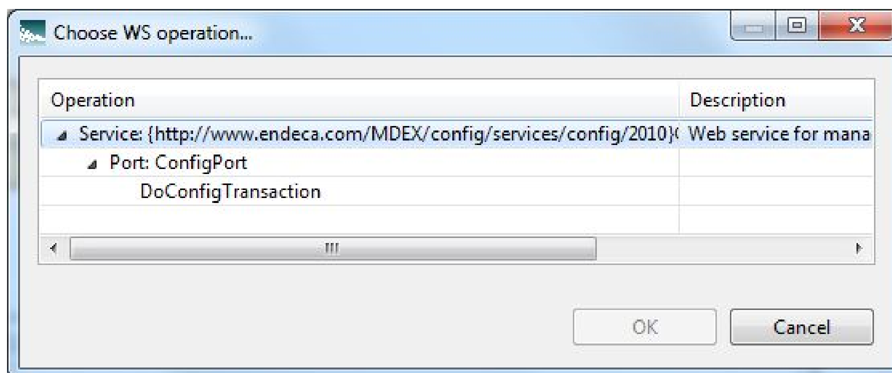
   ```
   http://localhost:5555/ws/config?wsdl
   ```

   The URL command returns the WSDL of the Web service.
2. In the Graph window, double-click the **WebServiceClient** component.

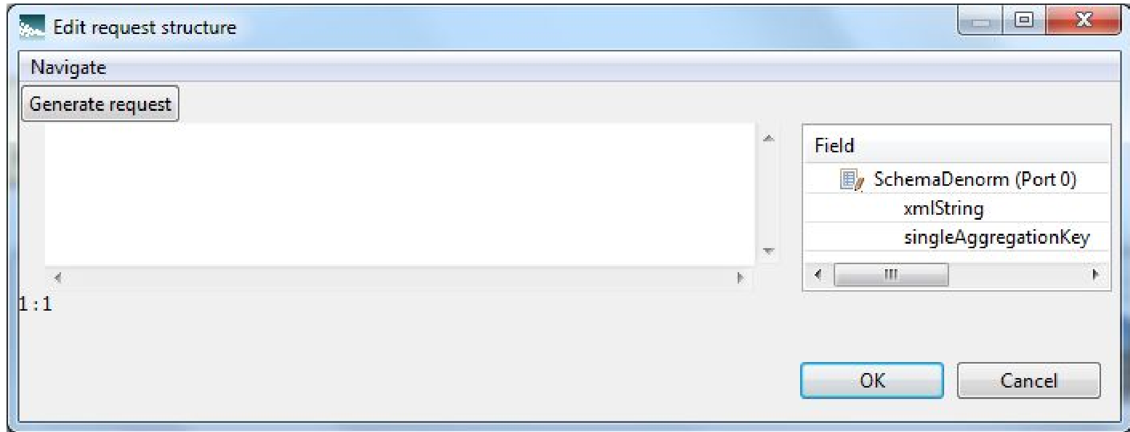
   The Writer Edit Component dialog is displayed.



3. In the **WSDL URL** field, enter the same URL as in Step 1.
4. In the **Operation name** field, click the **...** browse button, which displays the **Choose WS operation** dialog:



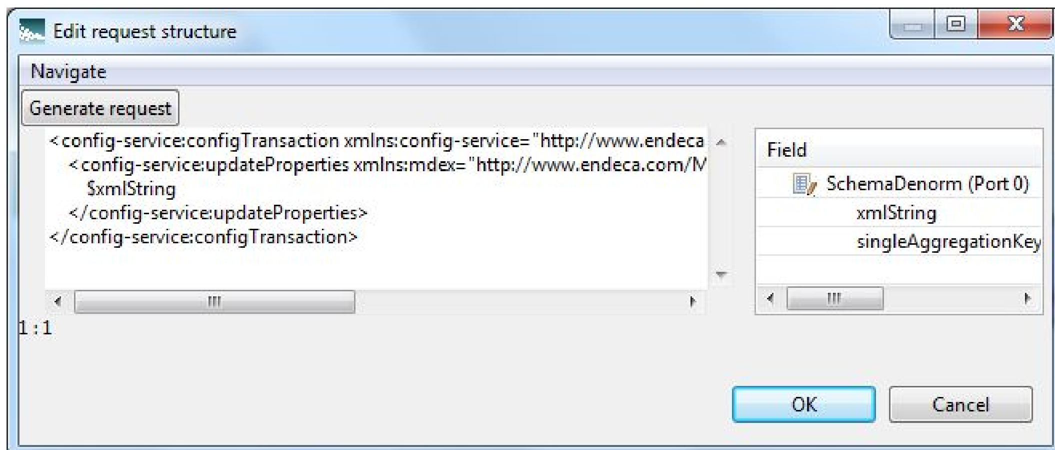
5. In the **Choose WS operation** dialog, select **DoConfigTransaction** and then click **OK**.
   The name of the Web service operation is entered in the **Operation name** field.
6. Click inside the **Request structure** field, which causes the **...** browse button to be displayed. Then click the browse button to display the **Edit request structure** dialog:



7. Add this text to the **Generate request** field:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
  <config-service:updateProperties
    xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
      $xmlString
  </config-service:updateProperties>
</config-service:configTransaction>
```

At this point, the **Edit request structure** dialog should look like this example:



8. After adding the request text in the **Edit request structure** dialog, click **OK**.
9. Optionally, you can use the **Component name** field to provide a customized name for this component.
10. When you have input all your changes, click **OK**.
11. Save the project.

Instead of running this graph directly, it is recommended that you create a transaction graph (with a **Transaction RunGraph** connector) with this LoadAttributeSchema graph as its child graph, and then run the transaction graph.

Loading the managed attribute schema

This topic provides an overview of the DDR load process.

From a high-level view, the steps you take to load your DDR schema into the MDEX Engine are as follows:

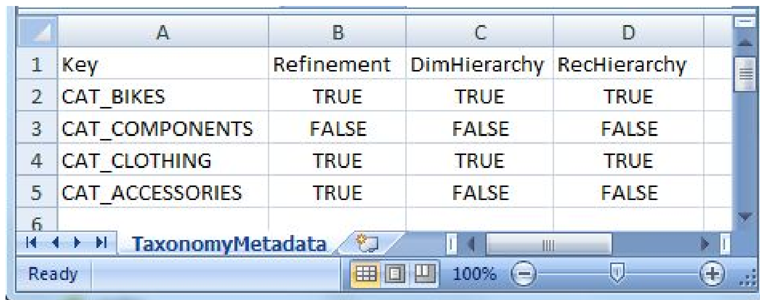
1. Create the managed attributes input file.
2. Either create a new project or re-use an existing one.
3. Create a graph and add the **UniversalDataReader**, **Reformat**, **Denormalizer**, and **WebServiceClient** components.
4. Configure the components.
5. Run this graph as part of a transaction graph.

Keep in mind that before loading DDR records, you should first load the PDR records that are associated with the DDRs (unless the appropriate PDRs have already been loaded into the MDEX Engine).

Format of the DDR input file

The DDR input file defines the Endeca managed attributes, with the specific settings of some DDR properties.

The `TaxonomyMetadata.csv` sample input file used for the managed attributes looks like this:



	A	B	C	D
1	Key	Refinement	DimHierarchy	RecHierarchy
2	CAT_BIKES	TRUE	TRUE	TRUE
3	CAT_COMPONENTS	FALSE	FALSE	FALSE
4	CAT_CLOTHING	TRUE	TRUE	TRUE
5	CAT_ACCESSORIES	TRUE	FALSE	FALSE
6				

The first line (the header row) of the sample file has these header properties:

```
Key,Refinement,DimSearch,RecHierarchy
```

The actual names of the header properties can be different from the names used here. The properties are delimited (for example, by the comma in the sample CSV file). After the header row, the second and following rows in the input file contain the values for the configuration properties.

The header properties map to these DDR properties:

Input Header Property	Maps to PDR Property
`Key`	`mdex-dimension_Key`
`Refinement`	`mdex-dimension_EnableRefinements`
`DimHierarchy`	`mdex-dimension_IsDimensionSearchHierarchical`
`RecHierarchy`	`mdex-dimension_IsRecordSearchHierarchical`

The **Reformat** component will take these header properties and values and construct DDRs for the managed attributes.

updateDimensions operation

The Configuration Service's `updateDimensions` operation can load the DDR files into the MDEX Engine. The operation creates the standard attributes or updates them if they already exist. The DDR properties are listed in the topic "Default values for new attributes" in Chapter 2 of this guide.

The following is an example of an `updateDimensions` operation:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"
  outerTransactionId="${MDEX_TRANSACTION_ID}">
    <config-service:updateDimensions
      xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
        $xmlString
    </config-service:updateDimensions>
</config-service:configTransaction>
```

This sample operation uses two variables:

- The `MDEX_TRANSACTION_ID` variable specifies the outer transaction ID for the request. The variable and its value is stored in the `workspace.prm` file of the LDI Designer project.
- The `$xmlString` variable contains the various DDRs that have been constructed by a `Reformat` component in the graph.

The operation would be specified in the request structure of a **WebServiceClient** connector, which will then send the request to the Configuration Service on the MDEX Engine.

Adding components to the managed attributes schema graph

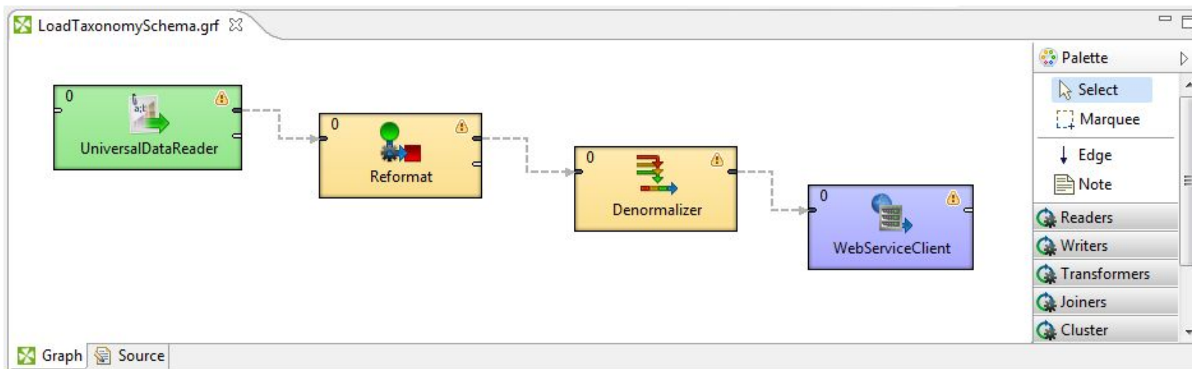
This topic describes the LDI components that must be added to the graph for your managed attributes schema.

This procedure assumes that you have created an empty graph for your managed attributes schema. This graph will use the same components as the graph for the standard attributes schema.

To add components to the graph that loads the managed attribute description files:

1. In the Palette pane, drag the following components into the Graph Editor:
    a) **UniversalDataReader** component
    b) **Reformat** component
    c) **Denormalizer** component
    d) **WebServiceClient** component
2. In the Palette pane, click **Edge** and use it to connect the components.
3. Save the graph.

The resulting graph should like this:



Configuring the Reader and the Edge for DDRs

These two configurations are very similar to those for PDR loads.

This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added an Edge and also added the DDR source file to the project's **data-in** folder.

To configure the **UniversalDataReader** and Edge components:

1. To configure the **UniversalDataReader** component for the DDR input file, use the same procedure as described in the topic titled "Configuring the Reader for the PDR input file" in this chapter.

   The only difference is that you will be using your DDR file as the input file.
2. To configure the Edge component, use the same procedure as described in the topic titled "Configuring PDR metadata" in this chapter.

   Be sure to use the **Extract names** and **Reparse** options on the Metadata Editor.
3. When you have finished your configuration, save the graph.

Configuring the Reformat component for managed attributes

A **Reformat** component is used to transform incoming configuration data into a Managed Attribute Description Record.

The transformation is done by this CTL function in the **Reformat** component:

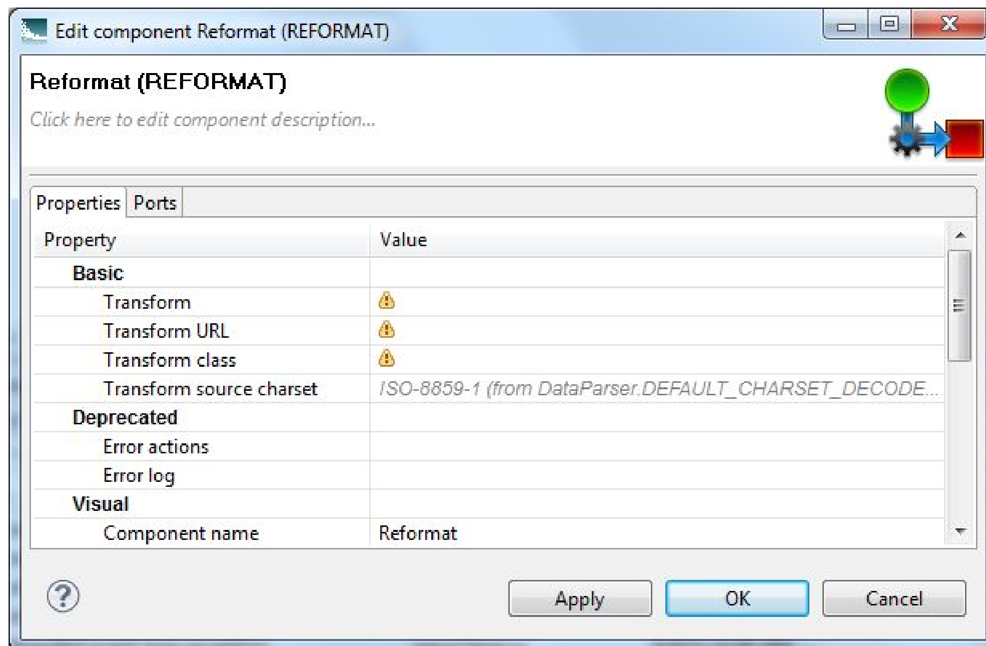
```
//#CTL2

integer n = 1;
integer aggrKey = 0;

// Transforms input record into output record.
function integer transform() {
   string maBool = "";
   string maRecord = "<mdex:record xmlns=\"\">";
   maRecord = maRecord + "<mdex-dimension_Key>" + $0.Key + "</mdex-dimen¬
sion_Key>";

   // Make sure to lower case the booleans in the CSV file
   maBool = lowerCase($0.Refinement);
   maRecord = maRecord + "<mdex-dimension_EnableRefinements>" + maBool +
"</mdex-dimension_EnableRefinements>";
```

```
   maBool = lowerCase($0.DimHierarchy);
   maRecord = maRecord + "<mdex-dimension_IsDimensionSearchHierarchical>"
+ maBool + "</mdex-dimension_IsDimensionSearchHierarchical>";

   maBool = lowerCase($0.RecHierarchy);
   maRecord = maRecord + "<mdex-dimension_IsRecordSearchHierarchical>" +
maBool + "</mdex-dimension_IsRecordSearchHierarchical>";

   $0.xmlString = maRecord + "</mdex:record>";

   // Batch up the web service requests.
   $0.singleAggregationKey = aggrKey;
   n++;
   if (n % 15 == 0) {
      aggrKey++;
   }

   return ALL;
}
```

The function builds each Managed Attribute Description Record using the configuration data in the input CSV file.

To configure the **Reformat** component in the managed attribute schema graph:

1. In the Graph window, double-click the **Reformat** component.
   The Reformat Edit Component dialog is displayed.



2. Single-click in the **Transform** field and then click the **...** button.
   The Transform editor is displayed.
3. Click the **Source** tab in the editor.
   The CTL template for the transform function is displayed.
4. Modify the CTL script so that it looks like the example above.

You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.

5. When you have finished your changes in the Transform editor, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
6. Optionally, you can change the **Component name** field to provide a customized name (such as "Transform Attribute Metadata") for this component.
7. Click **OK** to apply your configuration changes.
8. Save the graph.

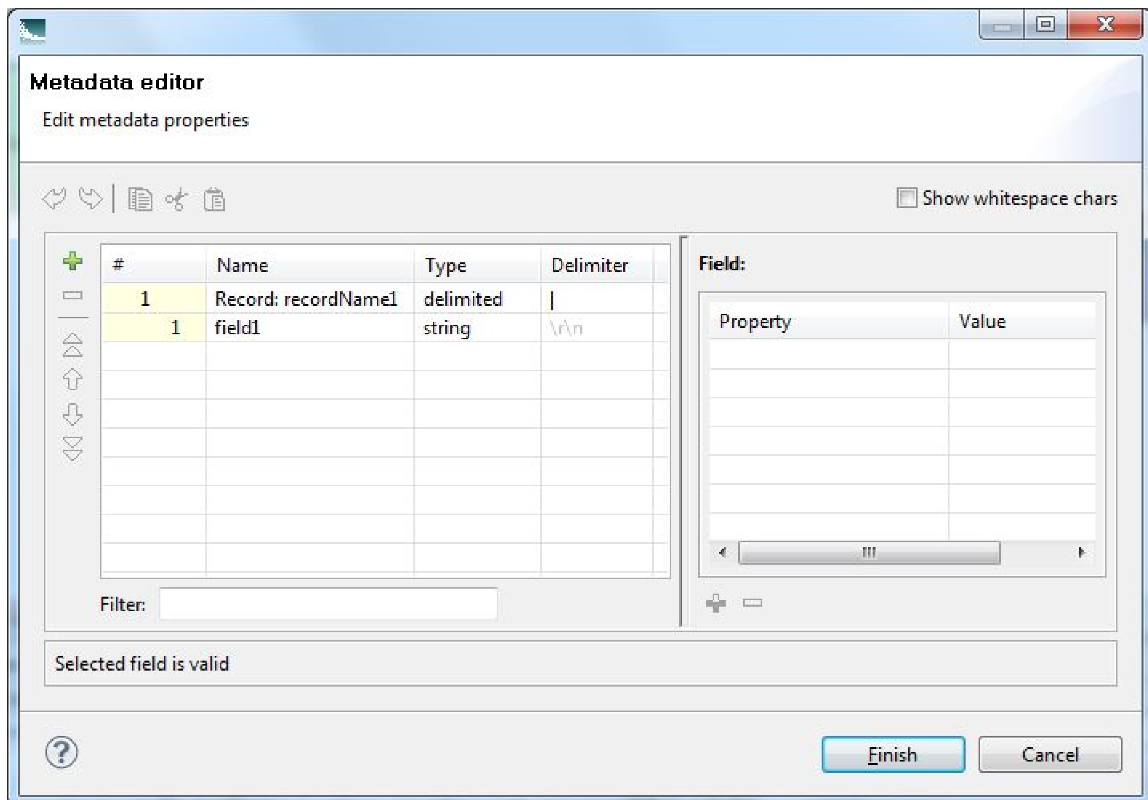
The two messages listed above should disappear once you configure the **Reformat** component Edge metadata.

Configuring the Reformat Edge

This task describes how to configure the Edge component that connects the **Reformat** and **Denormalizer** components.

To configure the **Reformat** component's Edge in the attribute schema graph:

1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a name that is appropriate for your data.
   b) Leave the **Type** field as `delimited`.

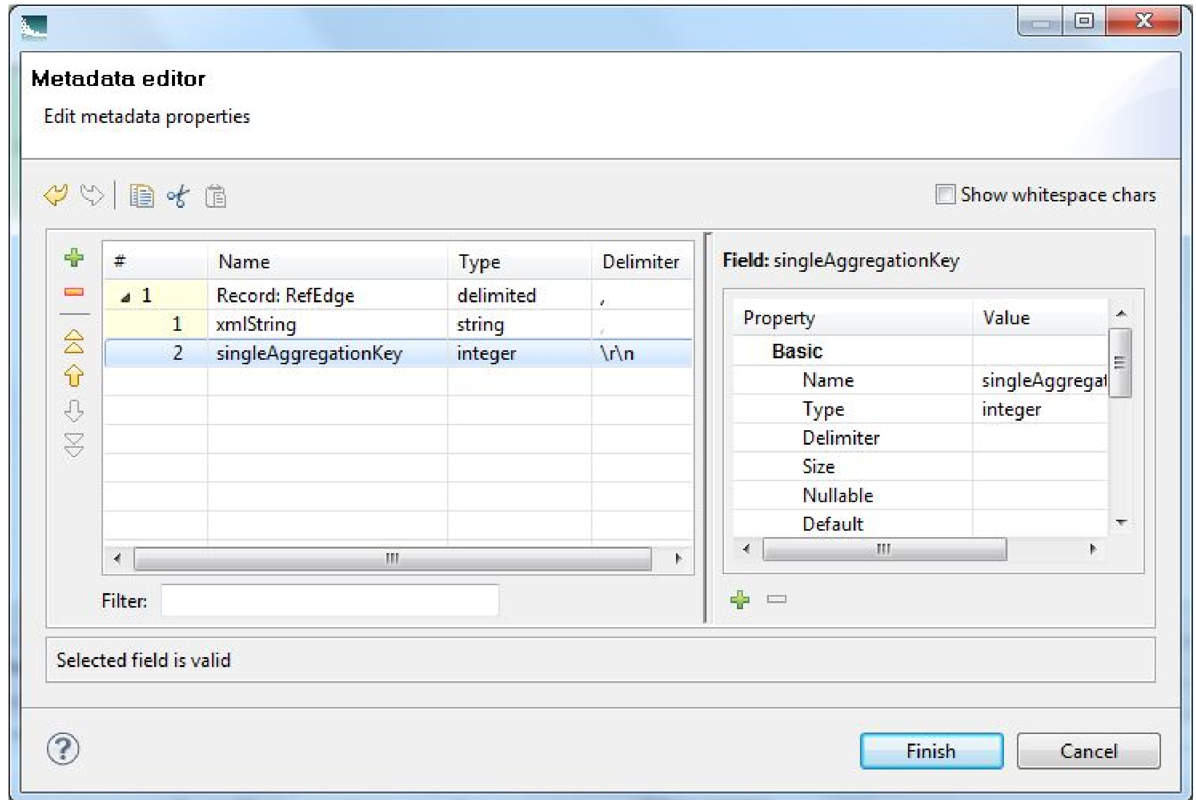
c) Set the **Delimiter** field to the delimiter character in your input file (which is the comma in our example).

3. For the other fields:

a) Change the **field1** name to `xmlString` and leave its **Type** as `string`.

b) Add a new field by using the + (plus sign control). Name the field `singleAggregationKey` and set its **Type** as `integer`.

At this point, the Metadata editor should look like this:



4. When you have input all your changes in the Metadata editor, click **Finish**.
5. Save the graph.

Configuring the Denormalizer and the Edge for DDRs

These two configurations are very similar to those for PDR loads.

To configure the **Denormalizer** and Edge components:

1. To configure the **Denormalizer** component for managed attributes, use the same procedure as described in the topic titled "Configuring the Denormalizer component" in this chapter.
2. To configure the Edge component, use the same procedure as described in the topic titled "Configuring the Denormalizer Edge" in this chapter.
3. When you have finished your configuration, save the graph.

Configuring the WebServiceClient component for managed attributes

This topic describes how to configure the **WebServiceClient** component for loading managed attribute metadata.

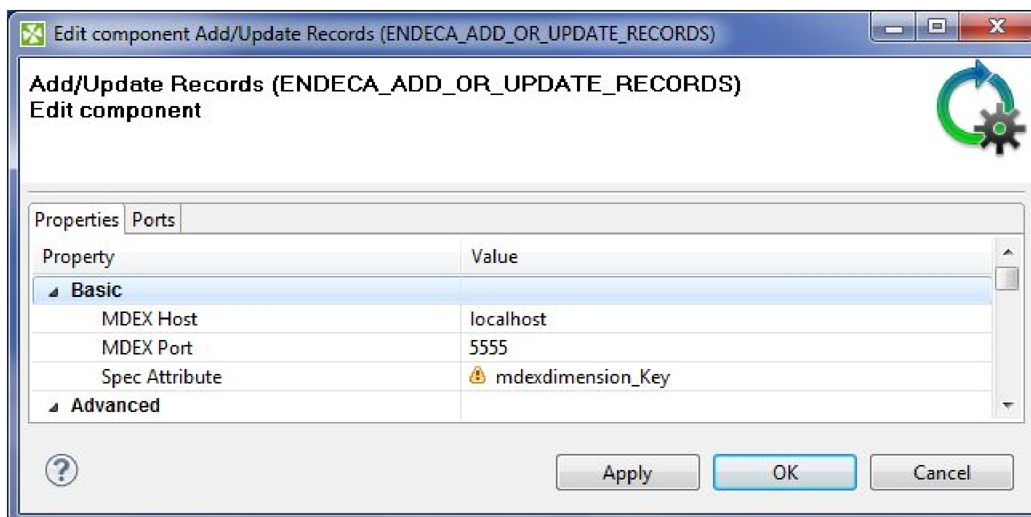
This procedure assumes that you have created a graph and added the **WebServiceClient** component.

To configure the **WebServiceClient** connector for managed attribute metadata:

1. With one exception, the configuration procedure is the same as that described in the topic titled "Configuring the WebServiceClient component for standard attributes" in this chapter. Therefore, first follow Steps 1-6 in that topic.
2. Replace Step 7 in that topic by adding this text to the **Generate request** field in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
  <config-service:updateDimensions
    xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
      $xmlString
  </config-service:updateDimensions>
</config-service:configTransaction>
```

At this point, the Basic section of the dialog should look like this example:



3. Continue with Steps 8-11 of the topic.

Instead of running this graph directly, it is recommended that you create a transaction graph (with a **Transaction RunGraph** connector) with this LoadTaxonomySchema graph as its child graph, and then run the transaction graph.

Using a transaction graph to load the schemas

You should run the two schema graphs with a transaction graph.

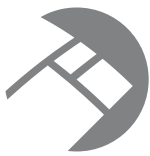
A transaction graph uses a **Transaction RunGraph** connector to safely run one or more graphs within the transaction environment of the MDEX Engine. This connector can start an outer transaction, run

the set of graphs so that they succeed or fail as a unit, and finally commit the transaction (or roll it back upon failure).

To use a transaction graph to load the two attribute schemas:

1. Create a transaction graph as described in the chapter titled "Working with Transaction Graphs".
   Note that the chapter uses the two attribute schema graphs as examples.
2. Run the transaction graph as you would run any other graph.

The transaction graph will first run the standard attribute schema graph and then the managed attribute schema graph.



Chapter 7

Loading Configuration Files

This chapter describes how to load the Global Configuration Record and the index configuration documents for the MDEX Engine.

Types of MDEX Engine configuration documents

The MDEX Engine offers a rich set of index configuration documents that allow you to customize your Endeca implementation.

The index configuration is the mechanism for implementing a number of Endeca features such as search and ranking. The index configuration documents are created automatically by the `mkmdex` utility with a set of defaults that are described in the following topics. The index configuration documents are stored in the MDEX indexes database and loaded into the MDEX Engine at startup.

The documents are as follows:

Index Configuration Document	Purpose
`dimsearch_config`	Configures attributes (both Standard Attributes and Managed Attributes) for value search.
`recsearch_config`	Configures record search, including search interfaces which control record search behavior for groups of attributes. Some of the features that can be specified for a search interface include relevance ranking, matching across multiple attributes, and partial matching.
`relrank_strategies`	Sets relevance ranking, which is used to control the order of results that are returned in response to a record search.
`stop_words`	Sets stop words, which are words that are set to be ignored by the MDEX Engine.
`thesaurus`	The thesaurus allows the system to return matches for related concepts to words or phrases contained in user queries.

Recommended order for loading the configuration documents

The recommended order of loading is:

1. Load the attribute schema (PDRs and DDRs) first. It does not matter if the actual data records are loaded, but the PDRs and DDRs are important because they create the properties that should be referenced by the configuration files.
2. `relrank_strategies` document (necessary if a relevance ranking strategy is referenced by the next two documents)
3. `recsearch_config` document
4. `dimsearch_config` document
5. `stop_words` document
6. `thesaurus` document

Global Configuration Record

The Global Configuration Record (GCR) stores global configuration settings for the MDEX Engine.

The GCR sets the configuration for wildcard search enablement, search characters, merge policy, and spelling correction settings. A full description of its properties and their default values is available in the *Latitude Developer's Guide*.

When loading your changes for the GCR, keep these requirements in mind:

- The `mdex-config_Key` property must be unique and single-assign. The value must be `global` for the property.
- The GCR must contain valid values for all of its properties. None of its properties can be omitted.
- The GCR cannot have any arbitrary, user-defined properties.

If you change any of the spelling settings, make sure you rebuild the aspell dictionary by running the `admin?op=updateaspell` administrative operation.

Sample GCR input file

The following is a sample GCR:

```
<mdex:record>
  <mdex-config_Key>global</mdex-config_Key>
  <mdex-config_EnableValueSearchWildcard>true</mdex-config_EnableValueSearch¬
Wildcard>
  <mdex-config_MergePolicy>aggressive</mdex-config_MergePolicy>
  <mdex-config_SearchChars>+_</mdex-config_SearchChars>
  <mdex-config_SpellingRecordMinWordOccur>2</mdex-config_SpellingRecordMin¬
WordOccur>
  <mdex-config_SpellingRecordMinWordLength>4</mdex-config_SpellingRecordMin¬
WordLength>
  <mdex-config_SpellingRecordMaxWordLength>24</mdex-config_SpellingRecord¬
MaxWordLength>
  <mdex-config_SpellingDValMinWordOccur>5</mdex-config_SpellingDValMinWor¬
dOccur>
  <mdex-config_SpellingDValMinWordLength>3</mdex-config_SpellingDValMin¬
WordLength>
  <mdex-config_SpellingDValMaxWordLength>20</mdex-config_SpellingDValMax¬
WordLength>
</mdex:record>
```

This GCR:

- Enables wildcard search by setting the `mdex-config_EnableValueSearchWildcard` property to `true`.
- Sets the merge policy to aggressive via the `mdex-config_MergePolicy` property.

- Adds the plus (+) and underscore (_) characters as search characters for value search and record search operations.

You can create the file in a text editor.

dimsearch_config document

This document sets the configuration for value search.

The default `dimsearch_config` document contains an empty configuration:

```
<DIMSEARCH_CONFIG/>
```

In the configuration document, you can use the `RELRANK_STRATEGY` attribute to specify a relevance ranking strategy to use on the results. If you do so, you must first use the `relrank_strategies` document to configure the relevance ranking strategy in the MDEX Engine.

Sample dimsearch_config document

To configure value search, you need to create an input file similar to this example:

```
<DIMSEARCH_CONFIG FILTER_FOR_ANCESTORS="FALSE" RELRANK_STRATEGY="ProductRel¬
Rank"/>
```

As mentioned above, the ProductRelRank strategy must have been configured previously with the `relrank_strategies` document.

Request structure text

When you configure the **WebServiceClient** component, you add the following request text in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="dimsearch_config">
<DIMSEARCH_CONFIG>
  $xmlString
</DIMSEARCH_CONFIG>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

The `name="dimsearch_config"` attribute references the `dimsearch_config` document.

Run-time error

If the `RELRANK_STRATEGY` attribute in the document references a non-existent relevance ranking strategy, the load operation will fail with an error similar to this example:

```
ERROR [WatchDog] - Graph execution finished with error
ERROR [WatchDog] - Node WEB_SERVICE_CLIENT3 finished with
status: ERROR caused by: Error applying updates:
Invalid Relevance ranking strategy "ProductRelRank" in DIMSEARCH_CONFIG
element.
ERROR [WatchDog] - Node WEB_SERVICE_CLIENT3 error details:
org.apache.axis2.AxisFault: Error applying updates: Invalid Relevance
```

```
ranking strategy
"ProductRelRank" in DIMSEARCH_CONFIG element.
```

To correct this error, first use the `relrank_strategies` document to create the relevance ranking strategy in the MDEX Engine before you attempt to load your `dimsearch_config` document.

recsearch_config document

This document configures record search, including search interfaces which control record search behavior for groups of attributes.

Some of the features that can be specified for a search interface include relevance ranking, matching across multiple Endeca attributes, partial matching, and enabling snippeting for one or more Endeca attributes.

The default `recsearch_config` document contains an empty configuration:

```
<RECSEARCH_CONFIG/>
```

In the configuration document, you can use the `RELRANK_STRATEGY` attribute to specify a relevance ranking strategy to use on the results. If you do so, you must first use the `relrank_strategies` document to configure the relevance ranking strategy in the MDEX Engine.

Sample recsearch_config document

The following example shows a `recsearch_config` document with three search interfaces:

```
<RECSEARCH_CONFIG>
 <SEARCH_INTERFACE NAME="Surveys">
    <MEMBER_NAME RELEVANCE_RANK="1">SurveyResponse</MEMBER_NAME>
 </SEARCH_INTERFACE>
 <SEARCH_INTERFACE NAME="Resellers">
    <MEMBER_NAME RELEVANCE_RANK="1">DimReseller_BusinessType</MEMBER_NAME>

    <MEMBER_NAME RELEVANCE_RANK="2">DimReseller_ResellerName</MEMBER_NAME>

 </SEARCH_INTERFACE>
 <SEARCH_INTERFACE NAME="Employees">
    <MEMBER_NAME RELEVANCE_RANK="1">DimEmployee_FullName</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="2">DimEmployee_LastName</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="3">DimEmployee_FirstName</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="4">DimEmployee_Title</MEMBER_NAME>
 </SEARCH_INTERFACE>
</RECSEARCH_CONFIG>
```

The example creates the Surveys, Resellers, and Employees search interfaces. All the configured standard attributes (such as DimEmployee_FullName) must already exist in the MDEX Engine.

Note that if you include a relevance ranking strategy, it must have been configured previously with the `relrank_strategies` document.

Request structure text

When you configure the **WebServiceClient** component, you add the following request text in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
```

```
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="recsearch_config">
<RECSEARCH_CONFIG>
  $xmlString
</RECSEARCH_CONFIG>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

The `$xmlString` variable contains the XML definition of the search interfaces and the `name="recsearch_config"` attribute references the `recsearch_config` document.

Run-time errors

If the `RELRANK_STRATEGY` attribute in the document references a non-existent relevance ranking strategy, the load operation will fail with an error similar to this example:

```
ERROR [WatchDog] - Graph execution finished with error
ERROR [WatchDog] - Node WEB_SERVICE_CLIENT0 finished with
status: ERROR caused by: Error applying updates:
Invalid Relevance Ranking Strategy "ProductRelRank" referenced
in SEARCH_INTERFACE "ProductSearch"
ERROR [WatchDog] - Node WEB_SERVICE_CLIENT0 error details:
org.apache.axis2.AxisFault: Error applying updates: Invalid
Relevance Ranking Strategy "ProductRelRank" referenced
in SEARCH_INTERFACE "ProductSearch"
```

To correct this error, first use the `relrank_strategies` document to create the relevance ranking strategy (named ProductRelRank in this example) in the MDEX Engine before you attempt to load your `recsearch_config` document.

In addition, the Endeca attributes referenced in the search interface must also exist in the MDEX Engine. Otherwise, the load operation will fail with an error similar to this example:

```
Error applying updates: No property with the name "ProductType" exists for
 search interface "ProductSearch"
```

To correct this error, first load your standard attribute schema before loading the configuration documents.

relrank_strategies document

This document configures the relevance ranking strategies for a Latitude application.

Relevance ranking is used to control the order of results that are returned in response to a record search. An individual relevance ranking strategy is expressed in a RELRANK_STRATEGY element, which in turn is made of individual relevance ranking modules such as RELRANK_EXACT, RELRANK_FIELD, and so on.

The default `relrank_strategies` document does not define any relevance ranking strategies:

```
<RELRANK_STRATEGIES/>
```

Sample relrank_strategies document

This example creates a relevance ranking strategy named ProductRelRank that consists of the RELRANK_INTERP and RELRANK_FIELD relevance ranking modules.

```
<RELRANK_STRATEGIES>
  <RELRANK_STRATEGY NAME="ProductRelRank">
```

```
    <RELRANK_INTERP/>
    <RELRANK_FIELD/>
  </RELRANK_STRATEGY>
</RELRANK_STRATEGIES>
```

Request structure text

When you configure the **WebServiceClient** component, you add the following request text in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="relrank_strategies">
<RELRANK_STRATEGIES>
  $xmlString
</RELRANK_STRATEGIES>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

The `name="relrank_strategies"` attribute references the `relrank_strategies` document.

stop_words document

This document sets the stop words for queries.

Stop words are words that should be eliminated from a query before it is processed by the MDEX Engine.

The default `stop_words` document does not define any stop words:

```
<STOP_WORDS/>
```

Sample stop_words document

This example sets the stop words for an application.

```
<STOP_WORDS>
  <STOP_WORD>bike</STOP_WORD>
  <STOP_WORD>component</STOP_WORD>
  <STOP_WORD>an</STOP_WORD>
  <STOP_WORD>of</STOP_WORD>
  <STOP_WORD>the</STOP_WORD>
</STOP_WORDS>
```

Request structure text

When you configure the **WebServiceClient** component, you add the following request text in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="stop_words">
```

```
<STOP_WORDS>
  $xmlString
</STOP_WORDS>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

The `name="stop_words"` attribute references the `stop_words` document.

thesaurus document

This document configures the thesaurus for your application.

The thesaurus allows the system to return matches for related concepts to words or phrases contained in user queries.

The default `thesaurus` document does not define any stop words:

```
<THESAURUS/>
```

Sample thesaurus document

This example sets two thesaurus entries:

```
<THESAURUS>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>italy</THESAURUS_FORM>
    <THESAURUS_FORM>italian</THESAURUS_FORM>
  </THESAURUS_ENTRY>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>france</THESAURUS_FORM>
    <THESAURUS_FORM>french</THESAURUS_FORM>
  </THESAURUS_ENTRY>
</THESAURUS>
```

Request structure text

When you configure the **WebServiceClient** component, you add the following request text in the **Edit request structure** dialog:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="thesaurus">
<THESAURUS>
  $xmlString
</THESAURUS>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

The `name="thesaurus"` attribute references the `thesaurus` document.

Loading the configuration documents

This section describes how to create and configure a graph for loading the index configuration documents.

The procedure is basically the same for all the index configuration documents. The only exceptions are the format of the input file and the document name used in this element in the **Edit request structure** dialog, as in this example:

```
<config-service:putConfigDocuments
    xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

    xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09
    outerTransactionId="${MDEX_TRANSACTION_ID}">
<mdex:configDocument name="recsearch_config">
<RECSEARCH_CONFIG>
   $xmlString
</RECSEARCH_CONFIG>
</mdex:configDocument>
</config-service:putConfigDocuments>
```

The example shows that the `recsearch_config` document is being loaded. The `$xmlString` variable holds the actual definition of the search interface.

The individual topics for the configuration documents in this chapter describe details of these request structures.

Graph components

The Latitude Sample Application (LSA) uses a graph named LoadIndexingConfiguration to create the search configuration (including creating the search interfaces). The sample graph in this chapter uses the same LSA components that create the search interfaces.

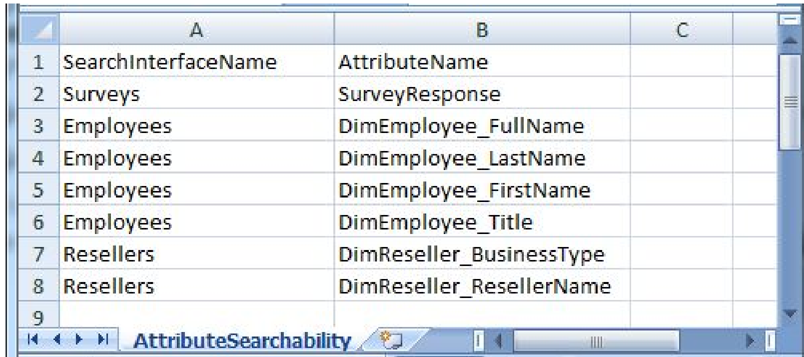
The sample graph in this chapter uses these components, in this order:

1. The **UniversalDataReader** component reads in the configuration document.
2. The **FastSort** transformer sorts the data before it is passed to the **Denormalizer** component (which requires sorted data).
3. The first **Denormalizer** component creates the search interface.
4. The second **Denormalizer** component creates a single output record for the whole group of input records.
5. The **WebServiceClient** writer component uses the Configuration Web Service's `config-service:putConfigDocuments` operation to load the configuration document into the MDEX Engine. The request structure is shown in the `recsearch_config` document example above.

As noted above, the request structure in the **WebServiceClient** component will vary with each configuration document type.

Source file

The sample graph uses the same input CSV file used by the Latitude QuickStart project. The file, named `AttributeSearchability.csv`, looks like this:



	A	B	C
1	SearchInterfaceName	AttributeName	
2	Surveys	SurveyResponse	
3	Employees	DimEmployee_FullName	
4	Employees	DimEmployee_LastName	
5	Employees	DimEmployee_FirstName	
6	Employees	DimEmployee_Title	
7	Resellers	DimReseller_BusinessType	
8	Resellers	DimReseller_ResellerName	
9			

AttributeSearchability

The file provides inputs for three search interfaces:

- The Surveys search interface consists of only the SurveyResponse attribute.
- The Employees search interface consists of the DimEmployee_FullName, DimEmployee_LastName, DimEmployee_FirstName, and DimEmployee_Title attributes.
- The Resellers search interface consists of the DimReseller_BusinessType and DimReseller_ResellerName attributes.

Creating a graph

This task describes how to create an empty graph for loading a configuration document.

The only prerequisite for this task is that you must have created a Data Integrator Designer project. Keep in mind that a project can have multiple graphs, which means that you can create this graph in an existing project.

To create an empty graph for your configuration documents:

1. In the Navigator pane, right-click the **graph** folder.
2. Select **New** > **ETL Graph**.
3. In the **Create new graph** dialog:
   a) Type in the name of the graph, such as **LoadConfigDocs** or **LoadSearchInterfaces**.
   b) Optionally, type in a description.
   c) You can leave the **Allow inclusion of parameters from external file** box checked.
   d) Click **Next** when you finish.
4. In the **Output** dialog, click **Finish**.

Adding components to the graph

This tasks describes how to add the **UniversalDataReader** and **WebServiceClient** components to the graph.

In addition, an Edge component will be added to connect the two components.

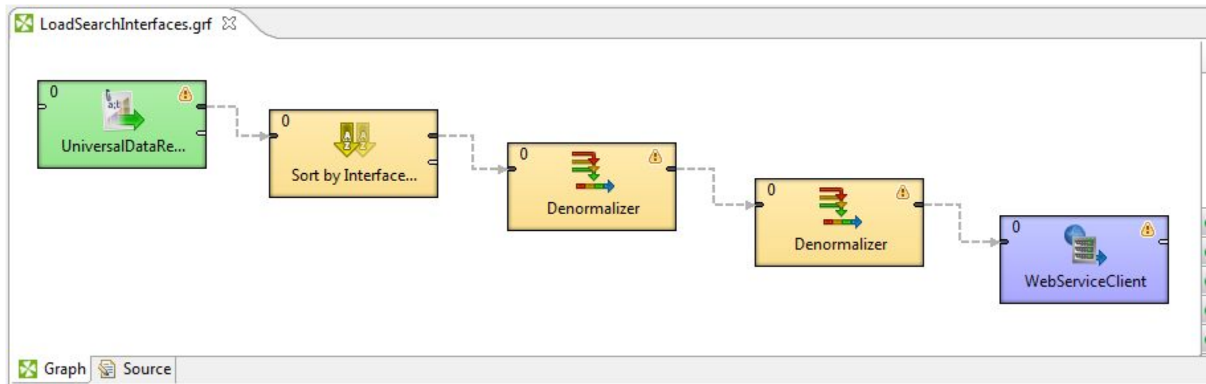
To add components to the graph:

1. In the Palette pane, drag the following components into the Graph Editor:
   a) Drag the **UniversalDataReader** component from the **Readers** section.
   b) Drag the **FastSort** component from the **Transformers** section.
   c) Drag the **Denormalizer** component from the **Transformers** section.

d) Drag a second **Denormalizer** component from the **Transformers** section.
e) Drag the **WebServiceClient** component from the **Others** section.

2. In the Palette pane, click **Edge** and use it to connect the components.
3. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the connected components should look like this:



The next tasks are to configure these components.

Configuring the Reader for the configuration document

This task describes how to configure the **UniversalDataReader** component to read in the configuration document.

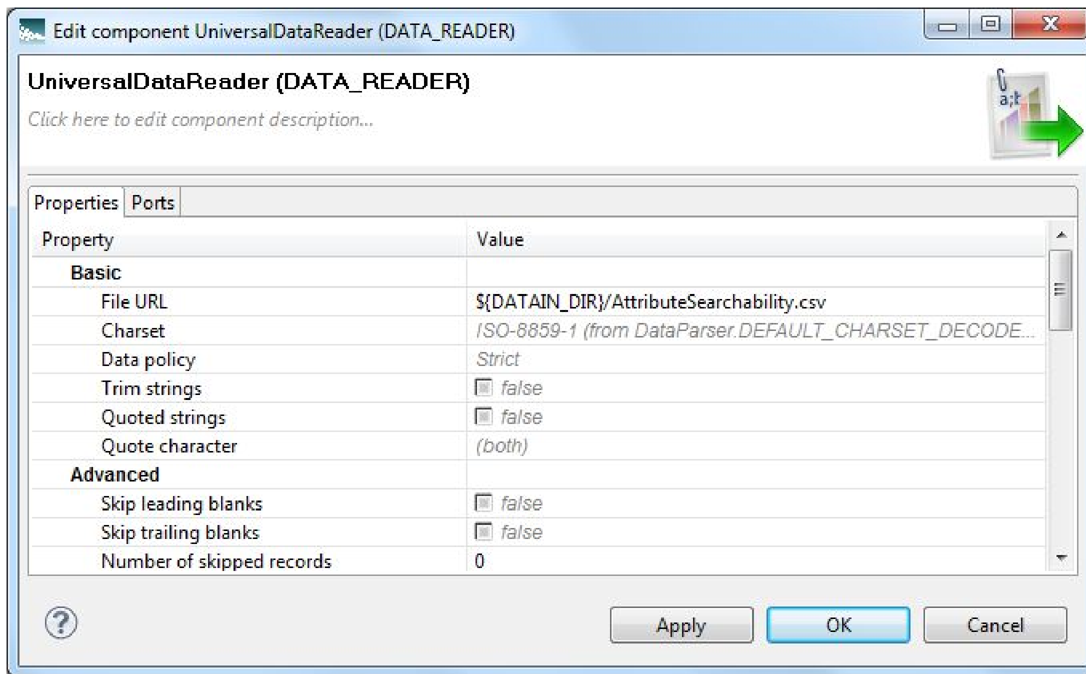
This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the configuration document source file to the project's **data-in** folder.

Important: The procedure also assumes that you have loaded your attribute schema (PDRs and DDRs) into the MDEX Engine. This is because if the configuration document specifies an attribute to use, that attribute should already exist in the MDEX Engine; if it does not exist, the MDEX Engine may reject the configuration document and LDI will display a load error.

To configure the **UniversalDataReader** component for the configuration input document:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the input file folder (either **config-in** or **data-in**).
   d) Select the source data file and click **OK**.
3. Leave the **Quoted strings** box to its default value of `false`.
4. Optionally, you can use the **Component name** field to provide a customized name (such as "Read Searchable Attributes") for this component.
5. Click **OK** to apply your configuration changes to the **UniversalDataReader** component.
6. Save the graph.

After step 5, the Reader Edit Component dialog should look like this example:



The next task is to configure the Reader's Edge.

Configuring metadata for the Reader Edge

The Edge component must be configured with a Metadata definition for loading a configuration document.

The prerequisite for this task is that an Edge component must connect the Reader and the following component.

Note: This procedure will configure metadata for loading the `recsearch_config` configuration document. The procedure for loading the other configuration documents is identical, with the exception that at Step 3 you select the name of the appropriate input file.

To configure the Metadata definition for configuration documents:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. In the URL Dialog, double-click the input folder (such as **config-in**), select the source data file, and click **OK**.
   As a result, the Flat File dialog is populated with source data from the input file.
4. In the Flat File dialog, click **Next**.
   The Metadata Editor is displayed.
5. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.

The correct property names are displayed in the upper and middle panes of the Metadata Editor, which should look like this example:

#	Name	Type	Delimiter
1	Record: AttributeSearchability_csv	delimited	,
1	SearchInterfaceName	string	,
2	AttributeName	string	\r\n

Filter:

6. In the Record pane, make these changes to the **Record** row:
   a) Click the **Record** Name field and change its name to a more descriptive one, such as **SearchInterfaces**.
   b) Leave the **Type** and **Delimiter** fields to their default settings.
7. Change the name of the **SearchInterfaceName** property to be **InterfaceName**.
8. When you have input all your changes in the Metadata Editor, click **Finish**.
9. Save the graph.

Configuring the FastSort component

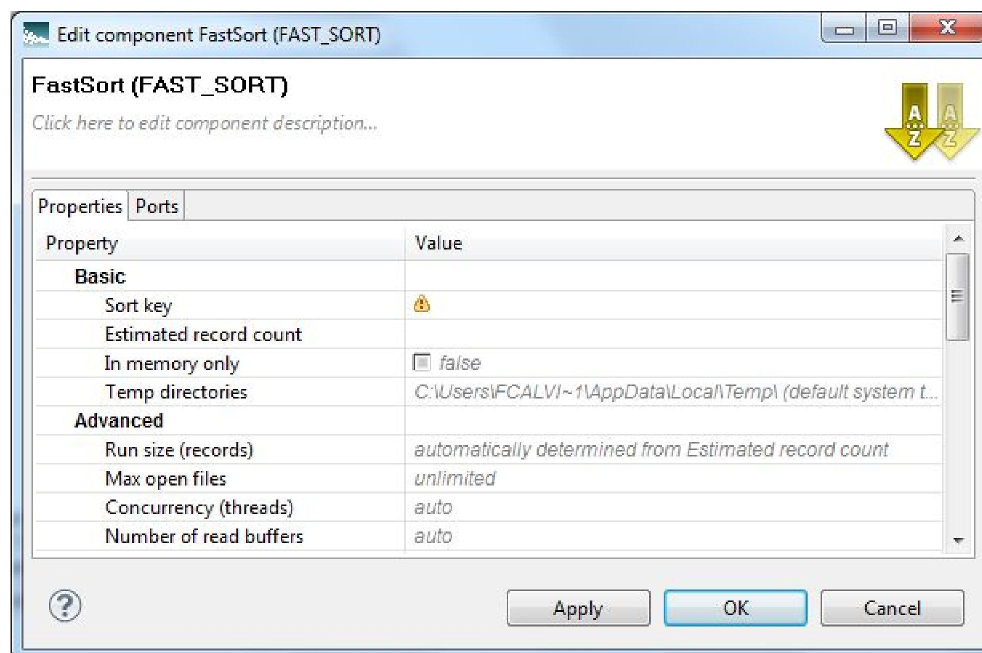
The FastSort component takes input records and sorts them using a sorting key.

This component is necessary because the **Denormalizer** component (the next component in the graph) takes sorted data.

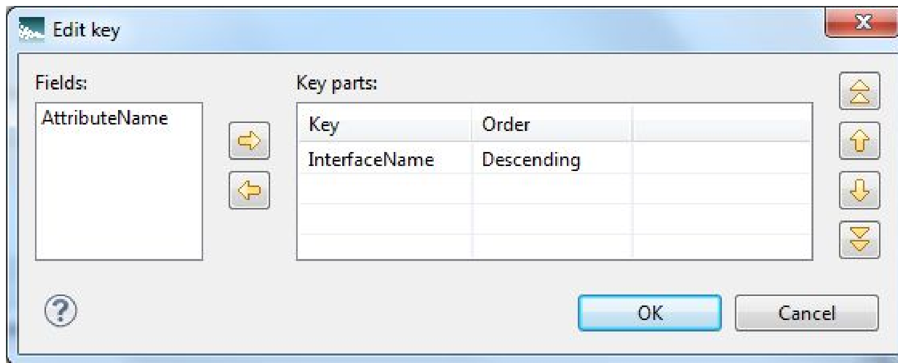
To configure the **FastSort** component:

1. In the Graph window, double-click the **FastSort** component.

   The FastSort Edit Component dialog is displayed.



2. Single-click in the **SortKey** field and then click the **...** button.
   The Edit Key editor is displayed.
3. In the Edit Key editor:
   a) In the Fields pane, select the **InterfaceName** attribute.
   b) Click the right-arrow button to move the SearchInterfaceName attribute to the Key Parts pane.
   c) In the Key Parts pane, toggle the Order field to **Descending**. At this point, the Edit Key editor should look like this:



   d) Click **OK** to exit the Edit Key editor.
4. Optionally, you can use the **Component name** field to provide a customized name (such as "Sort by Interface Name") for this component.
5. In the FastSort Edit Component dialog, click **OK** to apply your changes and exit the component.
6. Save the graph.

Setting metadata for the FastSort component

The configuration of the Edge for the **FastSort** component is the same as for the **UniversalDataReader** component.

This procedure assumes that **SearchInterfaces** is the name of the metadata of the **UniversalDataReader** component.

To set the metadata for the **FastSort** Edge:

1. Right-click on the Edge and choose **Select metadata** > **SearchInterfaces**.
2. Save the graph.

Configuring the first Denormalizer component

The first **Denormalizer** component creates the search interface from the input data.

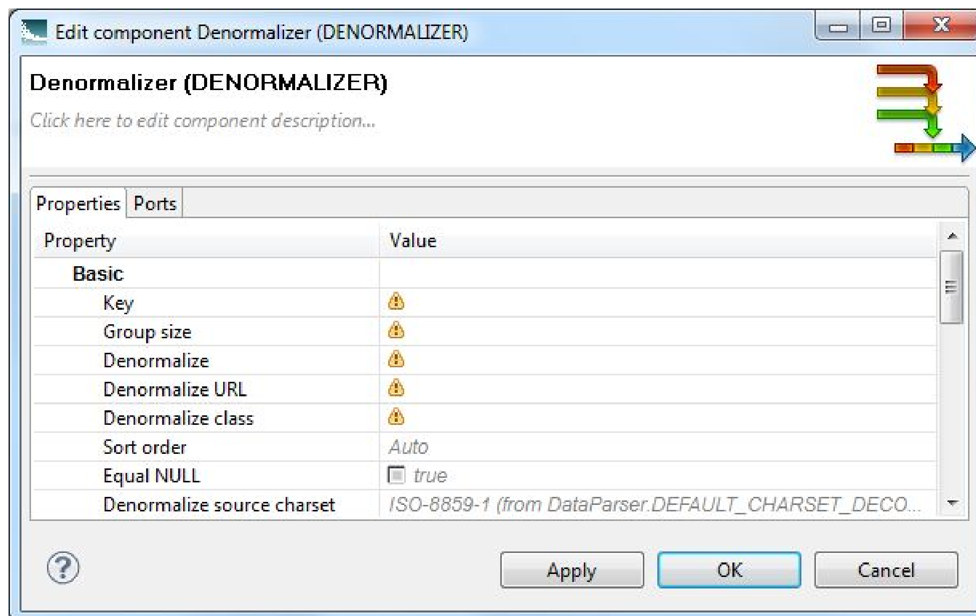
The transformation is done by these CTL functions in this **Denormalizer** component:

```
//#CTL2
// This transformation defines the way in which multiple input records
// (with the same key) are denormalized into one output record.
integer n = 0;
integer relRank = 1;
string value = "";
string nameOfInterface = "";

// This function is called for each input record from a group of records
```

```
// with the same key.
function integer append() {
  n++;
  value =
     value + "<MEMBER_NAME RELEVANCE_RANK='" + num2str(relRank) + "'>" +
     $0.AttributeName + "</MEMBER_NAME>";

  nameOfInterface = $0.InterfaceName;
  relRank++;

  return n;
}

// This function is called once after the append() function was called for
 all records
// of a group of input records defined by the key.
// It creates a single output record for the whole group.
function integer transform() {
  $0.xmlString = "<SEARCH_INTERFACE NAME=\"" + nameOfInterface + "\">"
       + value
       + "</SEARCH_INTERFACE>";
  $0.singleAggregationKey = 0; // constant (aggregate everything into one
request)

  value = "";
  nameOfInterface = "";
  relRank = 1;

  return OK;
}
```

To configure the first **Denormalizer** component in the graph:

1. In the Graph editor, double-click the first **Denormalizer** component.
   The Denormalizer Edit Component dialog is displayed.



2. Single-click in the **Key** field and then click the **...** button.

The Edit Key dialog is displayed.

3. In the **Fields** pane of the Edit Key dialog, select **InterfaceName** and move it to the **Key parts** pane by clicking the right-arrow button. Click **OK** to apply your change.
4. Single-click in the **Denormalize** field and then click the **...** button.
   The Transform editor is displayed.
5. In the **Source** tab of the editor, modify the CTL script so that it looks like the example above.
   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.
6. When you have finished your edits, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
7. Optionally, you can use the **Component name** field to provide a customized name for this component.
8. Click **OK** to apply your configuration changes.
9. Save the graph.

The two messages listed above should disappear once you configure the **Denormalizer** component Edge metadata.

Configuring metadata for the first Denormalizer

This task describes how to configure the Edge component that connects the first and second **Denormalizer** components.

To configure the metadata for the first **Denormalizer** component:

1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.
2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a descriptive name. Our example will use **DenormEdge** as the name.
   b) Leave the **Type** field as `delimited`.
   c) Set the **Delimiter** field to the delimiter character in your input file (which is the comma in our example).
3. For the other fields:
   a) Change the **field1** name to `xmlString` and leave its **Type** as `string`.
   b) Add a new field by using the + (plus sign control). Name the field **singleAggregationKey** and set its **Type** as `integer`.
4. When you have input all your changes in the Metadata editor, click **Finish**.
5. Save the graph.

Later, we will use this **DenormEdge** metadata for the second **Denormalizer** component.

Configuring the second Denormalizer component

The second **Denormalizer** component builds a single request.

The transformation is done by these CTL functions in this **Denormalizer** component:

```
//#CTL2
// This transformation defines the way in which multiple input records
```

```
// (with the same key) are denormalized into one output record.

// This function is called for each input record from a group of records
// with the same key.
integer n = 0;
string value = "";

function integer append() {
  value = value + $0.xmlString + "\n";
  n++;
  return n;
}

// This function is called once after the append() function was called for
 all records
// of a group of input records defined by the key.
// It creates a single output record for the whole group.
function integer transform() {
  $0.xmlString = value;
  value = "";
  return OK;
}
```

To configure the second **Denormalizer** component in the graph:

1. In the Graph editor, double-click the second **Denormalizer** component.
   The Denormalizer Edit Component dialog is displayed.
2. Single-click in the **Key** field and then click the **...** button.
   The Edit Key dialog is displayed.
3. In the **Fields** pane of the Edit Key dialog, select **singleAggregationKey** and move it to the **Key parts** pane by clicking the right-arrow button. Click **OK** to apply your change.
4. Single-click in the **Denormalize** field and then click the **...** button.
   The Transform editor is displayed.
5. In the **Source** tab of the editor, modify the CTL script so that it looks like the example above.
   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.
6. When you have finished your edits, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
7. Optionally, you can use the **Component name** field to provide a customized name for this component.
8. Click **OK** to apply your configuration changes.
9. Save the graph.

The two messages listed above should disappear once you configure this **Denormalizer** component Edge metadata.

The configuration of the Edge for this second **Denormalizer** component is the same as for the first **Denormalizer** component. In fact, you can use the same metadata that you created for the first one.

Setting metadata for the second Denormalizer

The configuration of the Edge for the second **Denormalizer** component is the same as for the first one.

This procedure assumes that **DenormEdge** is the name of the metadata of the first **Denormalizer** component. We will re-use that metadata for this second **Denormalizer** component.

To set the metadata for the Edge that connects the second **Denormalizer** component and the **WebServiceClient** component:

1. Right-click on the Edge and choose **Select metadata** > **DenormEdge**.
2. Save the graph.

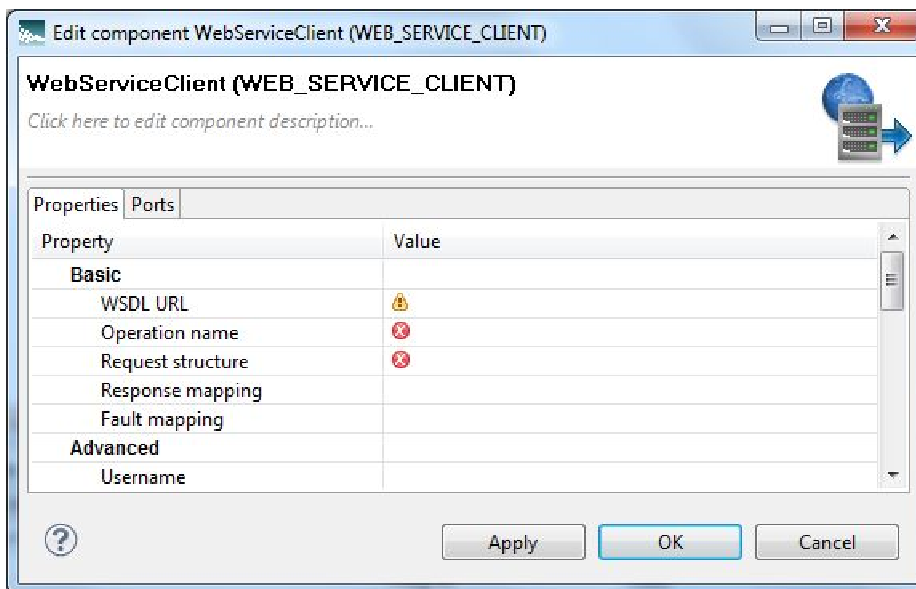
Configuring the WebServiceClient component

You must configure the **WebServiceClient** component to communicate with the Endeca Configuration Web service.

This procedure will configure metadata for loading the `recsearch_config` configuration document, and therefore assumes that you have added the configuration document source file to the project's **data-in** folder. The procedure for loading the other configuration documents with the **WebServiceClient** component is identical, with the exception that at Step 7 you specify the name of the appropriate configuration document in the `mdex:configDocument` element:

```
<mdex:configDocument name="recsearch_config">
```

The Writer Edit Component dialog is where you configure the **WebServiceClient** component:



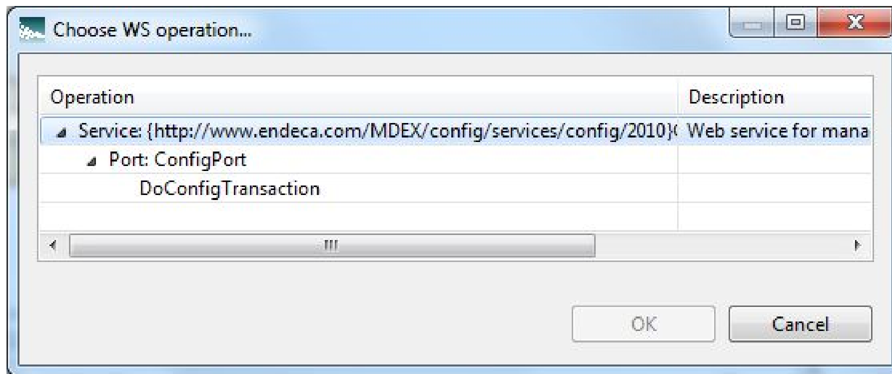
To configure the **WebServiceClient** component:

1. Make sure that the MDEX Engine is running and the Configuration Web service is available by issuing this URL command from your browser (be sure to use the correct port number for your MDEX Engine):

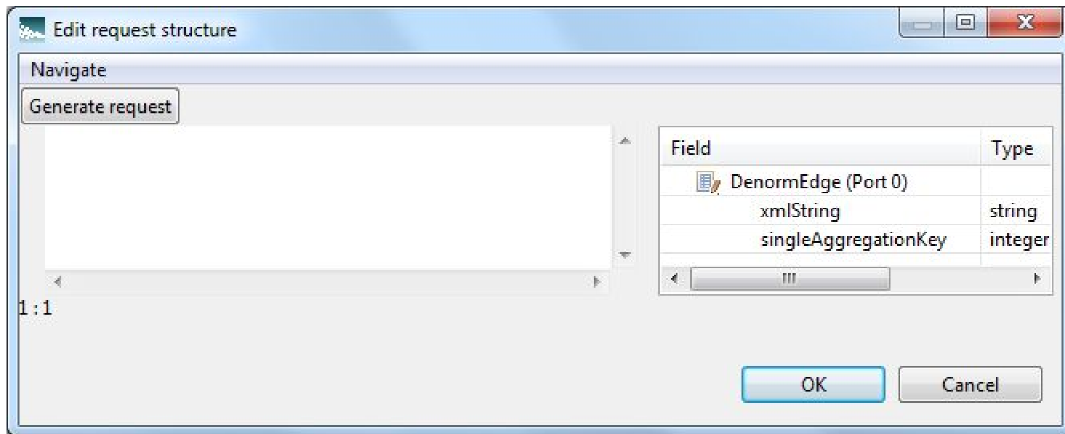
   ```
   http://localhost:5555/ws/config?wsdl
   ```

   The URL command returns the WSDL of the Web service.
2. In the Graph window, double-click the **WebServiceClient** component.
   The Writer Edit Component dialog is displayed.
3. In the **WSDL URL** field, enter the same URL as in Step 1.

4. In the **Operation name** field, click the **...** browse button, which displays the **Choose WS operation** dialog:



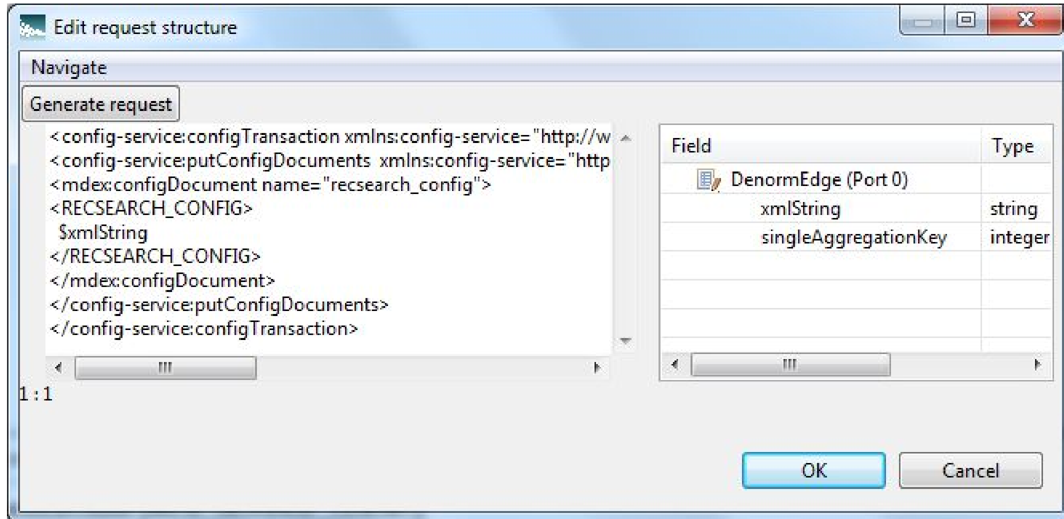
5. In the **Choose WS operation** dialog, select **DoConfigTransaction** and then click **OK**.
   The name of the Web service operation is entered in the **Operation name** field.
6. Click inside the **Request structure** field, which causes the **...** browse button to be displayed. Then click the browse button to display the **Edit request structure** dialog:



7. Add this text to the **Generate request** field:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putConfigDocuments
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
<mdex:configDocument name="recsearch_config">
<RECSEARCH_CONFIG>
  $xmlString
</RECSEARCH_CONFIG>
</mdex:configDocument>
</config-service:putConfigDocuments>
</config-service:configTransaction>
```

At this point, the **Edit request structure** dialog should look like this example:



8. After adding the request text in the **Edit request structure** dialog, click **OK**.
9. Optionally, you can use the **Component name** field to provide a customized name (such as "Load Configs") for this component.
10. When you have entered all your changes in the Edit Component dialog, click **OK**.
11. Save the graph.

Instead of running this graph directly, it is recommended that you create a transaction graph (with a **Transaction RunGraph** connector) with this LoadSearchInterfaces graph as its child graph, and then run the transaction graph. For details on transaction graphs, see Chapter 3 ("Working with Transaction Graphs").

Loading the GCR

This topic provides an overview of how to load the GCR into the MDEX Engine.

Loading the Global Configuration Record (GCR) into the MDEX Engine is very similar to loading the index configuration documents. The only difference is the format of the request text that you add to the **Edit request structure** dialog. This GCR-specific request text is shown in Step 6 below.

To load the GCR:

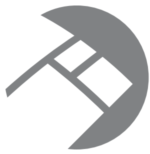
1. Create a graph, as described in the "Creating a graph" topic in this chapter.
2. Add your GCR input file to the project's **data-in** folder.
3. Add the **UniversalDataReader** and **WebServiceClient** components to the graph. You can build the GCR output xmlString with **Reformat** and **Denormalizer** components, similar to the graph that loads the standard attribute schema (described in Chapter 6, "Loading the Attribute Schema").
4. Configure the Reader and Transformation components and their metadata.
5. Configure the **WebServiceClient** component, as described in the "Configuring the WebServiceClient component" topic in this chapter. The only difference is that you add this text to the **Generate request** field:

```
<config-service:configTransaction
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

  outerTransactionId="${MDEX_TRANSACTION_ID}">
```

```
<config-service:putGlobalConfigRecord
  xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
  $xmlString
</config-service:putGlobalConfigRecord>
</config-service:configTransaction>
```

6. Make sure you save the graph.

Note that if you changed the spelling settings, you should rebuild the aspell dictionary by running the `admin?op=updateaspell` administrative operation.



Chapter 8

Loading Precedence Rules

This chapter describes how to load your precedence rules into the MDEX Engine.

About precedence rules

A precedence rule allows your application to suppress refinements for an Endeca attribute until some condition is met. This makes navigation through the data easier and is essential to avoid information overload problems.

Precedence rules allow your application to delay the display of Endeca standard or managed attributes the user triggers the display. In other words, precedence rules are triggers that cause attributes that were not previously displayed to now be available. This makes navigation through the data easier, and is essential to avoid information overload problems.

For example, suppose the records in an application have separate City and State attributes. It would make sense to hide the City attribute until the user has narrowed down to a specific State, because it doesn't make sense to pick a City before a State. (For example, choosing "Portland" would select records in both Portland, OR and Portland, ME.) To accomplish this, create a precedence rule with State as the trigger and City as the target.

The standard and/or managed attributes referenced in precedence rules do not have to exist in the MDEX Engine at ingest time. That is, no error checking is done for the existence of the attributes (this allows the rules to be created even before the data they reference is loaded). For this reason, you must make sure that the attributes are spelled correctly in the input file.

Note that if the trigger attribute in a precedence rule does not exist in the MDEX Engine but its target attribute does exist, then the precedence rule will never be triggered. This behavior effectively hides the target attribute from refinements. To correct this behavior, either remove the rule or create the trigger attribute in the MDEX Engine.

Schema for precedence rules

Each precedence rule is represented as a single record in the MDEX Engine.

The `config-service:putPrecedenceRules` operation creates each of the given precedence rules or updates them if they already exist. Each `precedenceRule` element uses this schema syntax:

```
<mdex:precedenceRule
   key="ruleName"
```

```
   triggerAttributeKey="triggerAttrName"
   triggerAttributeValue="mval|sval"
   targetAttributeKey="targetAttrName"
   isLeafTrigger="true|false"/>
```

The meanings of the `precedenceRule` attributes are as follows:

precedenceRule attribute	Meaning
`key`	Specifies a unique identifier for the precedence rule (that is, it is the name of the rule). The identifier is a string, which does not have to follow the NCName format.
`triggerAttributeKey`	Specifies the name of the Endeca standard attribute or managed attribute that will trigger the precedence rule. That is, the specified attribute must be selected before the user can see the target attribute.
`triggerAttributeValue`	Optional. If used, specifies the attribute value (either managed value spec or standard attribute value) that must be selected before the user can see the target attribute. If not used, then any value in the trigger attribute will trigger the rule. Use of `triggerAttributeValue` in effect further refines the trigger to a specific standard or managed value.
`targetAttributeKey`	Specifies the name of the Endeca standard or managed attribute that appears after the trigger attribute value is selected.
`isLeafTrigger`	If the trigger is a managed attribute, `isLeafTrigger` specifies a Boolean value (that must be in lower case) that denotes the type of the trigger attribute value: • If `true`, the trigger attribute is a leaf type, which means that the precedence rule will fire only if a leaf value is selected. That is, querying any leaf managed value from the trigger managed attribute will cause the target managed value to be displayed (many triggers, one target). • If `false` (the default), the trigger attribute is a non-leaf type, which means that the precedence rule will fire when any value is selected. That is, if the managed value specified as the trigger or any of its descendants are in the navigation state, then the target is presented (one trigger, one target). Note that `isLeafTrigger` does not apply to Endeca standard attributes. You must specify it when you create a precedence rule, but whichever value you use is ignored by the MDEX Engine when the precedence rule is run.

Precedence rule example

The following is an example of a `config-service:putPrecedenceRules` operation that creates a precedence rule named ProvinceRule:

```
<config-service:configTransaction
   xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

   xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
  <config-service:putPrecedenceRules>
```

```
    <mdex:precedenceRule
       key="ProvinceRule"
       triggerAttributeKey="DimGeography_StateProvinceName"
       triggerAttributeValue="Queensland"
       targetAttributeKey="DimGeography_City"
       isLeafTrigger="true"/>
  </config-service:putPrecedenceRules>
</config-service:configTransaction>
```

Note that this example does not use the optional `outerTransactionId` attribute for the operation. This operation can be placed in a request structure of a **WebServiceClient** component.

Format of the precedence rules input file

The input configuration file should contain five configuration properties and a corresponding set of value data.

The first line (the header row) of a precedence rules input file should have these header properties:

```
Key|TriggerAttribute|TriggerValue|TargetAttribute|isLeafTrigger
```

The actual names of the header properties in your input file can be different from the names used here (for example, you can use `RuleName` instead of `Key`). The properties are delimited (for example, by the comma in a CSV file or the pipe character in a text file).

The header properties map to the `precedenceRule` attributes as follows:

Input Header Property	Maps to precedenceRule attribute	Description
`Key`	`key`	Name of the precedence rule.
`TriggerAttribute`	`triggerAttributeKey`	Name of the standard or managed attribute trigger.
`TriggerValue`	`triggerAttributeValue`	Standard or managed attribute value for the trigger. Optional, so the value in the input file can be blank.
`TargetAttribute`	`targetAttributeKey`	Name of the standard or managed attribute target.
`isLeafTrigger`	`isLeafTrigger`	For managed attributes, specifies if the trigger attribute is a leaf.

After the header row, the second and following rows in the input file contain configuration data for the precedence rules. The following image shows a CSV configuration file for two precedence rules:

	A	B	C	D	E
1	Key	TriggerAttribute	TriggerValue	TargetAttribute	isLeafTrigger
2	AUS_Rule	DimGeography_CountryRegionName	Australia	DimGeography_StateProvinceName	FALSE
3	City_Rule	DimGeography_StateProvinceName		DimGeography_City	FALSE
4					
5					

PrecedenceRules

Ready 100%

Note that the `TriggerValue` for the second precedence rule is blank, which means that any value in the DimGeography_StateProvinceName attribute will trigger the rule.

Adding components to the precedence rules graph

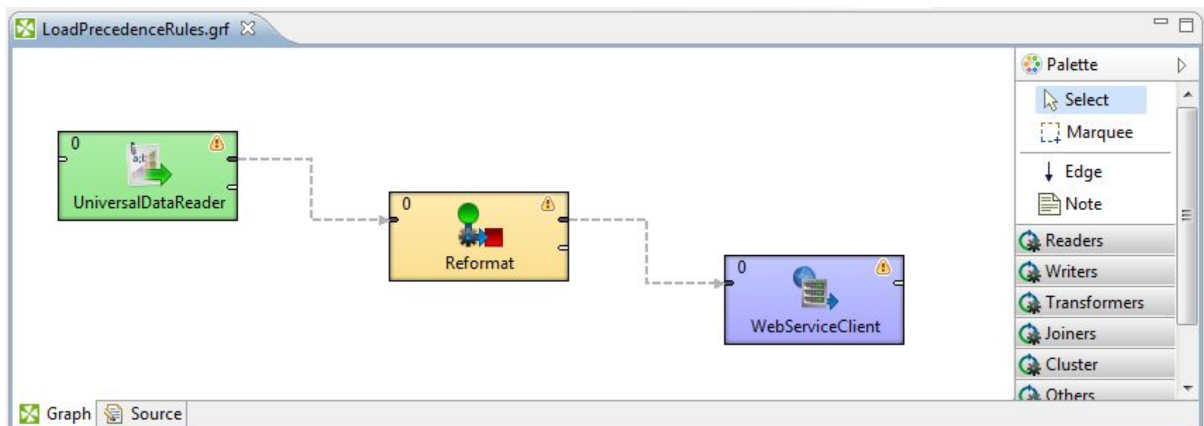
You must add the **UniversalDataReader**, **Reformat**, and **WebServiceClient** components to the graph.

This procedure assumes that you have created an empty graph.

To add components to the graph:

1. In the Palette pane, drag the following components into the Graph Editor:
   a) Drag the **UniversalDataReader** component from the **Readers** section.
   b) Drag the **Reformat** component from the **Transformers** section.
   c) Drag the **WebServiceClient** component from the **Others** section.
2. In the Palette pane, click **Edge** and use it to connect the components.
3. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the connected components should look like this:



The next tasks are to configure these components.

Configuring the precedence rules Reader

This task describes how to configure the **UniversalDataReader** component to read in the configuration file for creating precedence rules.

This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the precedence rule configuration file to the project's **data-in** (or **config-in**) folder.

To configure the **UniversalDataReader** component for the precedence rules configuration input file:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:

a) Click inside its Value field, which displays a **...** browse button.
b) Click the browse button.
c) Click the **Workspace view** tab and then double-click the input file folder (either **config-in** or **data-in**).
d) Select the configuration input file and click **OK**.

3. Optionally, you can use the **Component name** field to provide a customized name (such as "Read Rules Metadata") for this component.
4. Click **OK** to apply your configuration changes to the **UniversalDataReader** component.
5. Save the graph.

The next task is to configure the Reader's Edge.

Configuring the Reader Edge

This task describes how to configure the Reader Edge component for the Metadata definition.

To configure the Reader Edge component:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. In the URL Dialog:
   a) Double-click the input folder.
   b) Select the configuration source file and click **OK**.
4. In the Flat File dialog, click **Next**.
   The Metadata Editor is displayed.
5. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.
6. In the Record pane, you should change the **recordName1** default value to a name that is appropriate for your data.
7. When you have input all your changes, click **Finish**
8. Save the graph.

Configuring the Reformat component for precedence rules

A **Reformat** component is used to transform incoming configuration data into a `precedenceRule` record.

The transformation is done by this CTL function in the **Reformat** component:

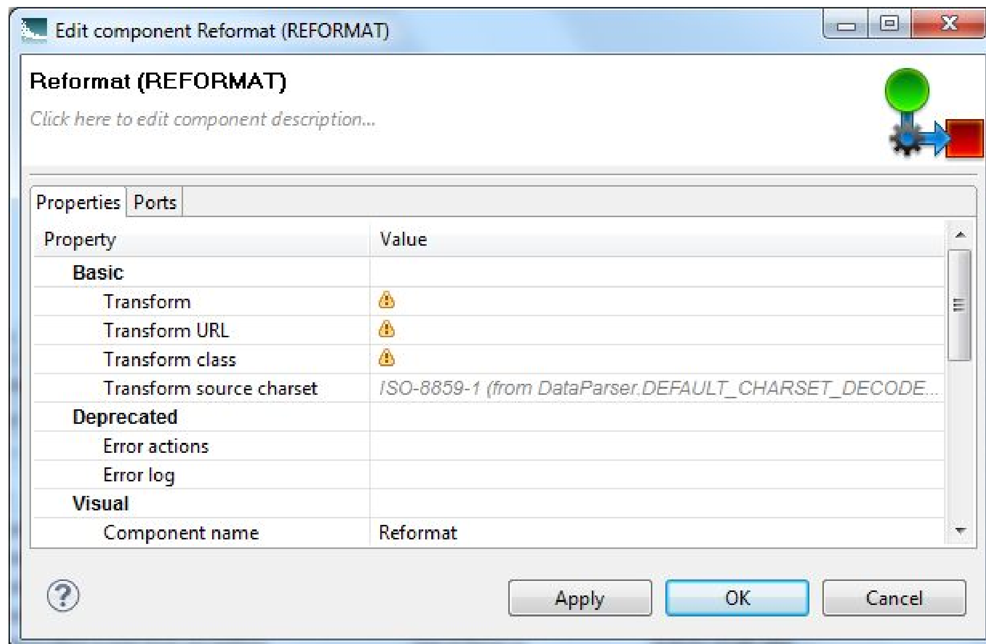
```
function integer transform() {
   string prRecord = "";
   string isLeaf = "";

   // Begin building the precedenceRule record
   prRecord = "<mdex:precedenceRule ";
   // Add the name of the rule.
```

```
   prRecord = prRecord + "key='" + $0.Key + "' ";
   // Add the name of the trigger attribute
   prRecord = prRecord + "triggerAttributeKey='" + $0.TriggerAttribute +
"' ";

   // Add mval or pval trigger value only if present in the input file
   if ($0.TriggerValue != null && !$0.TriggerValue.isBlank()) {
      prRecord = prRecord + "triggerAttributeValue='" + $0.TriggerValue +
"' ";
   }

   // Add the name of the target attribute
   prRecord = prRecord + "targetAttributeKey='" + $0.TargetAttribute + "'
";

   // Add the boolean that specifies if the trigger is a leaf
   // Lower case the boolean in the CSV file
   isLeaf = lowerCase($0.isLeafTrigger);
   prRecord = prRecord + "isLeafTrigger='" + isLeaf + "'/>";

   // Append the record to the xmlString variable, which stores all the
rules
   $0.xmlString = prRecord;

   return ALL;
}
```

When it runs, the component will build one or more `precedenceRule` elements and send them in the `xmlString` property to the **WebServiceClient** component in the graph.

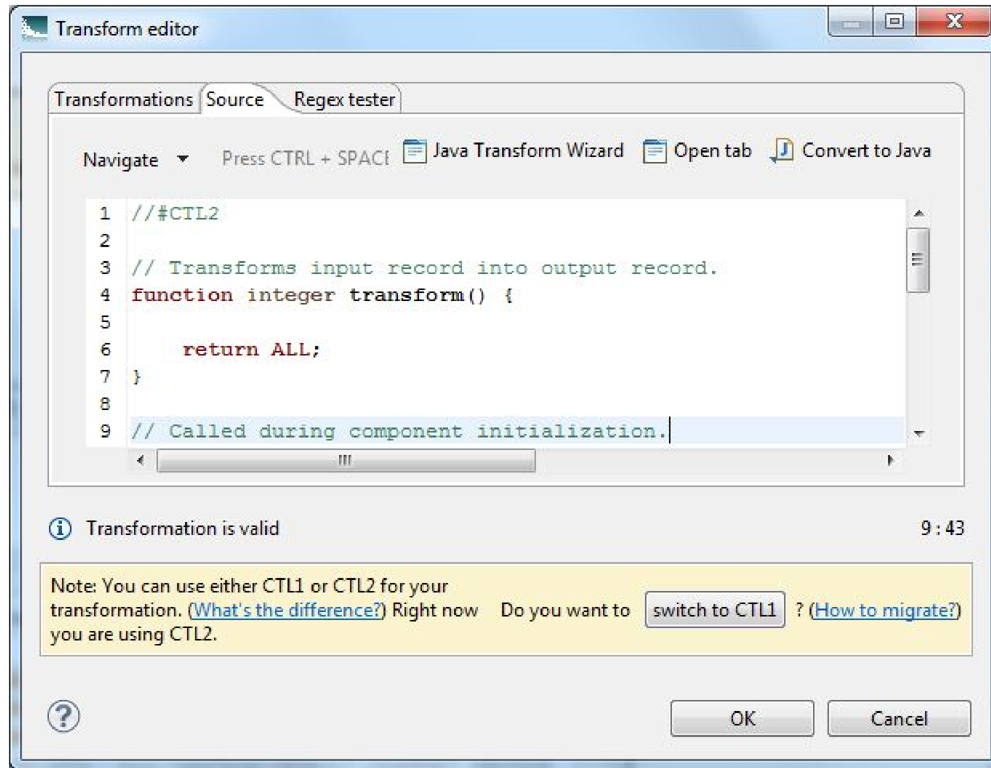
To configure the **Reformat** component in the precedence rules graph:

1. In the Graph window, double-click the **Reformat** component.
   The Reformat Edit Component dialog is displayed.



2. Single-click in the **Transform** field and then click the **...** button.
   The Transform editor is displayed.

3. Click the **Source** tab in the editor.
   The CTL template for the transform function is shown.



4. Modify the CTL script so that it looks like the example above.
   You may see the message "Cannot write to output port '0'" at the bottom of the editor. Assuming you have not made any coding errors, you may disregard the message for now.
5. When you have finished your edits, click **OK**.
   If you see the error "Transformation contains syntax errors! Accept it anyway?" in a pop-up message, click **Yes**.
6. Optionally, you can use the **Component name** field to provide a customized name (such as "Transform Precedence Rules") for this component.
7. Click **OK** to apply your configuration changes.
8. Save the graph.

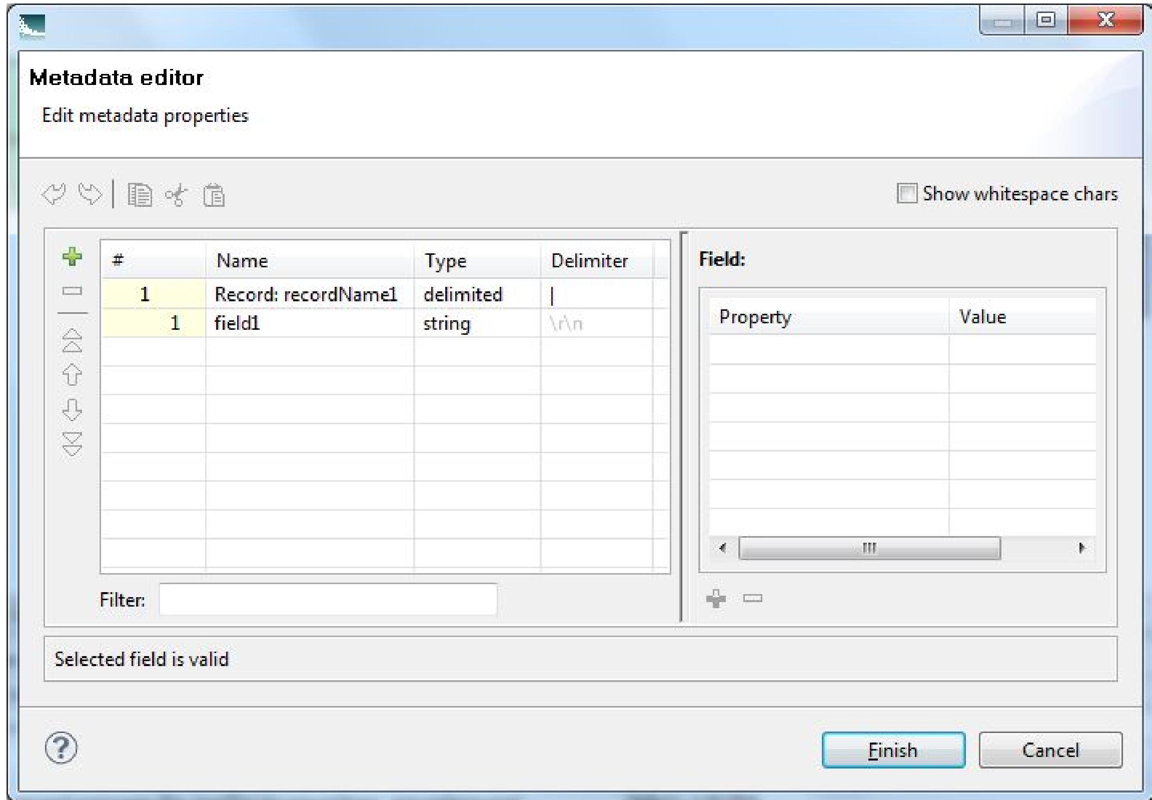
The two messages listed above should disappear once you configure the **Reformat** component Edge metadata.

Configuring the precedence rules Reformat Edge

This task describes how to configure the Edge component that connects the **Reformat** and **WebServiceClient** components.

To configure the **Reformat** component's Edge in the precedence rules graph:

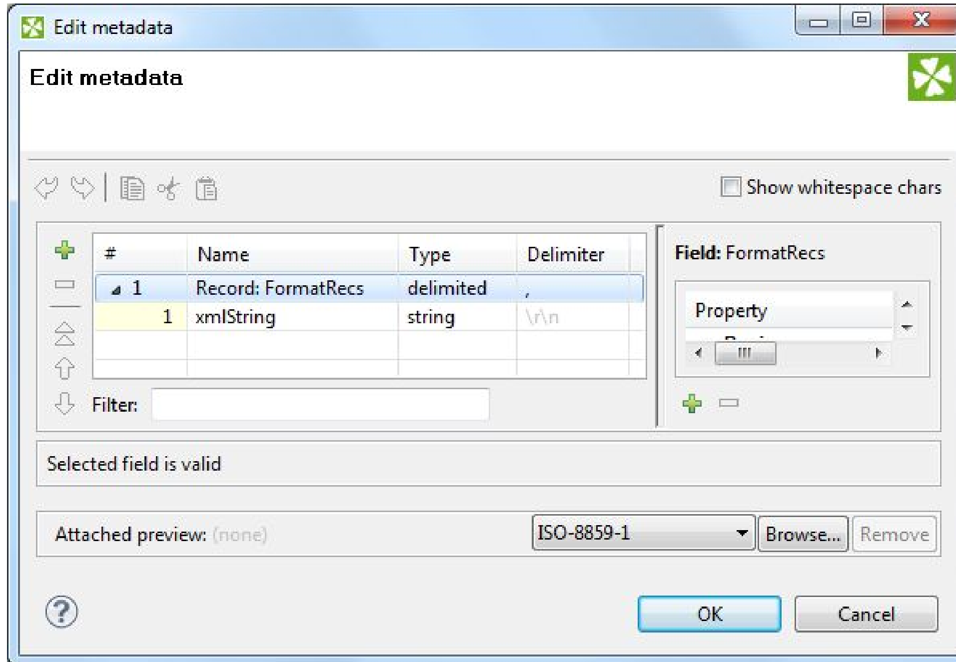
1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a name that is appropriate for your data, such as `FormatRules`.
   b) Leave the **Type** field as `delimited`.
   c) Set the **Delimiter** field to the delimiter character in your input file (which is the comma in our example).
3. Change the **field1** name to `xmlString` and leave its **Type** as `string`.

   You can leave the **Delimiter** field unchanged.

   At this point, the Metadata editor should look like this:



4. When you have input all your changes in the Metadata editor, click **Finish**.
5. Save the graph.

Configuring the precedence rules WebServiceClient component

The **WebServiceClient** component must be configured with the WSDL of the MDEX Engine's Configuration Web service.

In addition, you must add a `config-service:putPrecedenceRules` operation to the request structure of the component.

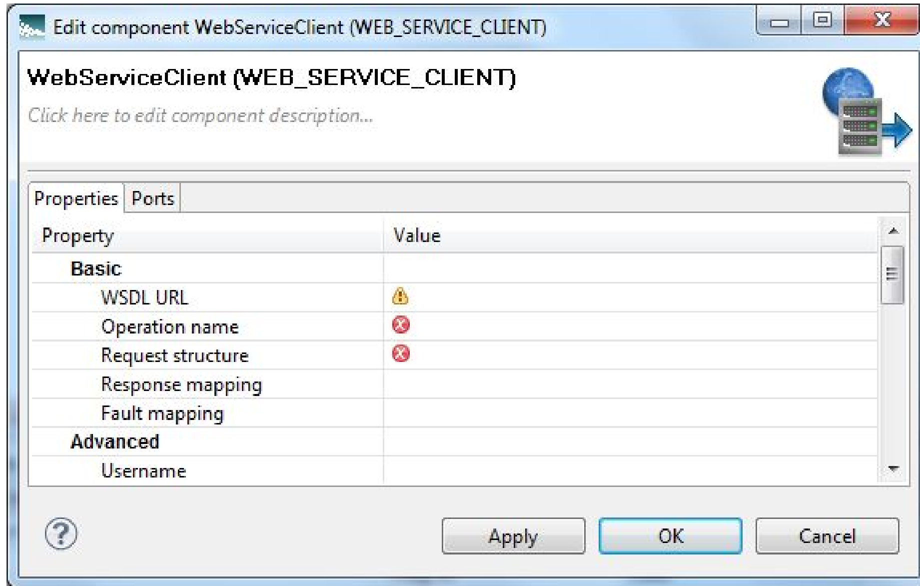
To configure the **WebServiceClient** component in the precedence rules graph:

1. Make sure that the MDEX Engine is running and the Configuration Web service is available by issuing this URL command from your browser (be sure to use the correct port number for your MDEX Engine):

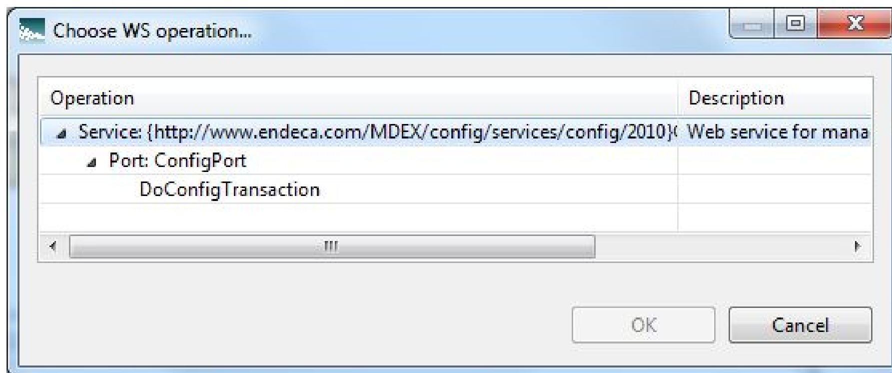
   ```
   http://localhost:5555/ws/config?wsdl
   ```

   The URL command returns the WSDL of the Web service.
2. In the Graph window, double-click the **WebServiceClient** component.

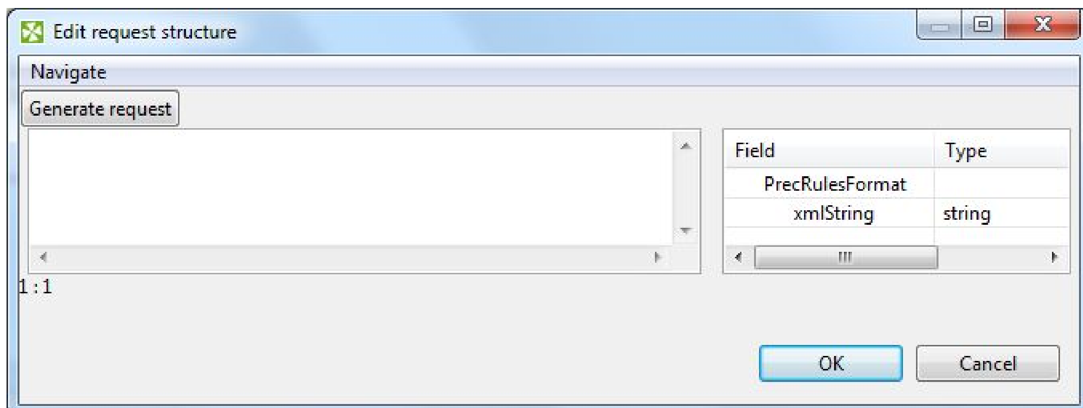
   The Writer Edit Component dialog is displayed.



3. In the **WSDL URL** field, enter the same URL as in Step 1.
4. In the **Operation name** field, click the **...** browse button, which displays the **Choose WS operation** dialog:



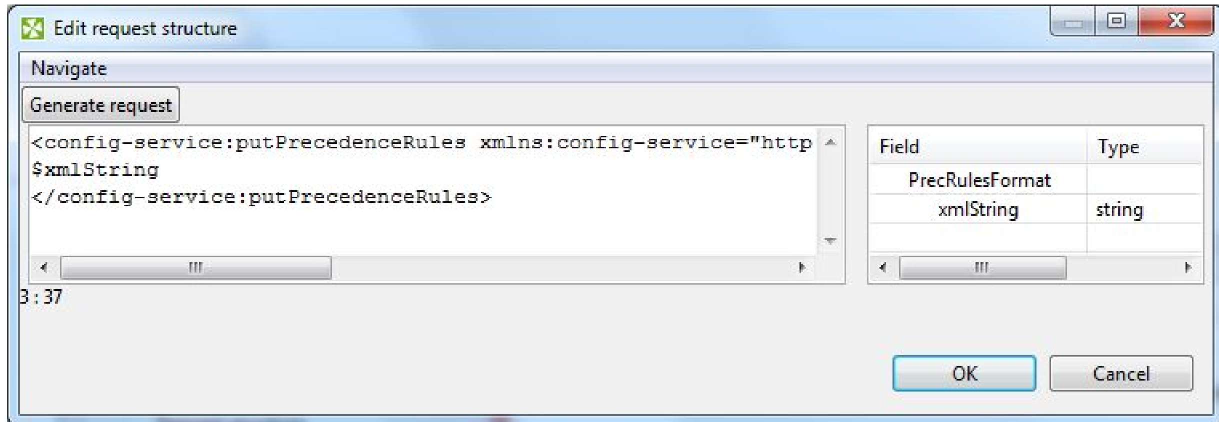
5. In the **Choose WS operation** dialog, select **DoConfigTransaction** and then click **OK**. The name of the Web service operation is entered in the **Operation name** field.
6. Click inside the **Request structure** field, which causes the **...** browse button to be displayed. Then click the browse button to display the **Edit request structure** dialog:



7. Add this text to the **Generate request** field:

```
<config-service:putPrecedenceRules
   xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

   xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
$xmlString
</config-service:putPrecedenceRules>
```

At this point, the **Edit request structure** dialog should look like this example:



8. After adding the request text in the **Edit request structure** dialog, click **OK**.
9. Optionally, you can use the **Component name** field to provide a customized name (such as "Load Precedence Rules") for this component.
10. When you have entered all your changes in the Edit Component dialog, click **OK**.
11. `Save the graph.`

After creating the graph and configuring the components, you can run the graph to send the configuration data to the MDEX Engine. You can run the graph by clicking the green circle with white triangle icon in the Tool bar:

Deleting precedence rules

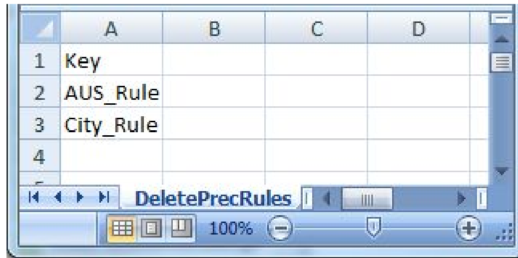
The `config-service:deletePrecedenceRules` operation lets you remove an existing precedence rule from the MDEX Engine.

The Configuration Web Service's `putPrecedenceRules` operation takes one or more `precedenceRule` elements that will be deleted. Because precedence rules are stored as records in the MDEX Engine, you need to specify only the `key` attribute of the precedence rule, as in this example that deletes a precedence rule named "ProvinceRule":

```
<config-service:configTransaction
    xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

     xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
   <config-service:deletePrecedenceRules>
      <mdex:precedenceRule key="ProvinceRule"/>
   </config-service:deletePrecedenceRules>
</config-service:configTransaction>
```

To delete precedence rules from the MDEX Engine:

1. Create an input file that contains one column, with a **Key** header name and with one or more rows of precedence rule names, as in this simple CSV example:

	A	B	C	D
1	Key			
2	AUS_Rule			
3	City_Rule			
4				

2. Create a graph and add the components described in the "Adding components to the precedence rules graph" topic in this chapter.
3. Configure the **UniversalDataReader** component as described in the "Configuring the precedence rules Reader" topic in this chapter.

   Make sure you use the file created in Step 1 as the input file and that the **Number of skipped records per source** field is set to 1.
4. Use the "Configuring the precedence rules Reader Edge" topic in this chapter to configure the Reader Edge.
   Note that the **Record** field's Delimiter field will be empty, as there is only one column.
5. Configure the **Reformat** component so that the CTL in the **Source** tab looks like this:

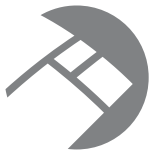
```
function integer transform() {
   string prRecord = "";

   prRecord = "<mdex:precedenceRule key='" + $0.Key + "'/>";

   $0.xmlString = prRecord;

   return ALL;
}
```

6. Use the "Configuring the precedence rules Reformat Edge" topic in this chapter to configure the Reformat Edge.
   Note that the **Record** field's Delimiter field will be empty, as there is only one column.
7. Use the "Configuring the precedence rules WebServiceClient component" topic in this chapter to configure the **WebServiceClient** component. The major difference is that you will add this text to the **Generate request** field:

```
<config-service:deletePrecedenceRules
  xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

   xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
$xmlString
</config-service:deletePrecedenceRules>
```

8. Save the graph.

After creating the graph and configuring the components, run the graph to delete the precedence rules listed in the input file.



Chapter 9

Adding Key-Value Pairs

This chapter describes how to add key-value pairs to Endeca records.

About key-value pair data

The **Add KVPs** connector can add key-value pair data to MDEX Engine records.

The two main use cases for the **Add KVPs** connector are:

- To ingest source data that is stored in a key-value pair format instead of the more traditional rectangular data model.
- When you do not what the schema is ahead of time.

With either case, you have the option of loading data in rows (with the **Add/Update Records** connector) that will be faster than loading the same data as key-value pairs.

Format of the KVP input file

The metadata schema of the **Add KVPs** connector is fixed and uses a specific ordering.

The first row of the data source input file is the record header row and must use this schema:

```
specKey|specValue|kvpKey|kvpValue|mdexType
```

The meanings of these schema properties are as follows:

Schema property	Meaning
`specKey`	The name of the primary key (record spec) of the record to which the key-value pair will be added.
`specValue`	The value of the record's primary key.
`kvpKey`	The name (key) of the Endeca standard attribute to be added to the record. If the standard attribute does not exist in the MDEX Engine, it is automatically created by DIWS with system default values.
`kvpValue`	The value of the standard attribute to be added to the record.
`mdexType`	Specifies the `mdex` type (such as `mdex:int` or `mdex:dateTime`) for the `kvpKey` standard attribute. This parameter is intended for use when

Schema property	Meaning
	you want to create a new standard attribute and want to specify its property type. If a new PDR for the standard attribute is created and `mdexType` is not specified, then the type of the new standard attribute will be `mdex:string`. If the standard attribute already exists, you can specify an empty value for `mdexType`.

The following is a simple example of an input file for the **Add KVPs** connector:

```
specKey|specValue|kvpKey|kvpValue|mdexType
ProductID|51841|Designation|Professional use|
ProductID|48191|Color|Crimson|
ProductID|48191|Color|Sea Blue|
ProductID|48197|Component|road rim|
ProductID|48197|Location|42.365615 -71.075647|mdex:geocode
```

The example adds a Designation assignment to Record 51841, two Color assignments to Record 48191 (Color is a multi-assign attribute), and a Component assignment to Record 48197. In addition, a new geocode standard attribute named Location is created in the MDEX Engine and added to Record 48197.

Configuring the Reader for the KVP input file

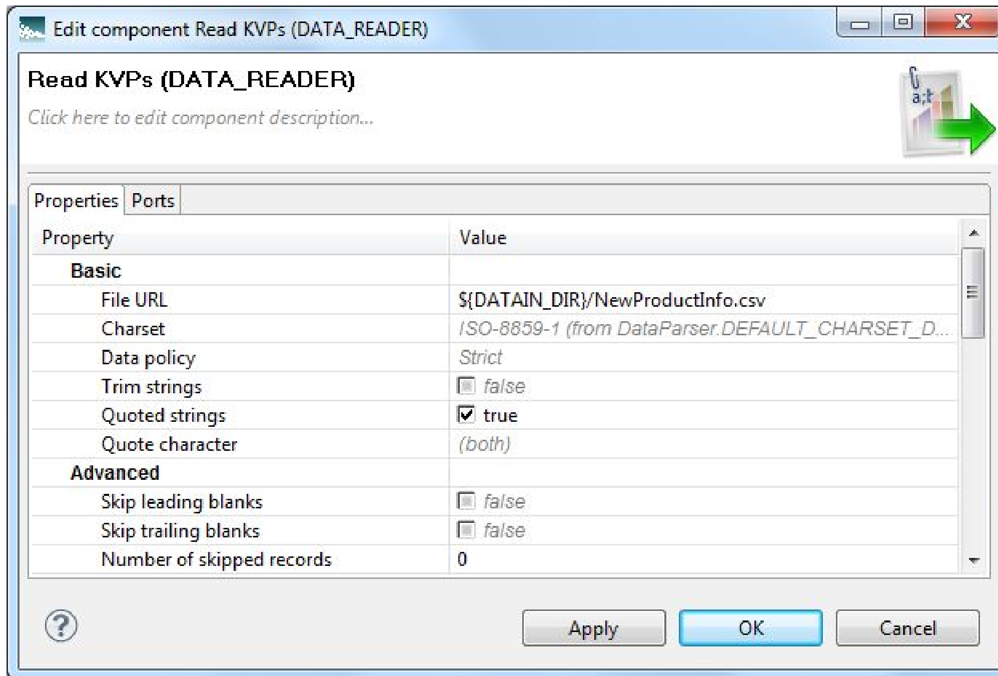
This task describes how to configure the **UniversalDataReader** component to read in the KVP data.

This procedure assumes that you have created a graph for the KVP components and that you have copied the input file (named `NewProductInfo.csv` in our example) into the **data-in** folder in the Navigation pane of the project. The procedure also assumes that you will be using the **UniversalDataReader** component to read in the KVP input data.

To configure the **UniversalDataReader** component for the KVP input file:

1. In the Palette pane, open the **Readers** section and drag the **UniversalDataReader** component into the Graph Editor.
2. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
3. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the KVP input file and click **OK**.
4. Check the **Quoted strings** box so that its value changes to `true`.
5. Leave the **Number of skipped records** field set to 0.
6. Optionally, you can use the **Component name** field to provide a customized name (such as "Read KVPs") for this component.
7. Click **OK** to apply your configuration changes to the **UniversalDataReader** component.
8. Save the graph.

After the component is configured, the Reader Edit Component dialog should look like this example:



Configuring the Add KVPs connector

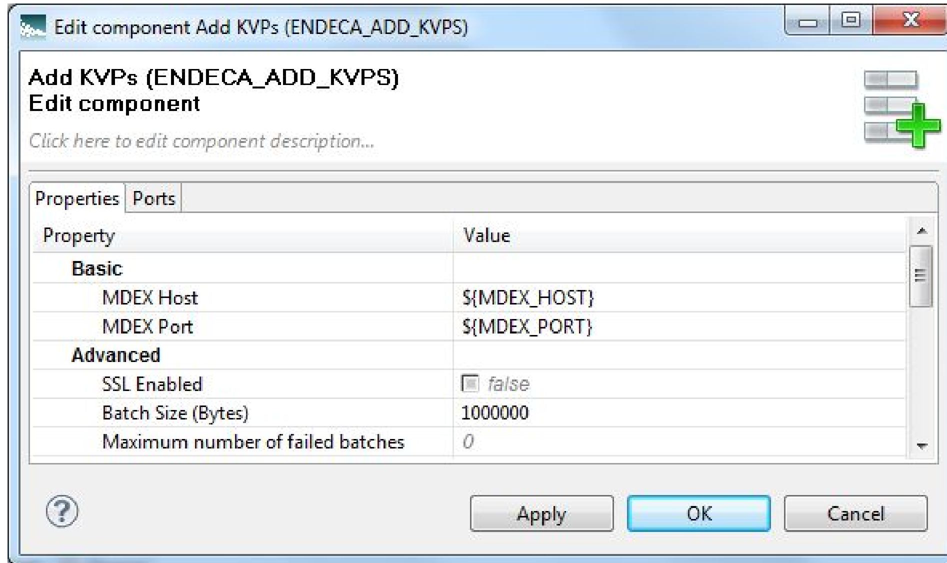
You must configure the **Add KVPs** connector to properly connect to your MDEX Engine.

This procedure assumes that you have created a graph for the **Add KVPs** connector.

To configure the **Add KVPs** connector:

1. In the Palette pane, open the **Latitude** section and drag the **Add KVPs** connector into the Graph Editor.
2. In the Graph window, double-click the **Add KVPs** connector.
   The Writer Edit Component dialog is displayed.
3. In the Writer Edit Component dialog, enter these settings:
   a) **MDEX Host**: The host name of the machine on which the MDEX Engine is running. You can specify **${MDEX_HOST}** if you have the `MDEX_HOST` variable defined in the `workspace.prm` file for your project.
   b) **MDEX Port**: The port on which the MDEX Engine is listening for requests. You can specify **${MDEX_PORT}** if you have the `MDEX_PORT` variable defined in the `workspace.prm` file for your project.
   c) **SSL Enabled**: Toggle this field to `true` if the MDEX Engine is SSL-enabled.
   d) **Batch Size (Bytes)**: To change the default batch size, enter a positive integer. Specifying 0 or a negative number will disable batching.
   e) **Maximum number of failed batches**: Enter a positive integer that sets the maximum number of batches that can fail before the ingest operation is ended. Entering 0 allows no failed batches.
4. When you have input all your changes, click **OK**.
5. Save the graph.

After configuration, the Writer Edit Component dialog should look like this example:



Configuring KVP metadata

The Edge component must be configured with a Metadata definition for loading the key-value pair data.

This procedure assumes that you have created a graph and added a reader component and the **Add KVPs** connector to it. It also assumes that you have added the key-value pair source file to the project's **data-in** folder.

To configure the Metadata definition for the KVP Edge:

1. In the Palette pane, click **Edge** and use it to connect the reader and the **Add KVPs** connector.
2. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
3. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
4. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the key-value pair source file and click **OK**.
5. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.
6. In the middle pane of the Metadata editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.
7. In the Record pane of the Metadata editor, make these changes:
   a) Click the **Record** Name field and change the default value to a name such as `KVPs`.
   b) Make sure that the **Type** field of all properties is set to type **string.** For example if the `specKey` property is set to **integer**, change it to **string**.

c) Verify that all properties have the correct delimiter character set (which is the comma character in our example).

At this point, the pane should look like this example:

#	Name	Type	Delimiter
1	Record: KVPs	delimited	,
1	specKey	string	,
2	specValue	string	,
3	kvpKey	string	,
4	kvpValue	string	,
5	mdexType	string	\r\n

Filter:

d) When you have input all your changes, click **Finish**.

8. Save the graph.

The Metadata definition for the Edge component is now set.

Running the KVPs graph

After creating the graph and configuring the components, you can run the graph to add the key-value pair record assignments to the MDEX Engine.

To run the graph to add key-value pairs to the MDEX Engine:

1. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Add KVPs** connector.
2. Run the graph using one of the run methods.

   For example, you can click the green circle with white triangle icon in the Tool bar:

As the graph runs, the process of the graph execution is listed in the Console Tab. The execution is completed successfully when you see final output similar to this example that adds five key-value pair assignments to the MDEX Engine:

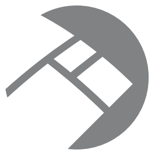
```
INFO  [WatchDog] - Successfully started all nodes in phase!
WARN  [Consumer-0] - Unrecognized assignment type "".  Using "mdex:string"
 instead.
WARN  [Consumer-0] - Unrecognized assignment type "".  Using "mdex:string"
 instead.
WARN  [Consumer-0] - Unrecognized assignment type "".  Using "mdex:string"
 instead.
WARN  [Consumer-0] - Unrecognized assignment type "".  Using "mdex:string"
 instead.
INFO  [WatchDog] - [Clover] Post-execute phase finalization: 0
INFO  [WatchDog] - [Clover] phase: 0 post-execute finalization successfully.
INFO  [WatchDog] - ----------------------** Final tracking Log for phase
[0] **---------------------
INFO  [WatchDog] - Time: 08/06/11 14:30:39
INFO  [WatchDog] - Node                  ID          Port      #Records
      #KB aRec/s   aKB/s
INFO  [WatchDog] - -------------------------------------------------------
--------------------------
INFO  [WatchDog] - UniversalDataReader    DATA_READER0
```

```
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                          Out:0           5
        0      5       0
INFO  [WatchDog] - Add KVPs               ENDECA_ADD_KVPS0
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                           In:0           5
        0      5       0
INFO  [WatchDog] - ----------------------------------** End of Log **------
--------------------------
INFO  [WatchDog] - Execution of phase [0] successfully finished - elapsed
time(sec): 1
INFO  [WatchDog] - -----------------------** Summary of Phases execution
**---------------------
INFO  [WatchDog] - Phase#            Finished Status         RunTime(sec)
   MemoryAllocation(KB)
INFO  [WatchDog] - 0                 FINISHED_OK                         1
             7146
INFO  [WatchDog] - ------------------------------** End of Summary **-----
----------------------
INFO  [WatchDog] - WatchDog thread finished - total execution time: 1 (sec)
INFO  [main] - Freeing graph resources.
INFO  [main] - Execution of graph successful !
```

The example also shows four occurrences of this benign message:

```
Unrecognized assignment type "".  Using "mdex:string" instead.
```

The message is simply informing you that the fifth input schema field (the `mdexType` field) is empty on four of the KVP entries and that the connector will use the `mdex:string` property type when ingesting the data.



Chapter 10

Loading Taxonomies

This chapter describes how to load an externally managed taxonomy (EMT) into the MDEX Engine.

Overview of loading a taxonomy

This chapter will walk you through the various tasks in creating a graph that can load a taxonomy into the MDEX Engine.

The **Add Managed Values** connector allows you to load an externally managed taxonomy (EMT) into the MDEX Engine. When loaded, externally managed taxonomies are added as managed values to a managed attribute. You must create a graph and add the **Add Managed Values** connector and a reader component (such as the **UniversalDataReader** component) to the graph.

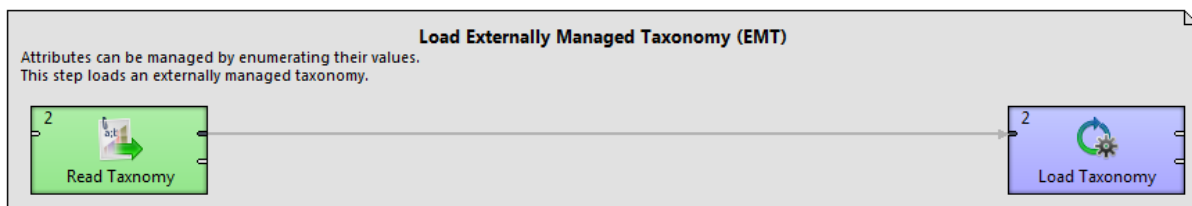
Keep the following two items in mind when adding a taxonomy:

- Managed values can be added to only one managed attribute in a taxonomy load operation. That is, you can specify the name of only one managed attribute in the **Add Managed Values** connector. This means that all the managed values in the taxonomy input file will be added to the same managed attribute.
- If the managed attribute (to which the taxonomy is being added) does not exist in the MDEX Engine, it will be created automatically by the Data Ingest Web Service. That is, the appropriate PDR and DDR for the managed attribute will be created with system default values. For these default values, see Chapter 1 in this guide.

For the procedure documented in this chapter, the definitions of the managed values to be added are in a flat file. However, the definitions can use other formats that are supported by the LDI reader components. The format of the source data is explained in a following topic.

Sample taxonomy graph

In the Latitude Sample Application, the LoadIndexingConfiguration graph has a step (named Load Externally Managed Taxonomy) that is used as the example for the load-taxonomy procedure:



The graph reads in a CSV file (named `ProductCategoryTaxonomy.csv`) and uses an **Add Managed Values** connector to load the data into the MDEX Engine.

Format of the taxonomy input file

The input must contain four mandatory configuration properties and a corresponding set of managed value data.

The first line of a taxonomy input file must have these managed value header properties, and in this order:

```
MvalSpec|Displayname|ParentKey|Synonym
```

The actual names of the header properties in your input file can be different from the names used here (for example, you can use `CategoryKey` instead of `MvalSpec`). However, the order (positions) of the header properties and their values is crucial. For example, the third position signifies the managed value's parent ID, regardless of the name used for that header property.

The meanings of these header properties are as follows:

Property	Purpose
`MvalSpec`	A unique string identifier for the managed value. This is the managed value spec.
`Displayname`	The name for the managed value.
`ParentKey`	Specifies the parent ID for this managed value, If this is a root managed value, use a forward slash (/) as the ID. If this is a child managed value, specify the unique ID of the parent managed value.
`Synonym`	Optionally defines the name of a synonym. You can add synonyms to a managed value so that users can search for other text strings and still get the same records as a search for the original managed value name. Synonyms can be added to both root and child managed values. If you add multiple synonyms for a managed value, the synonyms are separated by a delimiter that you specify in the configuration of the **Add Managed Values** connector.

After the header row, the second and following rows in the input file contain managed value data for the managed value properties.

The following image shows the beginning lines of the `ProductCategoryTaxonomy.csv` input file for the taxonomy graph:

	A	B	C	D
1	CategoryKey	CategoryDisplayName	ParentKey	Synonym
2	CAT_BIKES	Bikes	/	
3	CAT_COMPONENTS	Components	/	
4	CAT_CLOTHING	Clothing	/	
5	CAT_ACCESSORIES	Accessories	/	
6	1	Mountain Bikes	CAT_BIKES	
7	2	Road Bikes	CAT_BIKES	
8	3	Touring Bikes	CAT_BIKES	
9	4	Handlebars	CAT_COMPONENTS	
10	5	Bottom Brackets	CAT_COMPONENTS	
11	6	Brakes	CAT_COMPONENTS	
12	7	Chains	CAT_COMPONENTS	

ProductCategoryTaxonomy

Ready 100%

In this example:

- Four root managed values are created. Their managed value specs are CAT_BIKES, CAT_COMPONENTS, CAT_CLOTHING, and CAT_ACCESSORIES and they all have a `ParentKey` of a forward slash (/) because they are root managed values. Their `CategoryDisplayName` values set the names that will be displayed in the application UI.
- Seven child managed values are created. Three are children of the CAT_BIKES managed value and the other four are children of the CAT_COMPONENTS managed value.

Note that more child managed values are created from the `ProductCategoryTaxonomy.csv` specifications. The file is stored in the **config-in** folder of the project.

Creating a graph for the taxonomy

This task describes how to create an empty graph for loading a taxonomy.

The only prerequisite for this task is that you must have created a Data Integrator Designer project. Keep in mind that a project can have multiple graphs, which means that you can create this graph in an existing project.

Note: In the Latitude Sample Application, the taxonomy loader is a step in the LoadIndexingConfiguration graph. However, to simplify this description of loading taxonomies, we will create a taxonomy-only graph.

To create an empty graph for your taxonomy:

1. In the Navigator pane, right-click the **graph** folder.
2. Select **New** > **ETL Graph**.
   The **Create new graph** dialog is displayed.
3. In the **Create new graph** dialog:
   a) Type in the name of the graph, such as **LoadTaxonomy**.
   b) Optionally, type in a description.
   c) You can leave the **Allow inclusion of parameters from external file** box checked.
   d) Click **Next** when you finish.
4. In the **Output** dialog, click **Finish**.

Adding components to the taxonomy graph

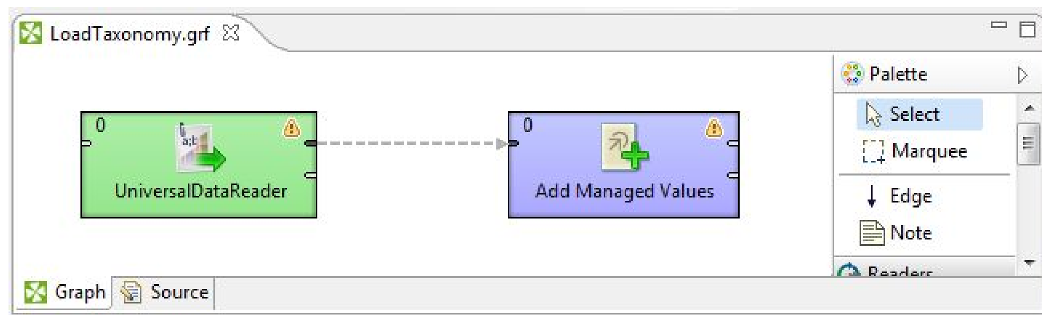
The process requires that you add the **UniversalDataReader** component and the **Add Managed Values** connector to the graph.

In addition, an Edge component will be added to connect the two components.

To add components to the graph:

1. In the Palette pane, open the **Readers** section and drag the **UniversalDataReader** component into the Graph Editor.
2. In the Palette pane, open the **Latitude** section and drag the **Add Managed Values** connector into the Graph Editor.
3. In the Palette pane, click **Edge** and use it to connect the two components.
4. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the two connected components should look like this:



The next tasks are to configure these components.

Configuring the Reader for the taxonomy input file

This task describes how to configure the **UniversalDataReader** component to read in the taxonomy data.

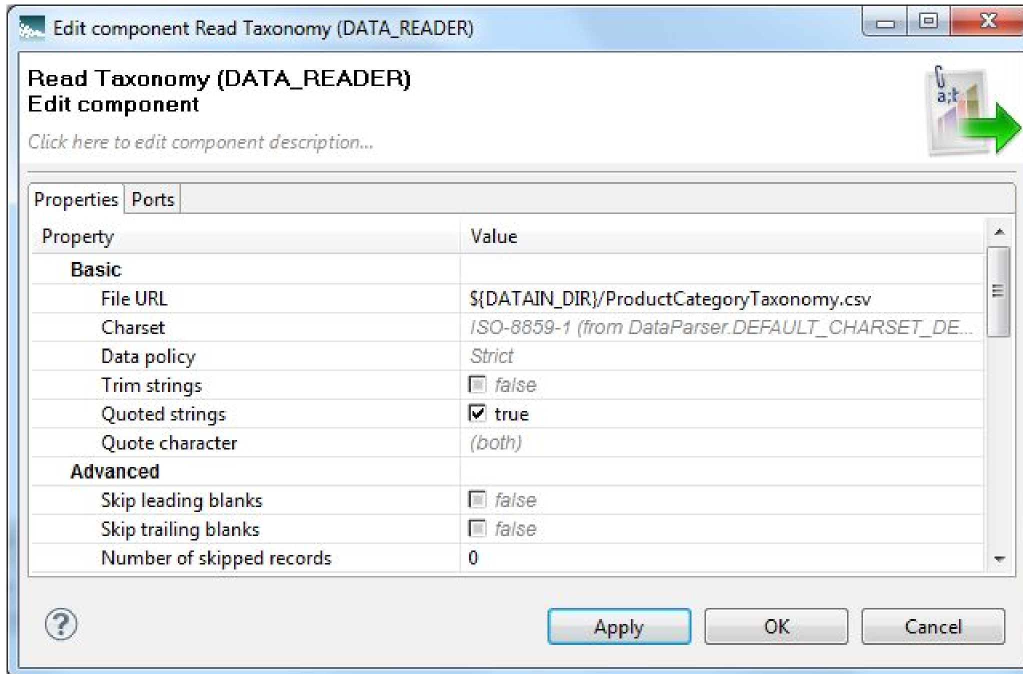
This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the taxonomy source file to the project's **config-in** folder (or alternatively, to the **data-in** folder).

To configure the **UniversalDataReader** component for the taxonomy input file:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **config-in** folder.
   d) Select the taxonomy input file and click **OK**.
3. Check the **Quoted strings** box so that its value changes to `true`.
4. Leave the **Number of skipped records** field to its default of 0.

5. Optionally, you can use the **Component name** field to provide a customized name (such as "Read Taxonomy") for this component.
6. Click **OK** to apply your configuration changes to the **UniversalDataReader** component.
7. Save the graph.

After the component is configured, the Reader Edit Component dialog should look like this example:



The next task is to configure the **Add Managed Values** connector.

Configuring the Add Managed Values connector

You must configure the **Add Managed Values** component with the location and port of the MDEX Engine, as well as the managed attribute name.

This procedure assumes that you have created a graph and added the **Add Managed Values** connector.

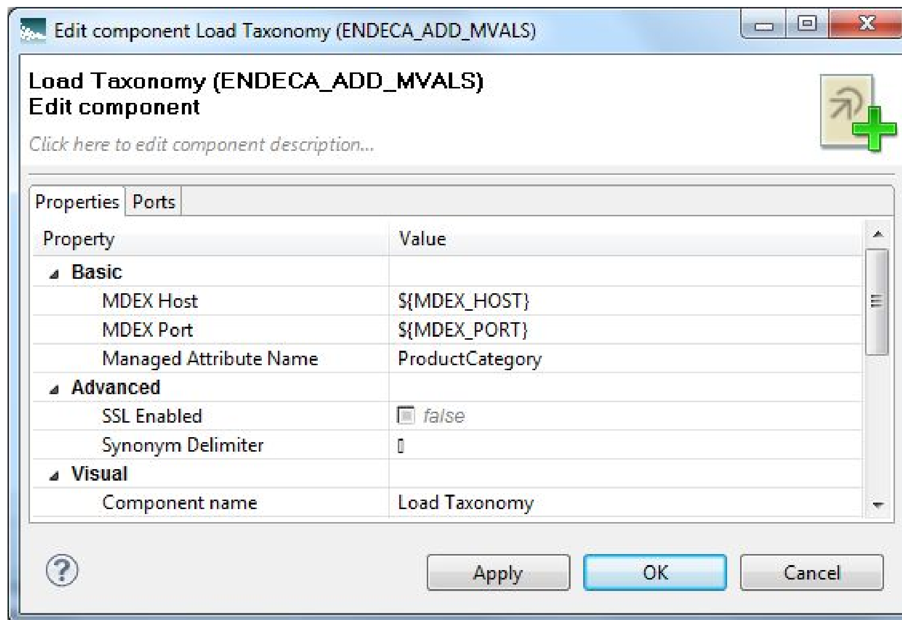
To configure the **Add Managed Values** connector:

1. In the Graph window, double-click the **Add Managed Values** connector.
   The Edit Component dialog is displayed.
2. In the Writer Edit Component dialog, enter these settings:
   a) **MDEX Host**: The host name of the machine on which the MDEX Engine is running. You can specify **${MDEX_HOST}** if you have the `MDEX_HOST` variable defined in the `workspace.prm` file for your project.
   b) **MDEX Port**: The port on which the MDEX Engine is listening for requests. You can specify **${MDEX_PORT}** if you have the `MDEX_PORT` variable defined in the `workspace.prm` file for your project.
   c) **Managed Attribute Name**: The name of the dimension to which the dimension values will be added.
   d) **SSL Enabled**: Toggle this field to `true` if the MDEX Engine is SSL-enabled.

e) **Synonym Delimiter**: Optionally, you can specify the character that separates multiple synonyms for a managed value. Keep in mind that this delimiter is different from the delimiter that separates the property fields.
f) Optionally, you can use the **Component name** field to provide a customized name (such as "Load Taxonomy") for this component.

3. When you have input all your changes, click **OK**.
4. Save the graph.

After configuration, the Edit Component dialog should look like this example:



In this sample **Add Managed Values** connector, the managed values will be added to the ProductCategory managed attribute.

Configuring taxonomy metadata

The Edge component must be configured with a Metadata definition for loading the taxonomy.

The prerequisite for this task is that an Edge component must exist in the graph.

To configure the Metadata definition for the taxonomy Edge:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the taxonomy source file and click **OK**.
4. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.
   The taxonomy data is loaded into the Metadata Editor.

5. In the middle pane of the Metadata Editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.
6. In the upper pane of the Metadata Editor:
   a) Click the **Record** Name field and change the default value to a name such as `Taxonomy`.
   b) Make sure that the **Type** field of all the properties is set to type **string**.
   c) Verify that all properties have the correct delimiter character set (which is the comma in our example).
   d) When you have input all your changes, click **Finish**.
7. Save the graph.

The Metadata definition for the Edge component is now set.

Running the taxonomy graph

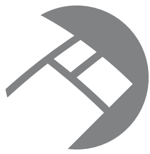
Endeca recommends that you use a transaction graph to run the load taxonomy graph.

To run the transaction graph to load a taxonomy:

1. Create a transaction graph as described in Chapter 3 (titled "Working with Transaction Graphs").
2. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Add Managed Values** connector.
3. Run the transaction graph using one of the run methods.

   For example, you can click the green circle with white triangle icon in the Tool bar:

As the graph runs, the process of the graph execution is listed in the Console Tab. The output lists the number of records that were read in by the **UniversalDataReader** component and the number of records that were sent to the MDEX Engine by the **Add Managed Values** connector.



Chapter 11

Importing and Exporting the Configuration

This chapter discusses how to import and export the MDEX Engine configuration and schema using the Latitude connectors.

About importing and exporting

Use the **Export Config** and **Import Config** connectors in the LDI to export and import the schema and configuration.

The configuration of your index and the schema for your records are created once you initially load the data, the schema, and the configuration into the MDEX Engine.

Use cases for exporting and importing schema and configuration

You may need to import and export your schema and configuration in several typical scenarios:

- As part of the baseline update process, when only updates to the data are required but the MDEX Engine index configuration and the schema must remain the same.
- If you have an implementation running in the development environment, it typically should match the implementation running in the production environment in terms of index configuration and the schema for your records. (Although the development application may contain a subset of data). You can use export and import connectors for sharing the configuration and schema between these environments.

What is being exported and imported

The following aspects of your configuration and schema are being exported and imported when you use the **Export Config** or **Import Config** connectors in the LDI:

- The schema for your records. The schema is represented by PDRs and DDRs that describe the behavior of attributes on your records, such as whether they are searchable, or have hierarchy.
- The configuration, which includes:
  - The indexed configuration — the XML configuration documents, such as documents describing your record search configuration and search interfaces, or thesaurus configuration.
  - All additional configuration information, such as display names or attribute groups, the configuration captured in the GCR, and precedence rules.

Note: The export and import connectors do not export or import the file that stores all word forms used for stemming dictionaries in the MDEX Engine. This file is created automatically

when you provision the MDEX Engine, and is typically not modified. In rare cases when you may need to make changes to this file, use a custom component in LDI that can export and re-import this file using the Configuration Web Service operations.

Memory considerations for the configuration graphs

Graphs that manipulate large amounts of data may result in `BufferOverflow` errors when running the graph. Exporting and importing a MDEX Engine configuration typically requires more memory than is allocated in the default LDI `defaultProperties` configuration file.

You can use these settings for your configuration graphs:

- `Record.MAX_RECORD_SIZE` = 524188
- `DataParser.FIELD_BUFFER_LENGTH` = 1048376
- `DataFormatter.FIELD_BUFFER_LENGTH` = 262144
- `DEFAULT_INTERNAL_IO_BUFFER_SIZE` = 524288

You will have to further increase these settings if they are not high enough for your configuration needs.

For details on increasing memory allocation, see the "BufferOverflow errors" topic in the "Troubleshooting Problems" chapter in this guide.

Exporting the configuration

You can export the configuration from the MDEX Engine by using the **Export Config** connector.

Adding components to the export graph

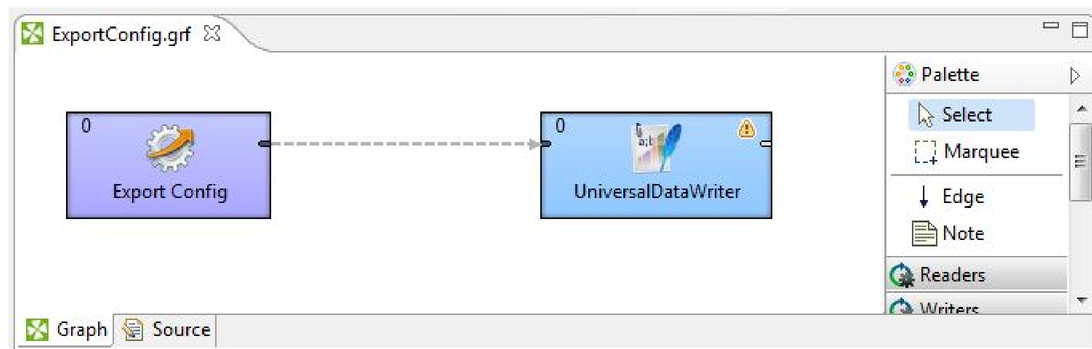
This topic describes the LDI components that must be added to the export graph.

This procedure assumes that you have created an empty graph (our example is named ExportConfig).

To add components to the graph that exports the configuration from an MDEX Engine:

1. In the Palette pane, drag the **Export Config** component from the **Latitude** section.
2. In the Palette pane, drag the **UniversalDataWriter** component from the **Writers** section.
3. In the Palette pane, click **Edge** and use it to connect the components.
4. Save the graph.

At this point, the Graph Editor with the connected components should look like this:

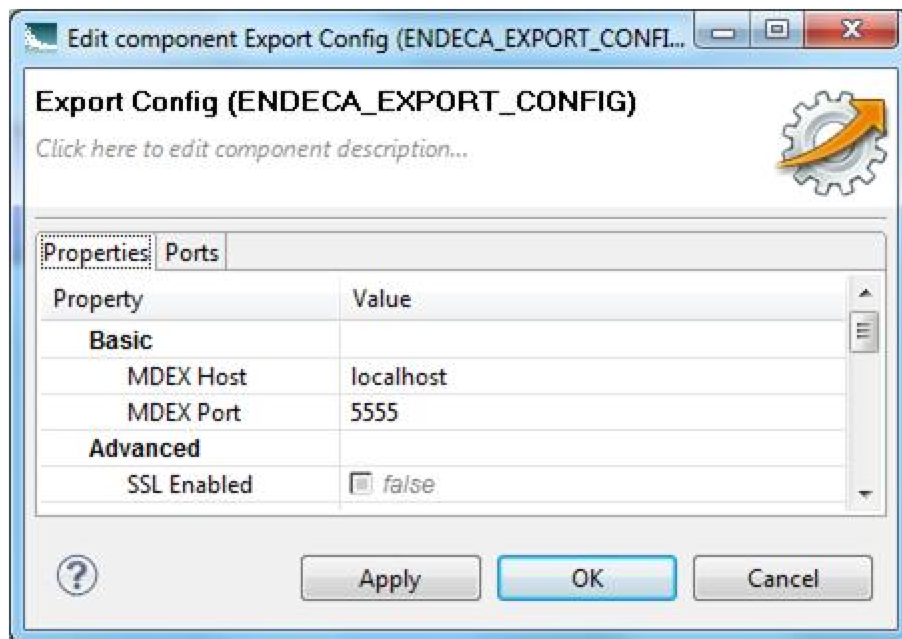


Configuring the Export Config connector

You must configure the **Export Config** connector with the location and port of the MDEX Engine.

To configure the **Export Config** connector:

1. In the Graph window, double-click the **Export Config** component.
   The Edit Component dialog is displayed.



2. In the Writer Edit Component dialog, enter these mandatory settings in the Basic section:
   a) **MDEX Host**: Enter the host name of the machine on which the MDEX Engine is running.
   b) **MDEX Port**: Enter the port on which the MDEX Engine is listening for requests.
3. Still in the Writer Edit Component dialog, you should toggle the **SSL Enabled** field to `true` if the MDEX Engine is SSL-enabled.
4. When you have input all your changes, click **OK**.
5. Save the graph.

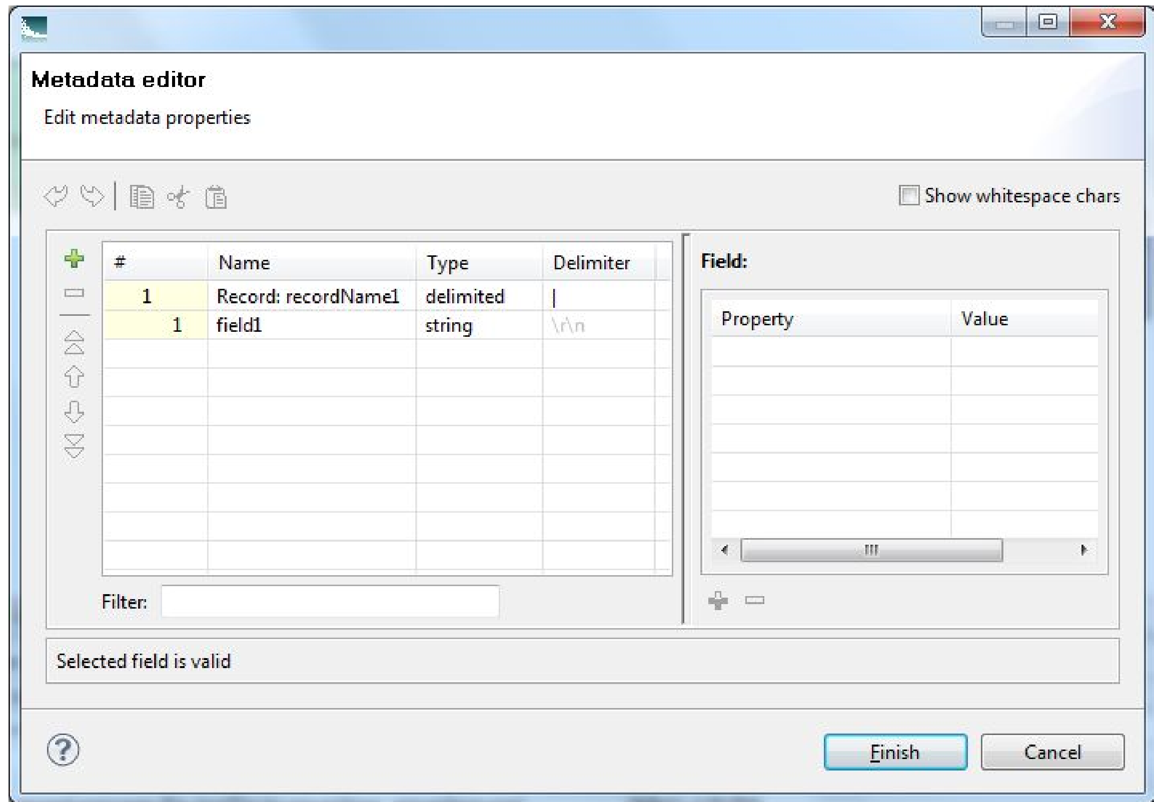
Configuring the Edge in the export graph

This topic describes how to configure the metadata for the Export Config Edge.

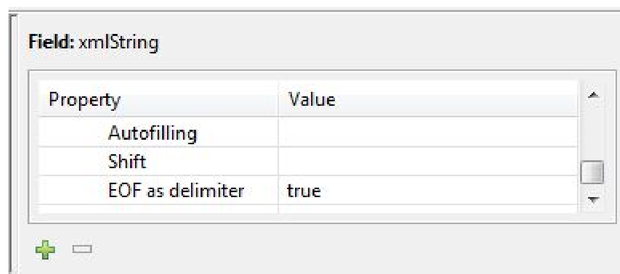
The metadata must be configured to have only one string field and no record delimiter. Therefore, the metadata of the Edge must be manually modified to remove the record and field delimiters from the metadata. This will leave the **EOF as delimiter** property as the sole delimiter.

To configure the Edge Metadata definition for exporting the configuration from the MDEX Engine:

1. Right-click on the Edge and select **New metadata** > **User defined**.
   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a more descriptive name (such as Export).
   b) Leave the **Type** field as `delimited`.
   c) Leave the **Delimiter** field as-is for now. (You will delete it in Step 5.)
3. In the Record pane, make these changes to the **field1** property:
   a) Change the **field1** default name to **xmlString**.
   b) Leave the **Type** field set to **String**.
   c) In the Field Details pane, set the **EOF as delimiter** property to `true`, as in this example:



4. When you have input all your changes in the Metadata Editor, click **Finish**.
5. Now you must manually remove the record and field delimiters from the metadata:
   a) In the Graph Editor, click the **Source** icon (which is next to the **Graph** icon).
   b) In the Record element (which is a child of the Metadata element), find the **fieldDelimiter** and **recordDelimiter** attributes, as shown in this example:

```
<Metadata id="Metadata0">
<Record fieldDelimiter="|" name="Export" recordDelimiter="\r\n"
```

```
type="delimited">
<Field eofAsDelimiter="true" name="xmlString" type="string"/>
</Record>
</Metadata>
```

c) Delete the **fieldDelimiter** and **recordDelimiter** attributes, so that the Record element now looks like this:

```
<Metadata id="Metadata0">
<Record name="Export" type="delimited">
<Field eofAsDelimiter="true" name="xmlString" type="string"/>
</Record>
</Metadata>
```

d) While still within the Source view, right-click and select **Save** to save the graph.

6. Click the **Graph** icon to return to the Graph Editor.

Configuring the UniversalDataWriter component

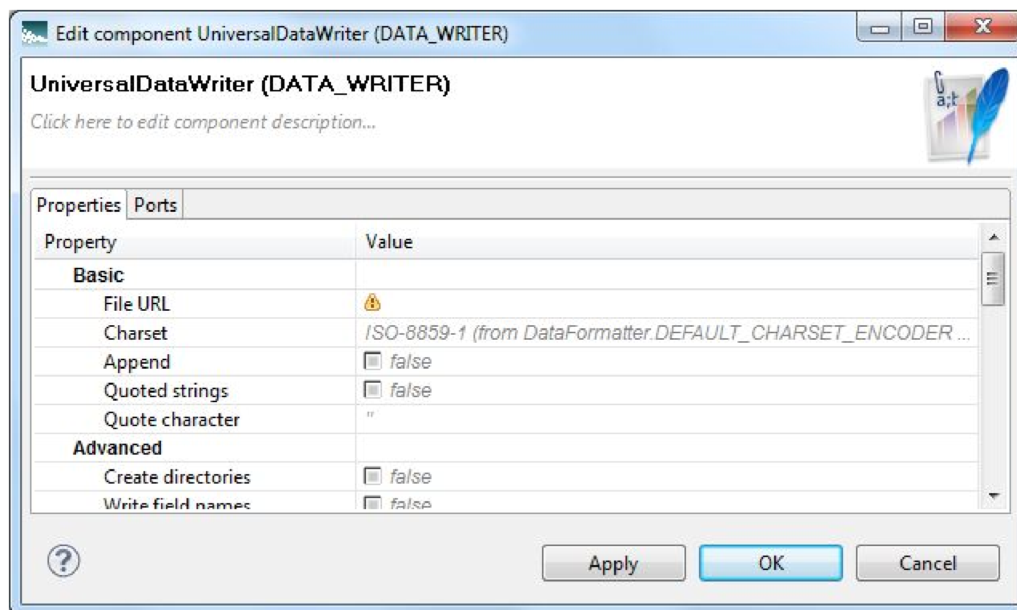
This task describes how to configure the **UniversalDataWriter** component to write out the configuration file.

You can configure the **UniversalDataWriter** to write out the exported configuration to file within the project or externally. (See the *Latitude Data Integrator Designer Guide* for more information on this component.) This procedure assumes that you are writing the output to a text file named `config.out` which is located in the project's **data-out** folder.

To configure the **UniversalDataReader** component to read the semantic entities configuration input file:

1. In the Graph Editor, double-click the **UniversalDataWriter** component to bring up the Reader Edit Component dialog.

   The Writer Edit Component dialog is displayed.



2. For the **File URL** property:

a) Click inside its Value field, which displays a **...** browse button.
b) Click the browse button, which brings up the **URL Dialog** screen.

3. In the **URL Dialog**:
   a) Click the **Workspace view** tab and then double-click the **data-out** folder.
   b) Select the `config.out` file and click **OK**.
4. In the Writer Edit Component dialog, click **OK**.
5. Save the graph.

Instead of running this graph directly, it is recommended that you create a transaction graph (with a **Transaction RunGraph** connector) with this ExportConfig graph as its child graph, and then run the transaction graph.

Importing the configuration

You can import the configuration to the MDEX Engine using the Import Config connector.

Adding components to the import graph

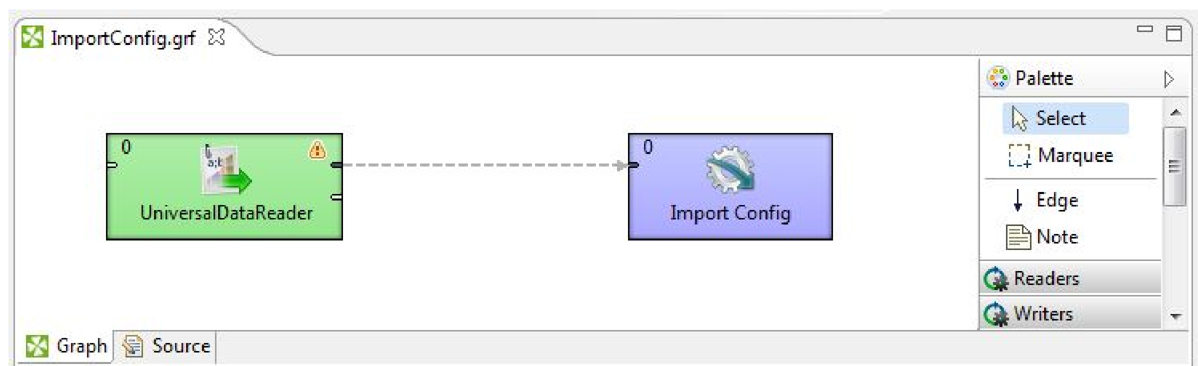
This topic describes the LDI components that must be added to the import graph.

This procedure assumes that you have created an empty graph (our example is named ImportConfig).

To add components to the graph that imports the configuration into a running MDEX Engine:

1. In the Palette pane, drag the **UniversalDataReader** component from the **Readers** section.
2. In the Palette pane, drag the **Import Config** component from the **Latitude** section.
3. In the Palette pane, click **Edge** and use it to connect the components.
4. Save the graph.

At this point, the Graph Editor with the connected components should look like this:



Configuring the Reader in the import graph

This task describes how to configure the **UniversalDataReader** component to read the configuration file.

This procedure assumes that the MDEX Engine's configuration was written to a text file named `config.out` which is located in the project's **data-out** folder. This reader will use that file as its input file.

To configure the Reader component in the import graph:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button, which brings up the **URL Dialog** screen.
3. In the **URL Dialog**:
   a) Click the **Workspace view** tab and then double-click the **data-out** folder.
   b) Select the `config.out` file and click **OK**.
4. In the Reader Edit Component dialog, click **OK**.
5. Save the graph.

Configuring the Edge in the import graph

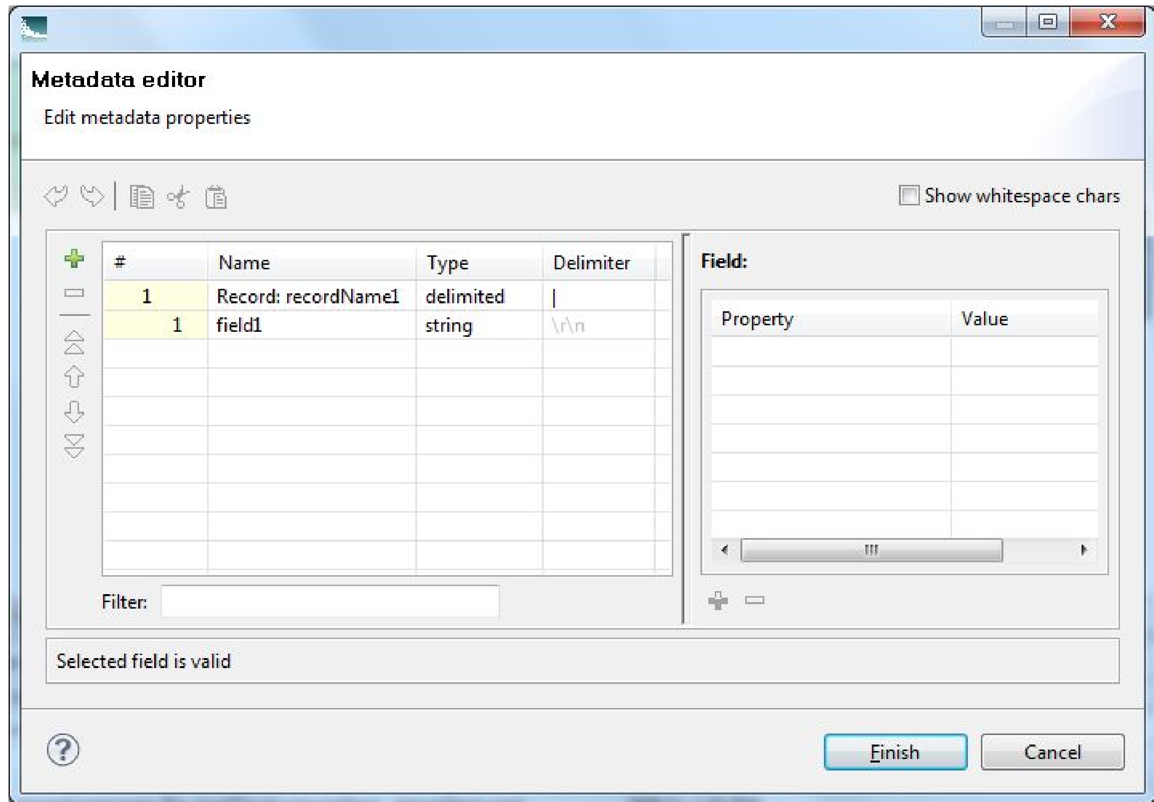
This topic describes how to configure the metadata for the Reader Edge.

The configuration of the Edge in the import graph is similar to that of the export graph. That is, the Edge metadata must be configured to have only one string field and no record delimiter. Therefore, the metadata of the Edge must be manually modified to remove the record and field delimiters from the metadata. This will leave the **EOF as delimiter** property as the sole delimiter.

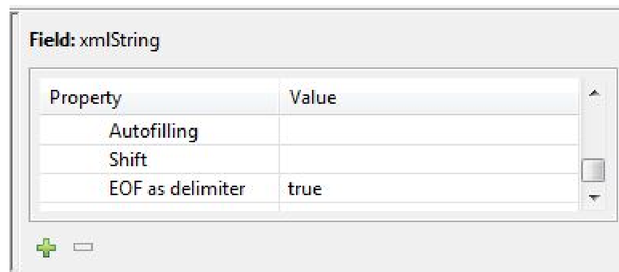
To configure the Edge Metadata definition for importing the configuration from a disk file to the MDEX Engine:

1. Right-click on the Edge and select **New metadata** > **User defined**.

   The Metadata editor is displayed with one default field.



2. In the **Record:recordName1** field:
   a) Change the **recordName1** default value to a more descriptive name (such as Import).
   b) Leave the **Type** field as `delimited`.
   c) Leave the **Delimiter** field as-is for now. (You will delete it in Step 5.)
3. In the Record pane, make these changes to the **field1** property:
   a) Change the **field1** default name to **xmlString**.
   b) Leave the **Type** field set to **String**.
   c) In the Field Details pane, set the **EOF as delimiter** property to `true`, as in this example:



4. When you have input all your changes in the Metadata Editor, click **Finish**.
5. Now you must manually remove the record and field delimiters from the metadata:
   a) In the Graph Editor, click the **Source** icon (which is next to the **Graph** icon).

b) In the Record element (which is a child of the Metadata element), find the **fieldDelimiter** and **recordDelimiter** attributes, as shown in this example:

```
<Metadata id="Metadata0">
<Record fieldDelimiter="|" name="Import" recordDelimiter="\r\n"
type="delimited">
<Field eofAsDelimiter="true" name="xmlString" type="string"/>
</Record>
</Metadata>
```

c) Delete the **fieldDelimiter** and **recordDelimiter** attributes, so that the Record element now looks like this:

```
<Metadata id="Metadata0">
<Record name="Import" type="delimited">
<Field eofAsDelimiter="true" name="xmlString" type="string"/>
</Record>
</Metadata>
```

d) While still within the Source view, right-click and select **Save** to save the graph.

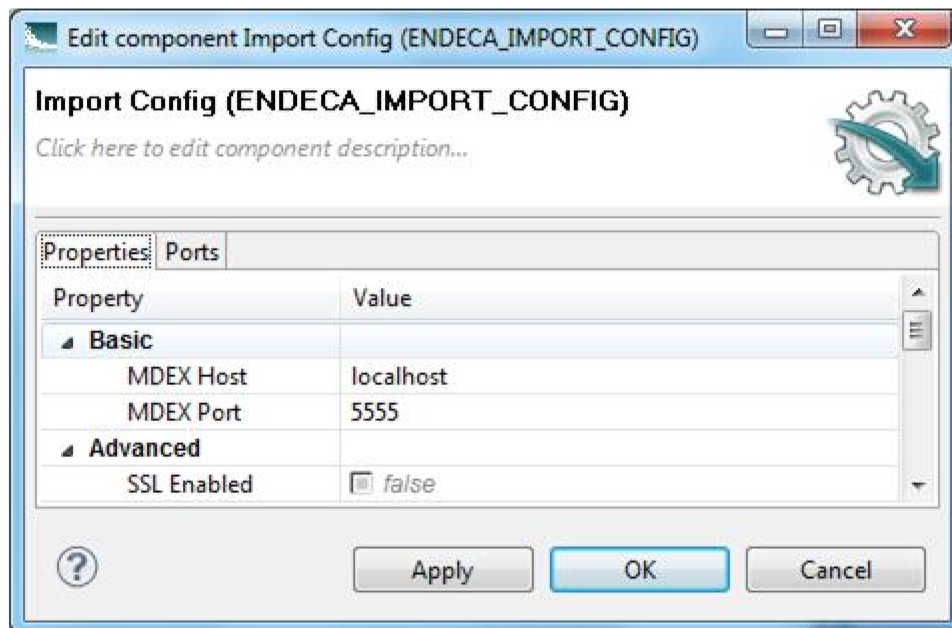
6. Click the **Graph** icon to return to the Graph Editor.

Configuring the Import Config connector

You must configure the **Import Config** connector with the location and port of the MDEX Engine.

To configure the **Import Config** connector:

1. In the Graph window, double-click the **Import Config** component.
   The Edit Component dialog is displayed.



2. In the Writer Edit Component dialog, enter these mandatory settings in the Basic section:
   a) **MDEX Host**: Enter the host name of the machine on which the MDEX Engine is running.
   b) **MDEX Port**: Enter the port on which the MDEX Engine is listening for requests.

3. Still in the Writer Edit Component dialog, you should toggle the **SSL Enabled** field to `true` if the MDEX Engine is SSL-enabled.
4. When you have input all your changes, click **OK**.
5. Save the graph.

Instead of running this graph directly, it is recommended that you create a transaction graph (with a **Transaction RunGraph** connector) with this ImportConfig graph as its child graph, and then run the transaction graph.

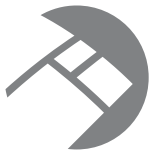
Running the configuration graphs with a transaction graph

You should run the export and import configuration graphs with a transaction graph.

A transaction graph uses a **Transaction RunGraph** connector to safely run one or more graphs within the transaction environment of the MDEX Engine. This connector can start an outer transaction, run the set of graphs so that they succeed or fail as a unit, and finally commit the transaction (or roll it back upon failure). It is therefore recommended that instead of running each of the configuration graphs in standalone mode, you instead build two transaction graphs (one for each configuration graph).

To use a transaction graph to run a configuration graph:

1. Create a transaction graph that runs the ExportConfig graph.
   The procedure is described in the chapter titled "Working with Transaction Graphs".
2. Run the transaction graph as you would run any other graph.
3. Repeat the procedure to create a second transaction graph that runs the ImportConfig graph.



Chapter 12

Deleting Data

This chapter describes how to delete records from the MDEX Engine data set. It also describes how to key/value pairs from individual records.

Format of the delete input file

The format of the delete input file uses a fixed schema and a specific ordering of the input fields.

The **Delete Data** connector can perform the following deletions of data in the MDEX Engine:

- Delete a full record.
- Delete a specific key/value pair from a record. All other key/value pairs on the record are not affected.
- Delete all key/value pairs (from the same standard attribute) from a record. This is a wildcard delete of the values from a specific standard attribute on the record. All other key/value pairs (on the record) from other standard attributes are not affected.

You can specify all three types of delete operations in the same input file.

The two restrictions of this connector are:

- It cannot delete managed values on the record.
- When deleting records, it cannot do wildcard deletes (for example, delete Records 50*) and it cannot delete ranges of records (for example, delete Records 5000 to 5100). You must specify each record explicitly by its primary key.

The format of the input file is fixed and uses a specific ordering:

- The first row of the input file is the record header row and must use a fixed schema.
- The second and following lines specify information about the records and/or record data to be deleted.

The schema of the record header row is:

```
specKey|specValue|kvpKey|kvpValue
```

where:

- *specKey* is the primary key (record spec) of the record.
- *specValue* is the primary key value.
- *kvpKey* is the name (key) of the Endeca standard attribute to which the assignment belongs. If both *kvpKey* and *kvpValue* are blank, the entire record is deleted.

- *kvpValue* is the assigned value to be removed. If this field is blank but *kvpKey* is not, then all assignments of *kvpKey* are deleted.

An example of a text input file for the **Delete Data** connector is:

```
specKey|specValue|kvpKey|kvpValue
ProductID|3000|Colors|green
ProductID|4000|Handling|
ProductID|5000||
```

When the connector is run with this input file:

- The assignment "green" from the Colors standard attribute is removed from Record 3000.
- All assignments from the Handling standard attribute are removed from Record 4000.
- Record 5000 is deleted from the MDEX Engine.

After creating the input file, you should add it to the project's **data-in** folder.

Adding components to the delete data graph

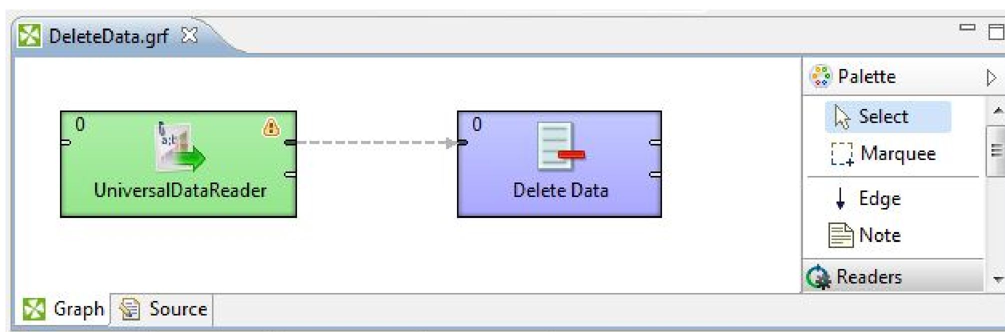
Building a graph to delete data requires that you add the **Delete Data** connector to the graph.

This procedure assumes that you have added the delete input file to the **data-in** folder of the project and have also created an empty graph.

To add components to a graph for deleting records:

1. In the Palette pane, open the **Readers** section and drag the **UniversalDataReader** component into the Graph Editor.
2. In the Palette pane, open the **Latitude** section and drag the **Delete Data** connector into the Graph Editor.
3. In the Palette pane, click **Edge** and use it to connect the two components.
4. From the File menu, click **Save** to save the graph.

At this point, the Graph Editor with the two connected components should look like this:



The next tasks are to configure the components.

Configuring the Reader for the delete input file

This task describes how to configure the **UniversalDataReader** component to read in the file that specifies what record data to delete.

This procedure assumes that you have created a graph and added the **UniversalDataReader** component. It also assumes that you have added the delete input file to the project's **data-in** folder.

To configure the **UniversalDataReader** component for the data delete input file:

1. In the Graph Editor, double-click the **UniversalDataReader** component to bring up the Reader Edit Component dialog.
2. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the data delete input file and click **OK**.
3. Check the **Quoted strings** box so that its value changes to `true`.
4. Leave the **Number of skipped records** field set to the default of 0.
5. Click **OK** to apply your configuration changes to the **UniversalDataReader** component.
6. Save the graph.

Configuring the metadata for data deletes

The Edge component must be configured with a Metadata definition.

The prerequisite for this task is that an Edge component must exist in the graph.

To configure the Metadata definition for the data delete Edge:

1. Right-click on the Edge and select **New metadata** > **Extract from flat file**.
   The Flat File dialog is displayed.
2. In the Flat File dialog, click the **Browse** button, which brings up the **URL Dialog**.
3. For the **File URL** property:
   a) Click inside its Value field, which displays a **...** browse button.
   b) Click the browse button.
   c) Click the **Workspace view** tab and then double-click the **data-in** folder.
   d) Select the data delete input file and click **OK**.
4. In the Flat File dialog, make sure that the **Record type** field is set to **Delimited** and then click **Next**.
5. In the middle pane of the Metadata Editor:
   a) Check the **Extract names** box.
   b) Click **Reparse**.
   c) Click **Yes** in the Warning message.
6. In the upper pane of the Metadata Editor:
   a) Click the **Record:recordName1** Name field and change the **recordName1** default value to a name such as `DeleteRecs`.
   b) Make sure that the **Type** field of ***all*** properties is set to type **string**. For example if the `specKey` property is set to **integer**, change it to **string**.
   c) Verify that all properties have the correct delimiter character set (which is the pipe character in our example). The final property should have a new-line as the delimiter (\n on Linux and \r\n on Windows).

   At this point, the pane should look like this example:

<table>
<tr><th>#</th><th>Name</th><th>Type</th><th>Delimiter</th></tr>
<tr><td>1</td><td>Record: DeleteRecs</td><td>delimited</td><td>|</td></tr>
<tr><td>1</td><td>specKey</td><td>string</td><td>|</td></tr>
<tr><td>2</td><td>specValue</td><td>string</td><td>|</td></tr>
<tr><td>3</td><td>kvpKey</td><td>string</td><td>|</td></tr>
<tr><td>4</td><td>kvpValue</td><td>string</td><td>\r\n</td></tr>
</table>

Filter:

d) When you have input all your changes, click **Finish**.

7. Save the graph.

The Metadata definition for the Edge component is now set.

Configuring the Delete Data connector

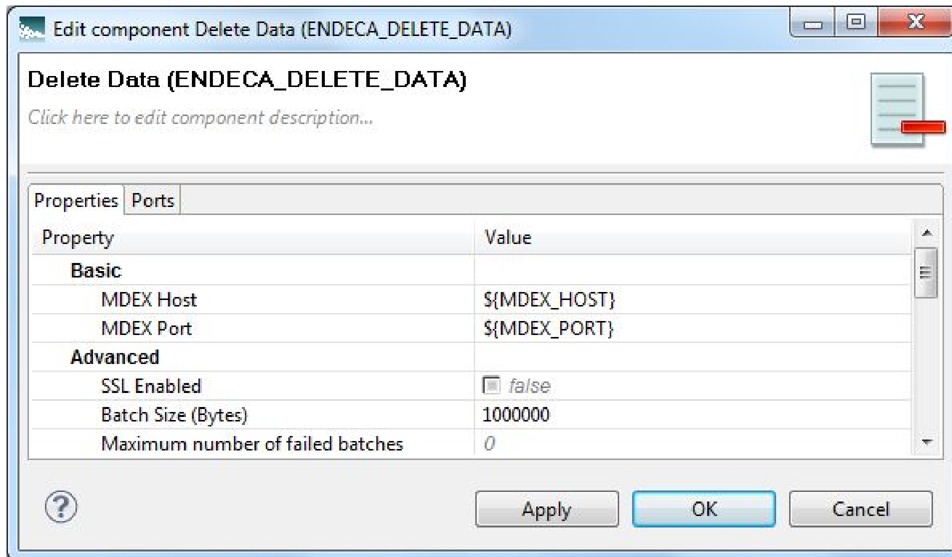
You must configure the **Delete Data** component with the location and port of the MDEX Engine.

This procedure assumes that you have created a graph and added the **Delete Data** connector.

To configure the **Delete Data** connector:

1. In the Graph window, double-click the **Delete Data** component.
   The Writer Edit Component dialog is displayed.
2. In the Writer Edit Component dialog, enter these settings:
   a) **MDEX Host**: The host name of the machine on which the MDEX Engine is running. You can specify **${MDEX_HOST}** if you have the `MDEX_HOST` variable defined in the `workspace.prm` file for your project.
   b) **MDEX Port**: The port on which the MDEX Engine is listening for requests. You can specify **${MDEX_PORT}** if you have the `MDEX_PORT` variable defined in the `workspace.prm` file for your project.
   c) **SSL Enabled**: Toggle this field to `true` if the MDEX Engine is SSL-enabled.
   d) **Batch Size (Bytes)** : Enter an integer greater than 0 to set the batch size in bytes. Specifying 0 or a negative number will disable batching.
   e) **Maximum number of failed batches**: Enter a positive integer that sets the maximum number of batches that can fail before the operation is ended. Entering 0 allows no failed batches.
3. When you have input all your changes, click **OK**.
4. Save the graph.

After configuration, the Writer Edit Component dialog should look like this example:



Running the delete data graph

After creating the graph and configuring the components, you can run the graph to delete the specified records and/or record assignments from the MDEX Engine.

To run the graph to delete data from the MDEX Engine:

1. Make sure that you have an MDEX Engine running on the host and port that are configured in the **Delete Data** connector.
2. Run the graph using one of the run methods.

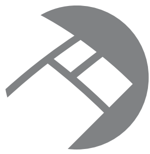
   For example, you can click the green circle with white triangle icon in the Tool bar:

As the graph runs, the process of the graph execution is listed in the Console Tab. The execution is completed successfully when you see final output similar to this example of deleting three records and/or record assignments:

```
INFO  [WatchDog] - Starting up all nodes in phase [0]
INFO  [WatchDog] - Successfully started all nodes in phase!
INFO  [ENDECA_DELETE_DATA0_0] - Sending in the last batch of deletes
INFO  [WatchDog] - [Clover] Post-execute phase finalization: 0
INFO  [WatchDog] - [Clover] phase: 0 post-execute finalization successfully.
INFO  [WatchDog] - ----------------------** Final tracking Log for phase
[0] **---------------------
INFO  [WatchDog] - Time: 27/05/11 10:17:39
INFO  [WatchDog] - Node                    ID          Port      #Records
      #KB aRec/s   aKB/s
INFO  [WatchDog] - ---------------------------------------------------------
-------------------------
INFO  [WatchDog] - UniversalDataReader    DATA_READER0
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                            Out:0            3
        0       3       0
INFO  [WatchDog] - Delete Data            ENDECA_DELETE_DATA0
             FINISHED_OK
INFO  [WatchDog] -  %cpu:..                             In:0            3
        0       3       0
```

```
INFO  [WatchDog] - ---------------------------------** End of Log **------
--------------------------
INFO  [WatchDog] - Execution of phase [0] successfully finished - elapsed
time(sec): 1
INFO  [WatchDog] - -----------------------** Summary of Phases execution
**---------------------
INFO  [WatchDog] - Phase#            Finished Status         RunTime(sec)
   MemoryAllocation(KB)
INFO  [WatchDog] - 0                 FINISHED_OK                        1
             6119
INFO  [WatchDog] - ------------------------------** End of Summary **-----
----------------------
INFO  [WatchDog] - WatchDog thread finished - total execution time: 1 (sec)
INFO  [main] - Freeing graph resources.
INFO  [main] - Execution of graph successful !
```

As the example shows, the Final Tracking Log lists the number of records that were read in by the **UniversalDataReader** component and the number of records that were sent to the MDEX Engine by the **Delete Data** connector.



Chapter 13

Latitude Connector Reference

This chapter provides a reference for the Endeca Latitude connectors available in the LDI Designer palette.

Bulk Add/Replace Records connector

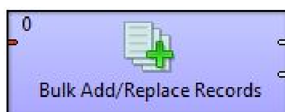
This connector adds new records or replaces existing records in the MDEX Engine.

The **Bulk Add/Replace Records** connector adds or replaces records via the MDEX Engine's bulk ingest interface (that is, it does not use the Data Ingest Web Service).

The characteristics of this connector are:

- The connector can load data source records only.
- Existing records in the MDEX Engine are replaced, not updated. That is, the replace operation is not additive. Therefore, the key/value pair list of the incoming record will completely replace the key/value pair list of the existing record.
- The connector cannot load PDRs, DDRs, managed attribute values, the GCR, nor the MDEX Engine index configuration files.
- A primary-key attribute (also called the record spec) is required for each record to be added or replaced.
- If an assignment is for a standard attribute (property) that does not exist in the MDEX Engine, the new standard attribute is automatically created with system default values for the PDR (see Chapter 1 in this guide for a list of these values).
- No client-side batching is used and there is only a single, streaming connection to the MDEX Engine.
- You can run this connector in a sub-graph within a top-level graph that starts an outer transaction. For the bulk records operations to run successfully within an outer transaction, the connector relies on an outer transaction ID. You should specify this ID in the `MDEX_TRANSACTION_ID` parameter in the `workspace.prm` file in your project.

When added to a graph, the connector icon looks like this:



Metadata schema

The metadata schema for the **Bulk Add/Replace Records** connector is not fixed. Therefore, each LDI field represents a property on an MDEX record.

The metadata type of the LDI field (as shown in the LDI Metadata Editor) translates to the `mdex` property type. For example, the LDI `integer` data type translates to the `mdex:int` data type. Note that this behavior can be overridden to support LDI non-native types (such as `mdex:duration`, `mdex:time`, and `mdex:geocode`).

Use cases

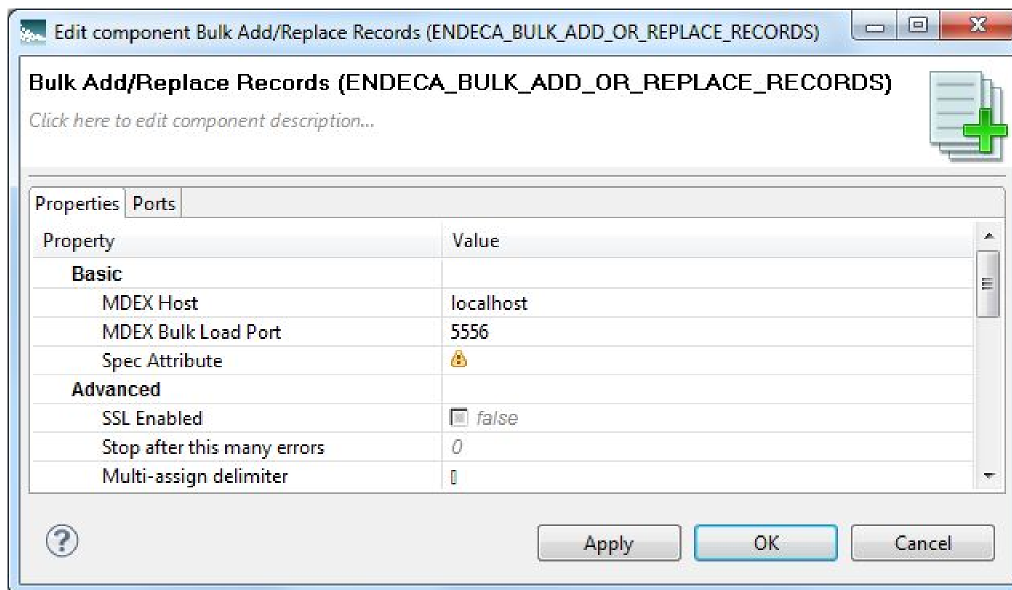
The **Bulk Add/Replace Records** connector is intended to be used with bulk data when delayed update visibility and compromised concurrent query performance are acceptable.

Some of the use cases for this connector are:

- Full index initial load of records, with no loaded schema. In this scenario, the MDEX Engine has no data records and also has no user-created schema (such as no existing PDRs). In this case, all new properties (including the primary-key properties) are created by DIWS with system default values (see Chapter 1 in this guide for a list of these values).
- Full index initial load of records, with your record schema already loaded. You can load the record schema (PDRs and DDRs) with the **Add/Update Records** connector.
- Adding more new records to the MDEX Engine any time after the initial loading of records. As in the initial load case, new standard attributes that do not exist in the MDEX Engine are automatically created with default system values.
- Replacing existing records in the MDEX Engine any time after the initial loading of records. In this case, all the key/value pairs of the existing record are replaced with the key/value pairs of the input file.

Configuration properties

The configuration for the **Bulk Add/Replace Records** connector is set via the Designer Edit component:



The **Basic** and **Advanced** configuration properties that you can set are listed in the following table. For the other properties, see the "Visual and Common configuration properties" topic in this chapter.

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Bulk Load Port	Identifies the bulk load port on which the MDEX Engine is listening. Note that this port is different from the HTTP port used by the all the other connectors.	The bulk load port is determined in one of two ways: • The Dgraph `--bulk_load_port` flag is used when the MDEX Engine is started. • If `--bulk_load_port` flag is not used, then the default bulk load port is the standard Dgraph port plus one. This means that the bulk load port is either 5556 (if the Dgraph `--port` flag is not used) or is the value of the `--port` flag plus one.
Spec Attribute	Sets the primary key (record spec) for the records to be added or updated.	The name of the primary key. If the primary-key property does not exist in the MDEX Engine, the property is automatically created with the system default values.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Stop after this many errors	Sets the maximum number of ingest errors that can occur. The ingest operation is ended after this number of errors is reached.	Either `0` (which means no failures are allowed) or a positive integer.
Multi-assign delimiter	Sets the character that separates multi-assign values in a property in a source record. Keep in mind that this delimiter is different from the delimiter that separates property fields on the source record.	A single character that is the multi-assign delimiter. You do not have to use this field if your source does not have multi-assign properties.

MDEX Engine status after a failed ingest operation

When a bulk load ingest operation is terminated because of an error, records that were ingested before the error should be in the MDEX Engine. Although the MDEX Engine may accept queries on the ingested records, you should consider the MDEX Engine to be in an inconsistent state.

Add/Update Records connector

This connector adds new records or updates existing records in the MDEX Engine.

The **Add/Update Records** connector adds or updates records via the Data Ingest Web Service (DIWS).

The characteristics of this connector are:

- The connector can load data source records, PDRs (Property Description Records), and DDRs (Dimension Description Records).
- The connector cannot load managed attribute values, the GCR (Global Configuration Record), nor the MDEX Engine index configuration files (such as the search interface configuration).
- A primary-key attribute (also called the record spec) is required for each record to be added or updated.
- If an assignment is for a standard attribute that does not exist in the MDEX Engine, the new standard attribute is automatically created with system default values for the PDR (see Chapter 1 for a list of these values).
- Updates are batched on the client-side with multiple concurrent connections to the MDEX Engine.

When added to a graph, the connector icon looks like this:

Metadata schema

The metadata schema for the **Add/Update Records** connector is not fixed. Therefore, each LDI field represents a property on an MDEX record.

The metadata type of the LDI field (as shown in the LDI Metadata Editor) translates to the `mdex` property type. For example, the LDI `integer` data type translates to the `mdex:int` property type. Note that this behavior can be overridden to support LDI non-native types (such as `mdex:duration`, `mdex:time`, and `mdex:geocode`).

Use cases

The **Add/Update Records** connector is intended to be used for non-bulk data when immediate update visibility is desired and/or high concurrent query performance is important.

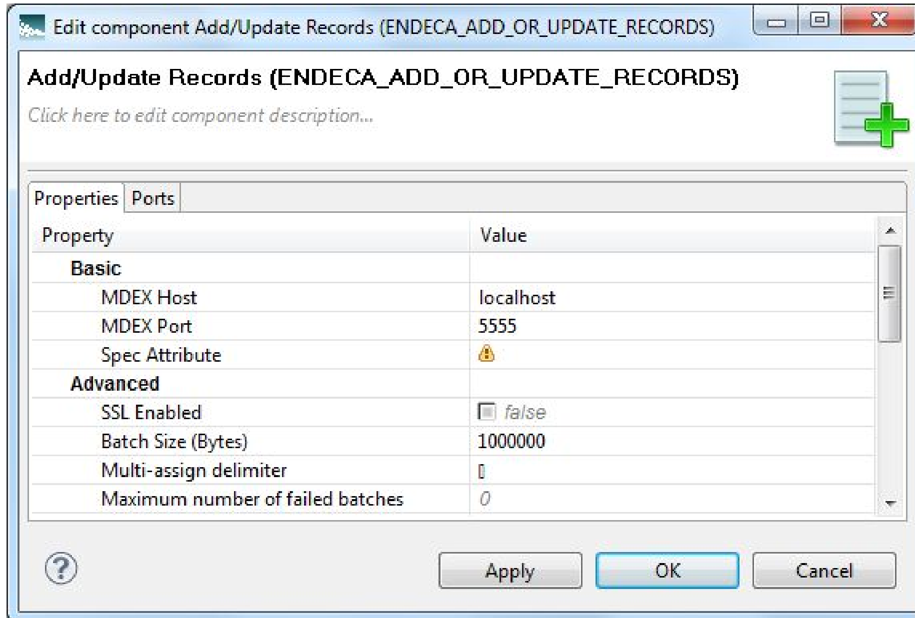
Some of the use cases for this connector are:

- Full index initial load of records, with no loaded schema. In this scenario, the MDEX Engine has no data records and also has no user-created schema (such as no existing PDRs). In this case, all new properties (including the primary-key properties) are created by DIWS with system default values (see the Chapter 1 in this guide for a list of these values).
- Loading of the record schema before an initial load. In this case, you load your PDR schema records (and, optionally, your DDR schema) before loading your data records.
- Full index initial load of records, with your record schema already loaded.
- Incremental updates involving the addition of new records to the MDEX Engine any time after the initial loading of records. As in the initial load case, new standard attributes that do not exist in the MDEX Engine are automatically created with default system values.
- Incremental updates to existing records, which means adding key-value pairs. If a standard attribute is configured as multi-assign, a record can have multiple assignments of that attribute. The records to be updated are considered totally additive. That is, the key-value pair list of the update record will be merged into the existing record. If attribute values with the same name already exist, then

the new values will be additional values for the same standard attribute (multi-assign). Keep in mind that this operation can also be performed by the **Add KVPs** connector.

Configuration properties

The configuration for the **Add/Update Records** connector is set via the Designer Edit component:



The **Basic** and **Advanced** configuration properties that you can set are listed in the following table. For the other properties, see the "Visual and Common configuration properties" topic in this chapter.

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
Spec Attribute	Sets the primary key (record spec) for the records to be added or updated.	The name of the primary key. If the primary-key property does not exist in the MDEX Engine, the property is automatically created with the system default values.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Batch Size (Bytes)	Sets the batch size for the ingest operation. Each record size is	• A number equal to or greater than 1 sets the batch size. If the batch

Configuration Property	Purpose	Valid Values
	calculated in bytes. A batch consists of one or more records.	size is too small to fit in a record, then it is reset to the size to accommodate that record. • Specifying 0 (zero) or a negative number will turn off batching. This means that all records are placed into one batch and sent to the MDEX Engine at the end of the ingest operation.
Multi-assign delimiter	Sets the character that separates multi-assign values in a property in a source record. Keep in mind that this delimiter is different from the delimiter that separates property fields on the source record.	A single character that is the multi-assign delimiter. You do not have to use this field if your source does not have multi-assign properties.
Maximum number of failed batches	Sets the maximum number of batches that can fail before the ingest operation is ended.	Either `0` (which allows no failed batches) or a number greater than `0`.

Batch size adjustments by the connector

Regardless of the batch size you have specified (assuming it is a non-zero, non-negative number), the **Add/Update Records** connector will adjust the batch size on the fly in order to ensure that all the assignments for a given record will fit in the batch. This ensures that assignments for a given record are not split between different batches.

Add KVPs connector

This connector updates Endeca records in the MDEX Engine by adding new key-value pairs to the records.

The **Add KVPs** connector is intended to update records by adding new key-value pair (KVP) assignments to those records. The connector updates records via the Data Ingest Web Service (DIWS).

The characteristics of this connector are:

- The connector can load a new key-value pair for a record.
- Only Endeca standard attribute values can be loaded. Adding managed attribute values is not supported.
- The key-value pairs can only be added. Existing key-value pairs on records cannot be deleted or replaced.
- Multi-assign properties cannot be added. To do this, you need to add separate rows in the input file for multiple assignments of a given property.
- If an assignment is for a standard attribute (property) that does not exist in the MDEX Engine, the new standard attribute is created by DIWS with system default values for the PDR (see Chapter 1 for a list of these values). You can, however, specify a property type for the new standard attribute.
- The main use case is one where your source data is stored in a key-value pair format, as opposed to something like a rectangular data model.

When added to a graph, the connector icon looks like this:

Metadata schema

The metadata schema of the **Add KVPs** connector is fixed and uses a specific ordering. The first row of the data source input file is the record header row and must use this schema:

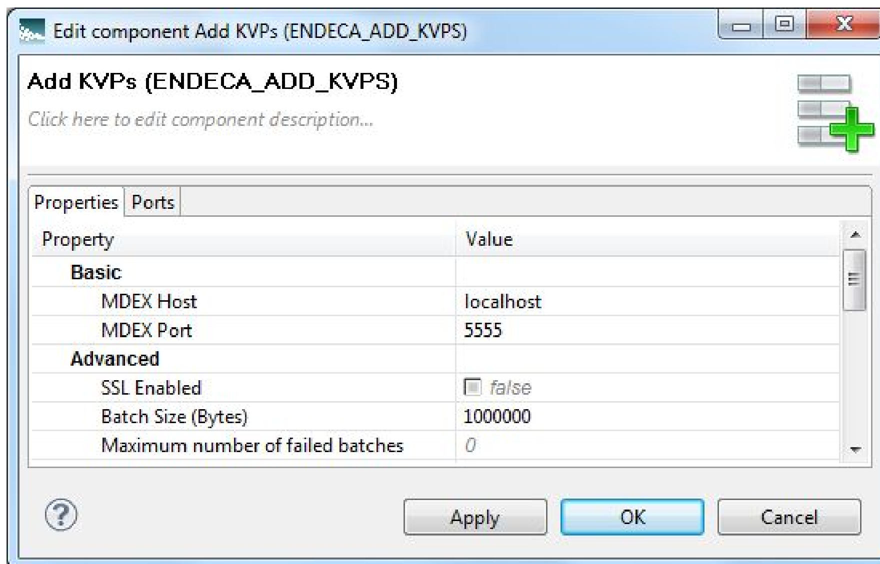
```
specKey|specValue|kvpKey|kvpValue|mdexType
```

where:

- *specKey* is the primary key (record spec) of the record to which the key-value pair will be added.
- *specValue* is the value of the record's primary key.
- *kvpKey* is the name (key) of the Endeca standard attribute to be added to the record. If the standard attribute does not exist in the MDEX Engine, it is automatically created by DIWS with system default values.
- *kvpValue* is the value of the standard attribute to be added.
- *mdexType* specifies the `mdex` property type (such as `mdex:int` or `mdex:dateTime`). This parameter is intended for use when you want to create a new standard attribute and want to specify its property type. If a new PDR for the standard attribute is created and *mdexType* is not specified, then the type of the new standard attribute will be `mdex:string`. If the standard attribute already exists, you can specify an empty value for *mdexType*.

Configuration properties

The configuration for the **Add KVPs** connector is set via the Designer Edit component:



The configuration properties that you can set are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Batch Size (Bytes)	Sets the batch size for the ingest operation. Each record size is calculated in bytes. A batch consists of one or more records.	• A number equal to or greater than 1 sets the batch size. If the batch size is too small to fit in a record, then it is reset to the size to accommodate that record. • Specifying 0 (zero) or a negative number will turn off batching. This means that all records are placed into one batch sent to the MDEX Engine at the end of the ingest operation.
Maximum number of failed batches	Sets the maximum number of batches that can fail before the ingest operation is ended.	Either `0` (which allows no failed batches) or a positive integer.

Add Managed Values connector

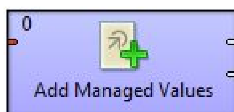
This connector loads a taxonomy into the MDEX Engine's data set.

The **Add Managed Values** connector is intended to load a taxonomy (Endeca managed attribute values) into the MDEX Engine. The taxonomy is loaded via the Data Ingest Web Service (DIWS).

The characteristics of this connector are:

- The connector loads only managed values (mvals). It does not load standard values (svals).
- All the managed values must belong to only one managed attribute.
- If the managed attribute does not exist in the MDEX Engine, the managed attribute is created by DIWS with system default values for the DDR and (if does not already exist) for the PDR. See Chapter 1 in this guide for a list of the default values.
- Optionally, synonyms can be created for managed values.

When added to a graph, the connector icon looks like this:



Metadata schema

The metadata schema of the **Add Managed Values** connector is fixed and uses a specific ordering. The first row of the data source input file is the record header row and must use this schema:

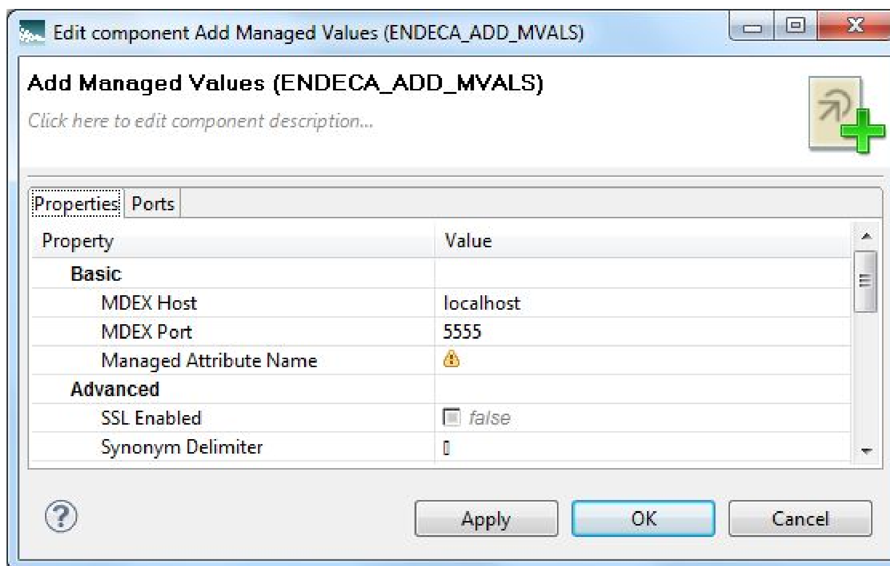
```
spec|displayname|parent|synonym
```

where:

- *spec* is a unique string identifier for the managed value. This is the managed value spec.
- *displayname* is the name of the managed value.
- *parent* is the parent ID for this managed value, If this is a root managed value, use a forward slash (/) as the ID. If this is a child managed value, specify the unique ID of the parent managed value.
- *synonym* optionally defines the name of a synonym. Synonyms can be added to both root and child managed values. You can add multiple synonyms to a single managed value, with the synonyms separated by a delimiter that you specify in the configuration dialog.

Configuration properties

The configuration for the **Add Managed Values** connector is set via the Designer Edit component:



The configuration properties that you can change in the Edit component are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
Managed Attribute Name	Sets the name of the managed attribute to which the managed values will be added.	The name of a managed attribute. The name must use the NCName format. If the managed attribute does not exist in the MDEX Engine, DIWS automatically creates the managed attribute with system default values.

Configuration Property	Purpose	Valid Values
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Synonym delimiter	Sets the delimiter for specifying multiple synonyms.	A single character that is the synonym delimiter.

Delete Data connector

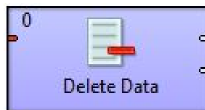
This connector performs delete operations on Endeca records.

The **Delete Data** connector performs these delete operations via the Data Ingest Web Service (DIWS):

- Deletes an entire record.
- Deletes a specific value assignment from a specific Endeca standard attribute on a specific record.
- Deletes all value assignments from a specific standard attribute on a specific record.

Note that the connector cannot remove managed values from records.

When added to a graph, the connector icon looks like this:



Metadata schema

The metadata schema of the **Delete Data** connector is fixed and uses a specific ordering. The first row of the data source input file is the record header row and must use this schema:

```
specKey|specValue|kvpKey|kvpValue
```

where:

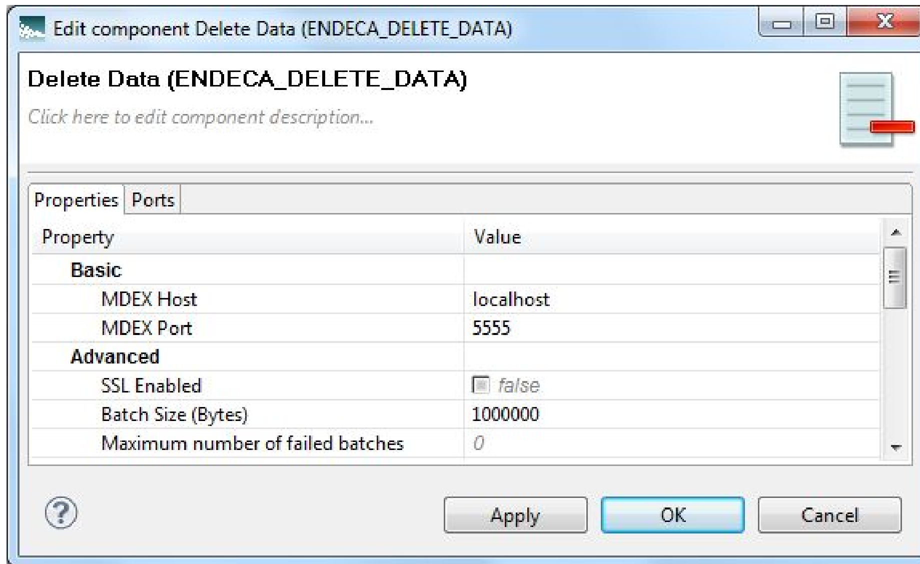
- *specKey* is the name of the primary key (record spec) of the record on which the delete operation will be performed.
- *specValue* is the value of the record's primary key.
- *kvpKey* is the name (key) of the Endeca standard attribute to which the assignment belongs. If *kvpValue* is blank, then all assignments of *kvpKey* are deleted. If both *kvpKey* and *kvpValue* are blank, then the entire record is deleted.
- *kvpValue* is the assigned value to be removed.

The following is a simple example of an input file for the **Delete Data** connector:

```
specKey|specValue|kvpKey|kvpValue
ProductID|3000|Color|purple
ProductID|4000|Availability|
ProductID|5000||
```

Configuration properties

The configuration for the **Delete Data** connector is set via the Designer Edit component:



The configuration properties that you can set are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Batch Size (Bytes)	Sets the batch size for the delete ingest operation. Each record size is calculated in bytes. A batch consists of one or more records to be sent to the MDEX Engine for deletion.	• A number equal to or greater than 1 sets the batch size. If the batch size is too small to fit in a record, then it is reset to the size to accommodate that record. • Specifying 0 (zero) or a negative number will turn off batching. This means that all records are placed into one batch sent to the MDEX Engine at the end of the ingest operation.

Configuration Property	Purpose	Valid Values
Maximum number of failed batches	Sets the maximum number of batches that can fail before the ingest operation is ended.	Either `0` (which allows no failed batches) or a positive integer.

Export Config connector

This connector lets you export the schema and configuration stored in the MDEX Engine.

The **Export Config** connector exports the schema and configuration using the Configuration Web Service requests. The characteristics of this connector are:

- The connector lets you export your configuration and schema by pointing to an output port. This port can be connected to another component, such as any writer component that would write the exported configuration and schema into a file. This file can later be used for importing.
- You can run this connector in a sub-graph within a top-level graph that starts an outer transaction. For the export operation to run successfully within an outer transaction, the connector relies on an outer transaction ID. You should specify this ID in the `MDEX_TRANSACTION_ID` parameter in the `workspace.prm` file in your project.
- The connector exports all configuration and schema but it does not export the file that stores all word forms used for stemming dictionaries in the MDEX Engine. This file is created automatically when you provision the MDEX Engine, and is typically not modified. In rare cases when you may need to make changes to this file, use a custom component in LDI that can export and re-import this file using the Configuration Web Service operations.

When added to a graph, the connector icon looks like this:

Use cases

The **Export Config** and **Import Config** connectors are intended to be used in the following cases:

- Both of these connectors support cases where, after loading the default configuration, you change portions of it in Latitude Studio, such as attribute groups, or attribute group names. From this point on, you may want to keep using this changed configuration, even if you run subsequent data updates. The connectors allow you to do this.
- The **Export Config** should also be used as part of the graph in which you run a baseline update for loading data (although, it is not intended to be used with the initial baseline update).

  In a typical scenario of a repeatable baseline update, you create a graph in which you start a transaction using the **Transaction RunGraph** component, export all configuration and schema using **Export Config**, run the **Reset MDEX** to remove all records and provision a new MDEX Engine, import the previously saved configuration and schema with **Import Config**, and then reload the records. At this point, the transaction can close and the node on which the baseline update was run can resume answering queries.

Metadata schema

The metadata schema for the **Export Config** connector is not fixed.

Configuration properties

The configuration for the **Export Config** connector is set via the Designer Edit component:

The configuration properties that you can set are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.

Import Config connector

This connector lets you import the schema and configuration into the MDEX Engine.

The **Import Config** connector imports schema and configuration using the Configuration Web Service operations. The characteristics of this connector are:

- This connector lets you import schema and configuration that was previously exported to a file. The **UniversalDataReader** component can read the file that stores all the previously exported configuration and schema. The reader's output port can point to the input port on **Import Config** connector which imports this file.
- You can run this connector in a sub-graph within a top-level graph that starts an outer transaction. For the import operation to run successfully within an outer transaction, the connector relies on an outer transaction ID that you must specify in the `MDEX_TRANSACTION_ID` parameter in the `workspace.prm` file in your project.
- When importing, be aware that only basic XML validation takes place. Since the **Import Config** connector uses the Configuration Web Service operations, the configuration that is sent to the MDEX Engine must be the one that the Configuration Web Service is designed to accept. Thus, the file that you are importing must comply with the requirements of the Configuration Web Service WSDL document, and contain only valid records describing the configuration and schema.

When added to a graph, the connector icon looks like this:

Use cases

The **Import Config** and **Export Config** connectors are intended to be used in the following cases:

- Both of these connectors support cases where, after loading the default configuration, you change portions of it in Latitude Studio, such as attribute groups, or attribute group names. From this point on, you may want to keep using this changed configuration, even if you run subsequent data updates. The connectors allow you to do this.
- The **Import Config** should be used as part of the graph in which you run a baseline update for loading data (although, it is not intended to be used with the initial baseline update).

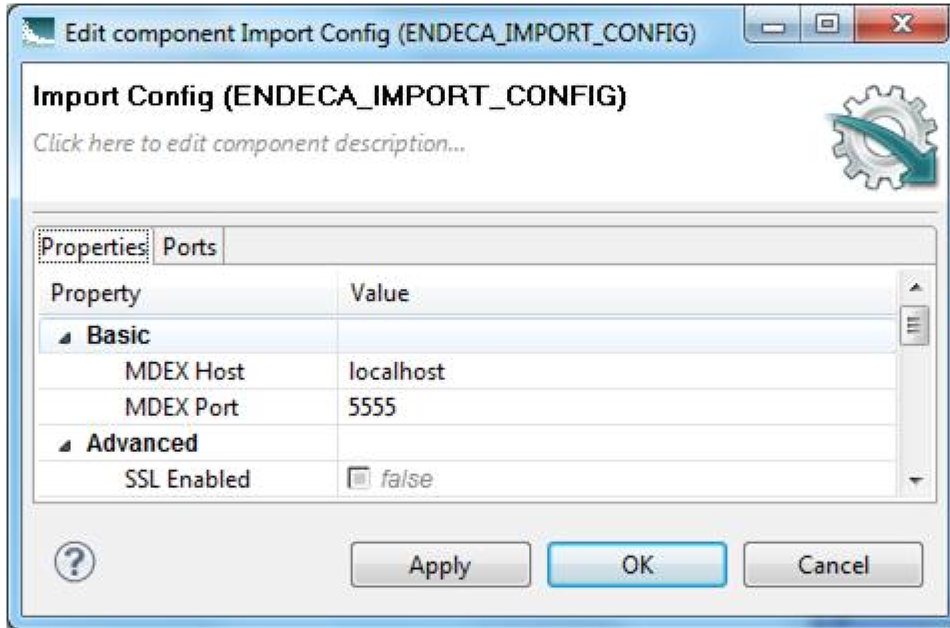
  In a typical scenario of a repeatable baseline update, you create a graph in which you start a transaction using the **Transaction RunGraph** component, export all configuration and schema using **Export Config**, run the **Reset MDEX** to remove all records provision a new MDEX Engine, import the previously saved configuration and schema with **Import Config**, and then reload the records. At this point, the transaction can close and the node on which the baseline update was run can resume answering queries.

Metadata schema

The metadata schema for the **Import Config** connector is not fixed.

Configuration properties

The configuration for the **Import Config** connector is set via the Designer Edit component:



The configuration properties that you can set are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.

Reset MDEX connector

This connector lets you reset the MDEX Engine index back to the empty state.

The connector does this by removing all the records (including the schema) from the MDEX Engine, provisioning the MDEX Engine and updating the spelling dictionary.

The characteristics of this connector are:

- The **Reset MDEX** connector utilizes operations from the Data Ingest Web Service. These operations delete all records (including schema records) and configuration, and provision the MDEX Engine. Next, this connector utilizes an administrative command for updating the spelling dictionary (`admin?op=updateaspell`).

- You can run this connector in its own graph, or within a graph that starts an outer transaction. In particular, Endeca recommends to run the **Reset MDEX** connector within a **Transaction RunGraph**. Note that only one outer transaction can be open at a time.
- You can run this connector in a sub-graph within a top-level graph that starts an outer transaction. For the reset operations to run successfully, the connector relies on an outer transaction ID. You should specify this ID in the `MDEX_TRANSACTION_ID` parameter in the `workspace.prm` file in your project.

When added to a graph, the connector icon looks like this:

Use cases

The **Reset MDEX** should be used as part of the graph in which you run a baseline update (although, it is not intended to be used with the initial baseline update).

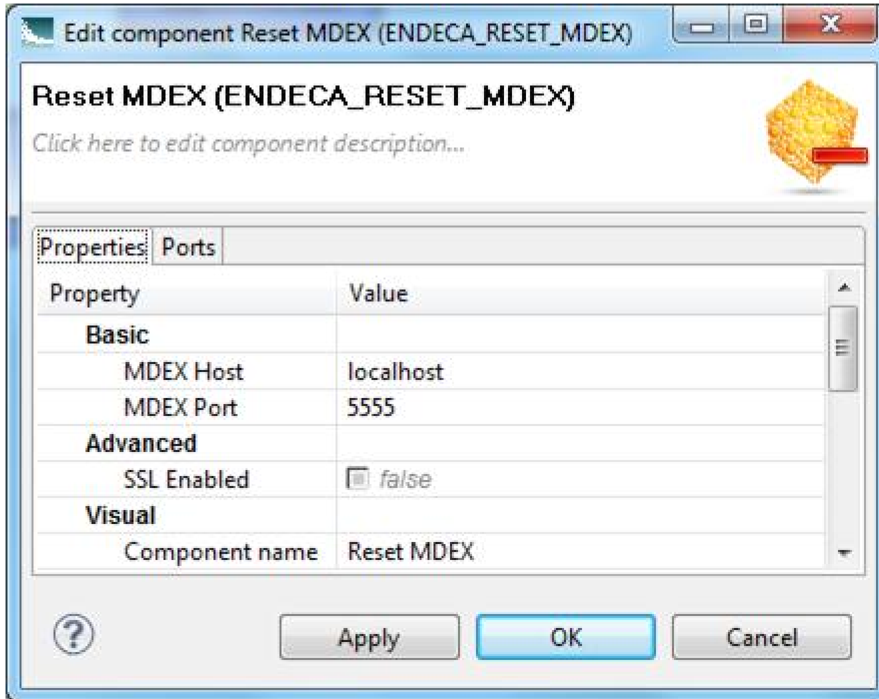
- In a typical scenario of a repeatable baseline update, you create a graph in which you start a transaction using the **Transaction RunGraph** component, export all configuration and schema using **Export Config**, run the **Reset MDEX** to remove all records and provision a new MDEX Engine, import the previously saved configuration and schema with **Import Config**, and then reload the records. At this point, the transaction can close and the node can resume answering queries.

Metadata schema

The metadata schema for the **Reset MDEX** connector is not fixed.

Configuration properties

The configuration for the **Reset MDEX** connector is set via the Designer Edit component:



The configuration properties that you can set are:

Configuration Property	Purpose	Valid Values
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.

Transaction RunGraph connector

Use this connector to run LDI graphs, similar to the standard **RunGraph** component available with the LDI. Unlike the standard **RunGraph**, **Transaction RunGraph** starts the outer transaction and runs multiple sub-graphs within that transaction.

The **Transaction RunGraph** connector has the following characteristics:

- It is similar to the **RunGraph** component in LDI — it runs one or more LDI graphs. If one sub-graph will be run inside **Transaction RunGraph**, you specify its name in the component's **Graph URL** attribute.

If more than one sub-graph will be run, you can include names of all sub-graphs in a single file and send this file through the **UniversalDataReader** to the input port of the **Transaction RunGraph** connector.

- The **Transaction RunGraph** starts and commits an outer transaction using the Transaction Web Service. Only one outer transaction can be open at a time.
-
- In case of transaction failure, the connector rolls back to the state before the transaction had started, and commits the transaction. (This is the default behavior in case of a transaction failure, but you can configure other options, **Commit** and **Do nothing**, described below.)
- Because this connector starts an outer transaction, it uses the outer transaction ID as follows: The **Transaction RunGraph** overrides the transaction ID with the string `transaction`, for the duration of the transaction. This assumes that the project uses the empty string value for the ID in `workspace.prm`, specified as follows: `MDEX_TRANSACTION_ID=`. When **Transaction RunGraph** runs, the empty ID string is overwritten by the string `transaction`.
- All connectors or sub-graphs that:
  - Utilize a request to the MDEX Engine through a web service or Bulk Ingest Interface, and
  - Run inside **Transaction RunGraph**

  must reference the outer transaction ID. You should specify this ID as an empty string as follows: `MDEX_TRANSACTION_ID=` in the `workspace.prm` file for your project. All Latitude-specific connectors that utilize MDEX Engine web services or the Bulk Interface automatically reference this ID. Additionally, if these components are run within **Transaction RunGraph**, they use the ID `transaction`.
- If you are using a **WebServiceClient** component in LDI that is configured to run any of the MDEX Engine web services, and plan to use this component inside **Transaction RunGraph**, the **Request Structure** field for the component must include an attribute `outerTransactionId` with a value of an outer transaction.

  Note: If you do not use transactions, then your web service-based components should still use the `outerTransactionID` referencing the value in `workspace.prm`. If the value is empty, the transaction ID attribute is ignored by the MDEX Engine. This allows e components to run outside of transactions, without having to modify `workspace.prm`.

  For example, the following request specified in the **Request Structure** references the outer transaction ID as a parameter:

```
<config-service:configTransaction
xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

outerTransactionId="${MDEX_TRANSACTION_ID}">
<config-service:putGroups
xmlns:config-service="http://www.endeca.com/MDEX/config/services/types"

xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
...
</config-service:putGroups>
</config-service:configTransaction>
```

  In this example, the string `outerTransactionId="${MDEX_TRANSACTION_ID}"` specifies the ID of the outer transaction listed in the `workspace.prm` file for your project.

When added to a graph, the connector icon looks like this:

Use cases

The **Transaction RunGraph** should be used as part of the graph in which you run a baseline update:

- In a typical scenario of an initial baseline update, you create a graph in which you start an outer transaction using the **Transaction RunGraph** component, provision the MDEX Engine using the **Reset MDEX** component, and then use one or more sub-graphs to load data and configuration.
- In a typical scenario of a repeatable baseline update, you create a graph in which you start an outer transaction using the **Transaction RunGraph** component, export all configuration and schema using **Export Config**, run **Reset MDEX** to remove all records and provision a new MDEX Engine, import the previously saved configuration and schema with **Import Config**, and then reload the records and the record attribute values. At this point, the outer transaction can close and the node on which the baseline update was run resumes processing query requests.

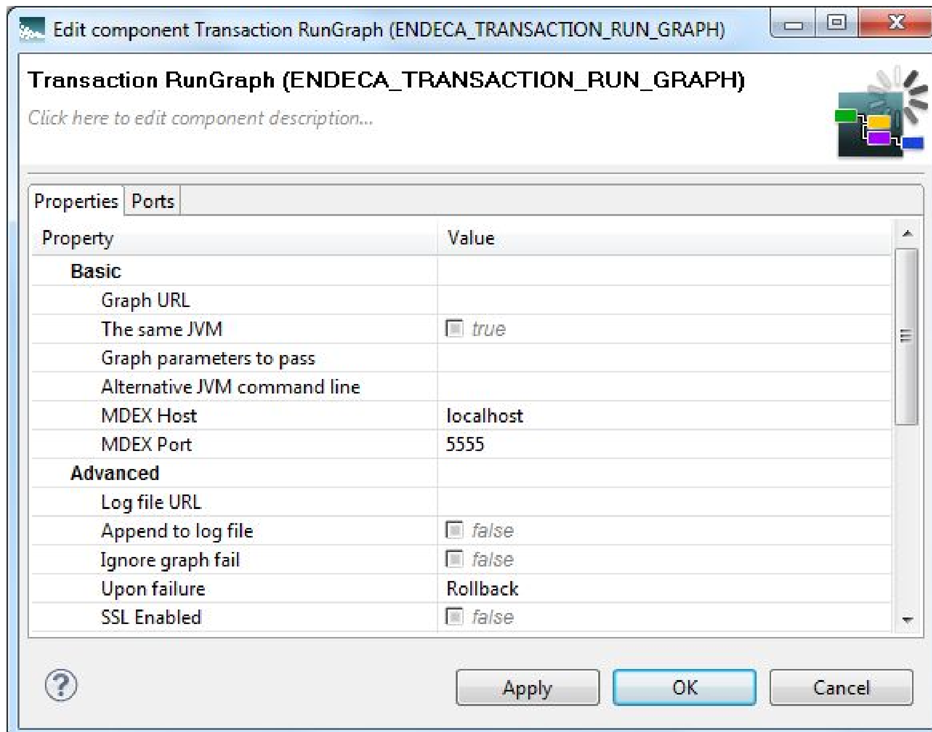
Metadata schema

The metadata schema for the **Transaction RunGraph** connector is not fixed.

The metadata type of the LDI field (as shown in the LDI Metadata Editor) translates to the `mdex` attribute type. For example, the LDI `integer` data type translates to the `mdex:int` data type. Note that this behavior can be overridden to support LDI non-native types (such as `mdex:duration`, `mdex:time`, and `mdex:geocode`).

Configuration properties

The configuration for the **Transaction RunGraph** connector is set via the Designer Edit component:



The **Basic** and **Advanced** configuration properties that you can set are listed in the following table. For the other properties, see the "Visual and Common configuration properties" topic in this chapter.

Configuration Property	Purpose	Valid Values
Graph URL	Identifies the name of one graph, including path, that should be executed by the component.	Any of the Latitude graphs, or other graphs can be used. If non-Latitude graphs are used that call any of the MDEX Engine web services, they should reference the outer transaction ID in their **Request Structure**: `outerTransaction¬ Id="${MDEX_TRANSACTION_ID}`
MDEX Host	Identifies the machine on which the MDEX Engine is running.	The name or IP address of the machine. `localhost` can be used as the name.
MDEX Port	Identifies the port on which the MDEX Engine is listening.	The port number on which the MDEX Engine was started.
SSL Enabled	Enables or disables SSL for the connector.	• If `false` (the default), SSL is disabled. • If `true`, SSL is used for connections to the MDEX Engine. In this case, the MDEX Engine must also be SSL-enabled.
Upon failure	Enables selecting one of the three options in case of failure: commit, rollback, and do nothing.	• **Rollback**. This is the default. In case of transaction failure, enables to roll back to the state before the transaction had started, and commit the transaction. • **Commit**. In case of transaction failure, enables to commit those changes that have been made successfully before the failure had occurred, and commit the transaction. • **Do nothing**. In case of failure, does nothing. In this case, you may need to investigate the logs, and decide whether you want to apply any of the actions that are configured within the transaction manually. Note that in this case you may also need to manually stop the outer transaction by using a graph that runs the "commit transaction" operation.

Related Links

Creating a Transaction RunGraph graph on page 34

This section describes how to build an LDI graph that uses the **Transaction RunGraph** connector to run a series of graphs within a single atomic transaction.

Visual and Common configuration properties

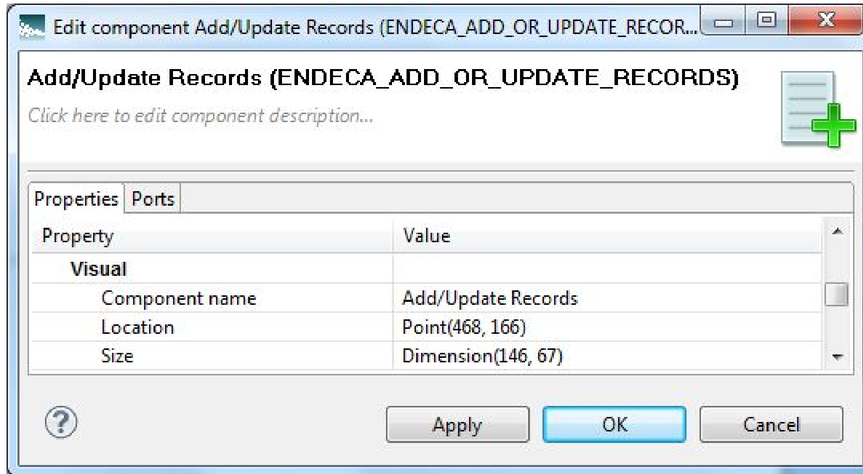
This topic describes the meanings of the **Visual** and **Common** configuration properties of connectors.

All Latitude connectors have **Visual** and **Common** properties in their configuration dialogs. Because the functionality of these properties is the same across all the connectors, an overview of these

properties can be described in a common topic. For more information on the purpose of these properties, see the *Latitude Data Integrator Designer Guide*.

Visual properties

Visual properties can be seen in the graph. The **Visual** section looks like this in the Edit component:

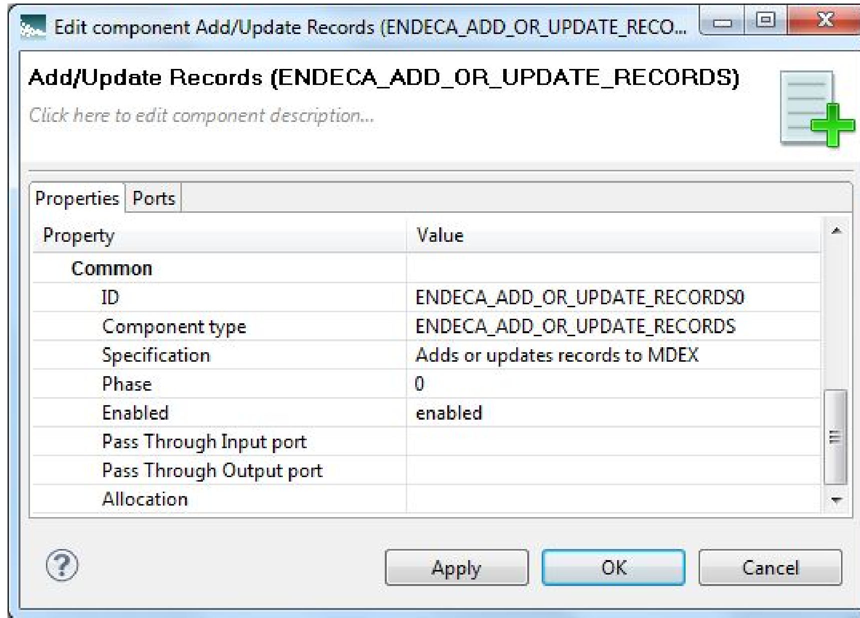


The **Visual** configuration properties are:

Visual property	Purpose	Valid values
Component name	Displays the component name when the component is placed on a graph.	You can change the default name to a more descriptive one.
Location	Describes the location (using an X-axis and Y-axis) of the component icon within the graph.	Do not edit this field. Instead, use your cursor to move the component in the graph to the desired position.
Size	Describes the dimensions (size) of the component icon within the graph.	Do not edit this field.
Click here to edit component description	Located in the header of the component, this field lets you add some descriptive text that is displayed in the component icon in the graph.	Text describing what this component does (for example, text that best describes what this component does in the graph).

Common properties

Common properties are common to all components. The **Common** section looks like this in the Edit component:



The **Common** configuration properties are:

Common property	Purpose	Valid values
ID	Identifies the component among all of the other components within the same component type.	Do not edit this field.
Component type	Describes the type of the component. By adding a number to this component type, you can get a component ID.	Do not edit this field.
Specification	Describes what this component can do.	Do not edit this field.
Phase	Sets the phase number for the component. Because each graph runs in parallel within the same phase number, all components and edges that have the same phase number run simultaneously.	An integer number of the phase to which the component belongs.
Enabled	Enables or disables the component for parsing data.	• `enabled` (the default) means the component can parse data. • `disabled` means the component does not parse data. • `passThrough` puts the component in passThrough mode, in which data records will pass through the component from input to output ports and the component will not change them.

Common property	Purpose	Valid values
Pass Through Input Port	If the component runs in passThrough mode, you can specify which input port should receive the data records.	Select the input port from the list of all input ports.
Pass Through Output Port	If the component runs in passThrough mode, you can specify which output port should send the data records out.	Select the output port from the list of all output ports.
Allocation	If the graph is executed by a Cluster of LDI Servers, this attribute must be specified in the graph.	For information on this property, see the *Latitude Data Integrator Designer Guide*.

Connector output ports

The Latitude connectors that deal with ingest have two output ports each.

The two output ports are:

- **Port 0** returns status information. That is, it describes how many batches of records were successfully ingested.
- **Port 1** returns error information. That is, it describes the batches of records that failed to ingest. Note that each record corresponds to a failed batch, not individual records.

Port 0 metadata

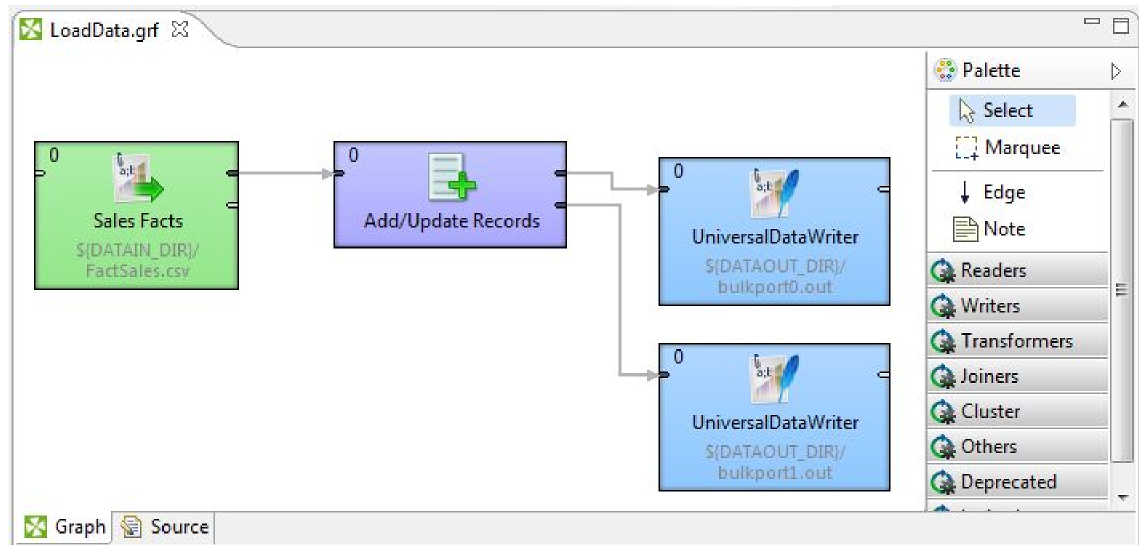
Connector	Field 1	Field 2	Field 3	Field 4	Field 5
Add/Update Records	Start Row (Long)	End Row (Long)	Number of Records Affected (Long)	Time Taken in Seconds (Numeric)	n/a
Add KVPs	Start Row (Long)	End Row (Long)	Number of Records Affected (Long)	Time Taken in Seconds (Numeric)	n/a
Add Managed Values	Start Row (Long)	End Row (Long)	Number of Managed Attributes Added (Long)	Number of Managed Values Added (Long)	Time Taken in Seconds (Numeric)
Delete Data	Start Row (Long)	End Row (Long)	Number of Records Deleted (Long)	Number of Records Affected (Long)	Time Taken in Seconds (Numeric)
Bulk Add/Replace Records	Records Added (Long)	Records Queued (Long)	Records Rejected (Long)	State (String)	n/a

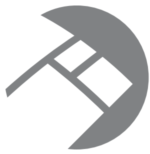
Port 1 metadata

Connector	Field 1	Field 2	Field 3
All DIWS connectors	Start Row (Long)	End Row (Long)	Fault Message (String)
Bulk Add/Replace Records	Fault Message (String)	n/a	n/a

Writing the output to a file

You can write the output port information to a file by connecting a Writer component to the output port of the Latitude connector. This sample graph has one **UniversalDataWriter** component writing out data from port 0 of the **Add/Update Records** connector and a second one attached to Port 1 of the connector:





Chapter 14

Troubleshooting Problems

This section provides information and solutions to problems you may encounter when working with connectors and graphs.

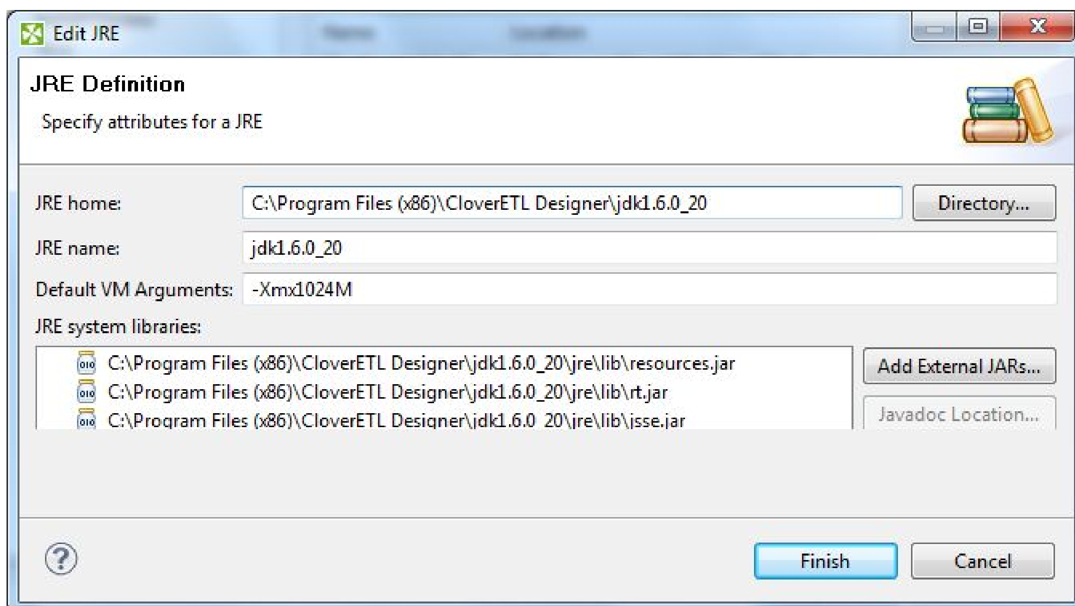
OutOfMemory errors

If the Java process has insufficient memory allocated, you may get `OutOfMemory` errors when running the graph.

In an unsuccessful run, the Console Tab will show an `OutOfMemory` error similar to this example:

```
ERROR [DataIngestBatchConsumer-0] - Failed with the following exception:
        java.lang.OutOfMemoryError: Java heap space
Exception in thread "DataIngestBatchConsumer-0" java.lang.OutOfMemoryError:
 Java heap space
```

You can avoid these errors by increasing the memory allocated to the Java process running the service. The **Edit JRE** menu lets you increase the memory size on a global basis.



To avoid `OutOfMemory` errors:

1. Select **Preferences** from the **Window** menu.
2. From the **Preferences** menu, select **Java** > **Installed JREs**.
3. In the **Installed JREs** menu, click on the checked JRE and then click **Edit**.
   The **Edit JRE** menu is displayed.
4. In the **Default VM Arguments** field, specify a Java option to set the heap size, such as `-Xmx1024M`.
   The Edit JRE menu should look like the example above.
5. Click **Finish** to apply your change and close the **Edit JRE** menu.
6. Click **OK** to close the **Preferences** menu.

BufferOverflow errors

If the size of the data buffer is too small, you may get `BufferOverflow` errors when running the graph.

In an unsuccessful run, the Console Tab will show a `BufferOverflowException` error similar to this example:

```
ERROR [WatchDog] - Node DATA_READER0 error details:
java.lang.RuntimeException: The size of data buffer is only 131072.
Set appropriate parameter in defautProperties file.
    at org.jetel.data.StringDataField.serialize(StringDataField.java:285)
    at org.jetel.data.DataRecord.serialize(DataRecord.java:466)
    at org.jetel.graph.DirectEdge.writeRecord(DirectEdge.java:234)
    at org.jetel.graph.Edge.writeRecord(Edge.java:371)
    at org.jetel.component.DataReader.execute(DataReader.java:264)
    at org.jetel.graph.Node.run(Node.java:425)
    at java.lang.Thread.run(Thread.java:619)
Caused by: java.nio.BufferOverflowException
    at java.nio.Buffer.nextPutIndex(Buffer.java:501)
    at java.nio.DirectByteBuffer.putChar(DirectByteBuffer.java:465)
    at org.jetel.data.StringDataField.serialize(StringDataField.java:282)
    ... 6 more
```

You can avoid these errors by increasing the buffer settings in the `defaultProperties` configuration file, copying the file into your LDI project, and then specifying the file to be used in the run configuration of a graph. The `defaultProperties` configuration file is located in the `cloveretl.engine.jar` JAR file, whose default location is:

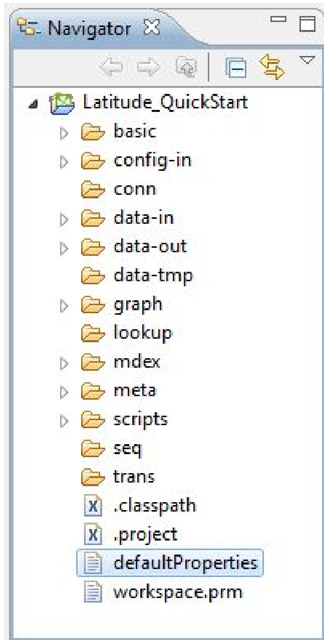
```
DataIntegrator\plugins\com.cloveretl.gui_3.1.0\lib\lib\cloveretl.engine.jar
```

To modify the `defaultProperties` configuration file and add it to your LDI project:

1. Copy the `cloveretl.engine.jar` JAR file to a temporary location (for example, a `temp` directory).
2. Extract the file `org\jetel\data\defaultProperties` from the JAR file into the `temp` directory.
3. Open the `defaultProperties` in a text editor.
4. Increase the sizes of one or more of these properties:
   - `Record.MAX_RECORD_SIZE` (default is 131072)
   - `DataParser.FIELD_BUFFER_LENGTH` (default is 65536)
   - `DataFormatter.FIELD_BUFFER_LENGTH` (default is 65536; this property is typically set to the same size as `DataParser.FIELD_BUFFER_LENGTH`)
   - `DEFAULT_INTERNAL_IO_BUFFER_SIZE` (default is 262144)

The size of your final settings depend on the characteristics of your data set, which could mean that you may have to make several runs of the graph and keep increasing these settings until your ingest operations no longer fail due to memory problems.

5. Place the `defaultProperties` configuration file in your LDI project folder, by copying it into the Navigator pane.



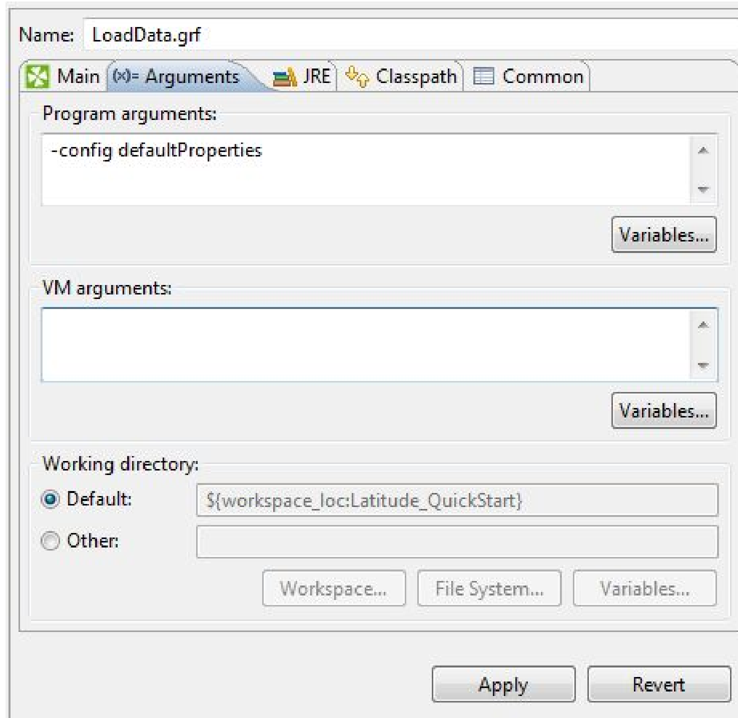
6. From the Designer tool bar, choose **Run** > **Run Configurations**.
7. From the left pane of the Run Configurations menu, select a graph to edit and then click the **Arguments** tab in the run configuration.

   If the graph you want to edit is not listed, you can either run the graph (so that its name will be listed) or create a new configuration for the graph.

8. Enter the following text in the Program arguments field:

```
-config defaultProperties
```

   At this point, the Arguments tab should look like this example:



9. Click **Apply** to save your changes.
10. Click either Run (to run the graph with the modified run configuration) or **Close** (to close the Run Configurations menu).

Transaction-related errors

You may receive various transaction-related errors if you attempt to overlap running graphs wrapped in transactions with graphs that do not start an outer transaction.

In general, only one outer transaction can be open and running at a time.

The following examples illustrate a few possible scenarios in which transaction-related errors may occur:

- Suppose you have two LDI projects, one without a transaction (A), and one with it (B). Project A runs successfully until project B starts (and opens an outer transaction). If project A is half-way through its run and project B starts, the remaining steps in the project A will begin to fail because their components do not reference the outer transaction ID.
- Suppose you have two projects containing **Transaction RunGraph** connectors. They will run successfully if you run them serially, but any attempt to run them in parallel will result in the second project failing.
- Suppose you have a **Transaction RunGraph** set to **Do Nothing** as its failure action. If this graph fails the first time, it will also fail the second time you try to run it, because it is trying to open an outer transaction that has already been started. Therefore, if such a graph fails, to troubleshoot it, run the inner graphs separately, without running the **Transaction RunGraph**. Alternatively, you can manually commit the transaction after each failure, using the `/admin?op=rollback&outer¬TransactionId="myID"` command, where *myID* is the ID of the transaction. Note that `myID` defaults to the `transaction` string when run with a **Transaction RunGraph**.

Consider implementing one of the following recommendations (depending on your use case):

- Identify whether an outer transaction is currently running by issuing an `http://mdex/admin?op=ping` request. An HTTP code 403 means that a transaction is open.
- Before running a graph that is configured to open its own outer transaction, verify that an already running transaction commits successfully.
- In some instances, when an already running transaction fails to commit, you may need to manually commit it. One of the ways to do so is to issue an `/admin?op=rollback` operation on the MDEX Engine server referencing the outer transaction ID. This operation rolls back all the changes from that transaction, and then commits it. Once one transaction is closed, you can start a new transaction, if needed.
- Instead of running a new graph that starts a transaction separately, add a component that was previously part of this graph to any existing graph that starts an outer transaction.

  For example, you can run a graph for importing or exporting configuration and schema inside a **Transaction RunGraph**, or any other sample graph for running a baseline update (for subsequent data loading). Similarly, Endeca recommends that you run the **Reset MDEX** connector inside a graph that starts an outer transaction.

To summarize, to avoid transaction-related errors, ensure that projects containing transactions do not overlap. Errors are avoided if at any given time, only one outer transaction is open.

Related Links

Committing an outer transaction on page 39

To manually commit an outer transaction that failed to commit successfully, run the **RollBackTransaction** or **CommitTransaction** graph from the Latitude Quick Start project, or issue an `/admin?op=rollback&outerTransactionId="ID"` command on the MDEX Engine server, specifying the transaction ID.

Requirements for running graphs within a transaction on page 31

If you would like to use outer transactions in your graphs, consider these requirements.

Connection errors

This topic illustrates connection errors that may occur between your Endeca connectors and the MDEX Engine.

If the MDEX Engine is not running, this error will result when an Endeca Latitude connector attempts to make a connection to the MDEX Engine:

```
ERROR [ENDECA_ADD_KVPS0_0] - Connection refused: connect Error connecting
to the dgraph.
If applicable, ensure your SSL settings are correct.
ERROR [ENDECA_ADD_KVPS0_0] - Failed with the following exception:
 java.rmi.RemoteException: Connection refused: connect Error connecting to
 the dgraph.
 If applicable, ensure your SSL settings are correct.; nested exception is:

 org.apache.axis2.AxisFault: Connection refused: connect
ERROR [WatchDog] - Graph execution finished with error
...
ERROR [WatchDog] - !!! Phase finished with error - stopping graph run !!!
```

The error will also occur if the connector is incorrectly configured as to the MDEX Engine's host name and/or port number, or if a connector that is not enabled for SSL attempts to connect to an SSL-enabled MDEX Engine.

Multi-assign delimiter error

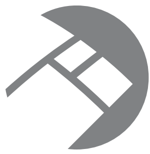
A multi-assign delimiter must be specified when loading multi-assign data.

When loading multi-assign attribute data with either the **Bulk Add/Replace Records** connector or the **Add/Update Records** connector, you must remember to specify the multi-assign delimiter character when configuring the connector.

If you do not specify the delimiter (or specify the wrong one), the ingest operation should fail with an error like the following:

```
ERROR [SocketReader] - Received error message from server: Attempt to
   add/replace record ProductID:34699 with unknown dimension value
   "Red;Green" within dimension "ProductType"
ERROR [WatchDog] - Graph execution finished with error
ERROR [WatchDog] - Node ENDECA_BULK_ADD_OR_REPLACE_RECORDS0 finished
   with status: ERROR
```

In this example, the multi-assign source is "Red;Green" (with the semi-colon being the delimiter). To correct the problem, specify the correct multi-assign delimiter in the **Multi-assign delimiter** field of the connector's configuration screen.



Appendix A

MDEX Engine Index Configuration Reference

This reference describes the XML elements in the MDEX Engine configuration documents. The reference describes each element's format, attributes, and sub-elements, and provides an example of its usage.

XML elements

These common elements are available for use in multiple Endeca index configuration files.

COMMENT

The COMMENT element associates a comment with a pipeline component and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.

Format

```
<!ELEMENT COMMENT (#PCDATA)>
```

Attributes

The COMMENT element has no attributes.

Sub-elements

The COMMENT element has no sub-elements.

Example

This example includes an informational comment.

```
<DIMSEARCH_CONFIG FILTER_FOR_ANCESTORS="FALSE"
  <COMMENT>Displays ancestor managed values.</COMMENT>
/DIMSEARCH_CONFIG>
```

DIMNAME

The DIMNAME element specifies the name of a managed attribute.

Format

```
<!ELEMENT DIMNAME (#PCDATA)>
```

Attributes

The DIMNAME element has no attributes.

Sub-elements

The DIMNAME element has no sub-elements.

Example

This example shows the name of a managed attribute.

```
<RECORD>
   <DIMNAME="ProductType">
   ...
</RECORD>
```

PROP

The PROP element represents an Endeca standard attribute. it can optionally contain a PVAL element.

Format

```
<!ELEMENT PROP (PVAL?)>
<!ATTLIST PROP
 NAME CDATA  #REQUIRED
>
```

Attributes

The PROP element has the following attributes.

NAME

Identifies the name of the standard attribute.

Sub-elements

The PROP element can optionally contain a PVAL element (or it can have no PVAL elements).

Example

This example shows a standard attribute name.

```
<RECORD>
   <PROP NAME="Endeca.Title">
      <PVAL>The Simpsons Archive</PVAL>
   </PROP>
   ...
</RECORD>
```

PROPNAME

The PROPNAME element represents an Endeca standard attribute.

Format

```
<!ELEMENT PROPNAME (#PCDATA)>
```

Attributes

The PROPNAME element has no attributes.

Sub-elements

The PROPNAME element has no sub-elements.

Example

This example shows a standard attribute name.

```
<RECORD>
   <PROPNAME="P_Price">
   ...
</RECORD>
```

PVAL

The PVAL element represents a standard attribute value.

Format

```
<!ELEMENT PVAL (#PCDATA)>
```

Attributes

The PVAL element has no attributes.

Sub-elements

The PVAL element has no sub-elements.

Example

This example shows a standard attribute value.

```
<PROP NAME="Endeca.Title">
   <PVAL>The Simpsons Archive</PVAL>
</PROP>
```

Dimsearch_config elements

The Dimsearch_config element controls how value searches behave.

This file configures search matching, spelling correction, filtering, and relevance ranking for value search. These options are configured in the file's DIMSEARCH_CONFIG root element.

DIMSEARCH_CONFIG

A DIMSEARCH_CONFIG element sets up the configuration of standard and managed attributes for value searches. Value searches search against the text collection that consists of the names of all the attribute values in the data set.

Format

```
<!ELEMENT DIMSEARCH_CONFIG (COMMENT?, PARTIAL_MATCH?)>
<!ATTLIST DIMSEARCH_CONFIG
   FILTER_FOR_ANCESTORS      (TRUE | FALSE)    "FALSE"
   RELRANK_STRATEGY          CDATA             #IMPLIED
>
```

Attributes

The DIMSEARCH_CONFIG element has the following attributes.

FILTER_FOR_ANCESTORS

When set to TRUE, the results of a value search return only the highest ancestor attribute value. This means that if both *red zinfandel* and *red wine* match a search query for "red" and FILTER_FOR_ANCESTORS is set to true, only the red wine attribute value is returned. When set to FALSE, then both attribute values are returned. The default value is FALSE.

RELRANK_STRATEGY

Specifies the name of a relevance ranking strategy for value search.

Sub-elements

The following table provides a brief overview of the DIMSEARCH_CONFIG sub-elements.

Sub-element	Brief description
COMMENT	Associates a comment with a parent element and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.
PARTIAL_MATCH	Specifies if partial query matches should be supported for the dimension.

Example

This example shows a configuration that displays ancestor attribute values.

```
<DIMSEARCH_CONFIG FILTER_FOR_ANCESTORS="FALSE"/>
```

Recsearch_config elements

The Recsearch_config element configures record search.

RECSEARCH_CONFIG

A RECSEARCH_CONFIG element sets up the configuration of attributes for record searches.

Record searches search against the text collection that consists of the names of all the attribute values in the data set.

Format

```
<!ELEMENT RECSEARCH_CONFIG
    ( COMMENT?
    , SEARCH_INTERFACE*
    )
>
<!ATTLIST RECSEARCH_CONFIG
    WORD_INTERP    (TRUE | FALSE)    "FALSE"
>
```

Attributes

The RECSEARCH_CONFIG element has the following attributes.

WORD_INTERP

Specifies whether to enable word interpretation forms (see-also suggestions) of user query terms considered by the text search engine while processing record search requests. The default value is FALSE.

Sub-elements

The following table provides a brief overview of the RECSEARCH_CONFIG sub-elements.

Sub-element	Brief description
COMMENT	Associates a comment with a parent element and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.
SEARCH_INTERFACE	Represents a named collection of dimensions and/or properties.

Example

This example shows the configuration for a business implementation.

```
<RECSEARCH_CONFIG>
   <SEARCH_INTERFACE CROSS_FIELD_BOUNDARY="NEVER"
      CROSS_FIELD_RELEVANCE_RANK="0"
      DEFAULT_RELRANK_STRATEGY="All" NAME="All">
     <MEMBER_NAME RELEVANCE_RANK="4">ProductType</MEMBER_NAME>
     <MEMBER_NAME RELEVANCE_RANK="3">Name</MEMBER_NAME>
     <MEMBER_NAME RELEVANCE_RANK="2">Region</MEMBER_NAME>
     <MEMBER_NAME RELEVANCE_RANK="1">Description</MEMBER_NAME>
   </SEARCH_INTERFACE>
</RECSEARCH_CONFIG>
```

Relrank_strategies elements

The Relrank_strategies elements contain the relevance ranking strategies for an application.

The strategies are grouped in the root element RELRANK_STRATEGIES. Each strategy is expressed in a RELRANK_STRATEGY element, which in turn is made of individual relevance ranking modules such as RELRANK_EXACT, RELRANK_FIELD, and so on.

For more information about relevance ranking, see the *Latitude Developer's Guide*.

RELRANK_APPROXPHRASE

The RELRANK_APPROXPHRASE element implements the Approximate Phrase relevance ranking module.

This module is similar to RELRANK_PHRASE, except that in the higher stratum, only the first instance of an exact match of the user's phrase is considered, which improves system performance.

Note: The RELRANK_APPROXPHRASE element is no longer supported. Use the RELRANK_PHRASE element with the APPROXIMATE attribute instead.

Format

```
<!ELEMENT RELRANK_APPROXPHRASE EMPTY>
```

Attributes

The RELRANK_APPROXPHRASE element has no attributes.

Sub-elements

The RELRANK_APPROXPHRASE element has no sub-elements.

RELRANK_EXACT

The RELRANK_EXACT element implements the Exact relevance ranking module.

This module groups results into strata based on how well they match a query string, with the highest stratum containing results that match the user's query exactly. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_EXACT EMPTY>
```

Attributes

The RELRANK_EXACT element has no attributes.

Sub-elements

The RELRANK_EXACT element has no sub-elements.

Example

In this example, the ranking strategy MyStrategy includes the RELRANK_EXACT element.

```
<RELRANK_STRATEGY NAME="MyStrategy">
   <RELRANK_STATIC NAME="Availability" ORDER="DESCENDING"/>
   <RELRANK_EXACT/>
```

```
   <RELRANK_STATIC NAME="Price" ORDER="ASCENDING"/>
</RELRANK_STRATEGY>
```

RELRANK_FIELD

The RELRANK_FIELD element implements the Field relevance ranking module.

This module assigns a score to each result based on the static rank of the standard attribute or managed attribute member of the search interface that caused the document to match the query. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_FIELD EMPTY>
```

Attributes

The RELRANK_FIELD element has no attributes.

Sub-elements

The RELRANK_FIELD element has no sub-elements.

Example

In this example, the field module is included in a strategy called All_Fields.

```
<RELRANK_STRATEGY NAME="All_Fields">
   <RELRANK_EXACT/>
   <RELRANK_INTERP/>
   <RELRANK_FIELD/>
</RELRANK_STRATEGY>
```

RELRANK_FIRST

The RELRANK_FIRST element implements the First relevance ranking module.

This module ranks documents by how close the query terms are to the beginning of the document. This module takes advantage of the fact that the closer something is to the beginning of a document, the more likely it is to be relevant. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_FIRST EMPTY>
```

Attributes

The RELRANK_FIRST element has no attributes.

Sub-elements

The RELRANK_FIRST element has no sub-elements.

Example

In this example, the ranking strategy All includes the First relevance ranking module.

```
<RELRANK_STRATEGY NAME="All">
   <RELRANK_FIRST/>
   <RELRANK_INTERP/>
   <RELRANK_FIELD/>
</RELRANK_STRATEGY>
```

RELRANK_FREQ

The RELRANK_FREQ element implements the Frequency relevance ranking module.

This module provides result scoring based on the frequency (number of occurrences) of the user's query terms in the result text. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_FREQ EMPTY>
```

Attributes

The RELRANK_FREQ element has no attributes.

Sub-elements

The RELRANK_FREQ element has no sub-elements.

Example

This example implements a strategy called Frequency.

```
<RELRANK_STRATEGY NAME="Frequency">
   <RELRANK_FREQ/>
</RELRANK_STRATEGY>
```

RELRANK_GLOM

The RELRANK_GLOM element implements the Glom relevance ranking module.

This module ranks single-field matches ahead of cross-field matches. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_GLOM EMPTY>
```

Attributes

The RELRANK_GLOM element has no attributes.

Sub-elements

The RELRANK_GLOM element has no sub-elements.

Example

This example implements a strategy called Single_Field.

```
<RELRANK_STRATEGY NAME="Single_Field">
   <RELRANK_GLOM/>
</RELRANK_STRATEGY>
```

RELRANK_INTERP

The RELRANK_INTERP element implements the Interpreted (Interp) relevance ranking module.

This module provides a general-purpose strategy that assigns a score to each result document based on the query processing techniques used to obtain the match. Matching techniques considered include partial matching, cross-attribute matching, spelling correction, thesaurus, and stemming matching. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_INTERP EMPTY>
```

Attributes

The RELRANK_INTERP element has no attributes.

Sub-elements

The RELRANK_INTERP element has no sub-elements.

Example

In this example, the Interpreted module is included in a strategy called All_Fields.

```
<RELRANK_STRATEGY NAME="All_Fields">
   <RELRANK_EXACT/>
   <RELRANK_INTERP/>
   <RELRANK_FIELD/>
</RELRANK_STRATEGY>
```

RELRANK_MAXFIELD

The RELRANK_MAXFIELD element implements the Maximum Field (Maxfield) relevance ranking module.

This module is similar to the Field strategy module, except it selects the static field-specific score of the highest-ranked field that contributed to the match. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_MAXFIELD EMPTY>
```

Attributes

The RELRANK_MAXFIELD element has no attributes.

Sub-elements

The RELRANK_MAXFIELD element has no sub-elements.

Example

This example implements a strategy called High_Rank.

```
<RELRANK_STRATEGY NAME="High_Rank">
   <RELRANK_MAXFIELD/>
</RELRANK_STRATEGY>
```

RELRANK_MODULE

The RELRANK_MODULE element is used to refer to and compose other relevance ranking modules into strategies.

Format

```
<!ELEMENT RELRANK_MODULE (RELRANK_MODULE_PARAM*)>
<!ATTLIST RELRANK_MODULE
     NAME     CDATA     #REQUIRED
>
```

Attributes

The RELRANK_MODULE element has the following attribute.

NAME

NAME refers to another defined relevance ranking module.

Sub-elements

The RELRANK_MODULE element has no supported sub-elements. RELRANK_MODULE_PARAM is not supported.

Example

In this example, a strategy called Best Price is defined. Later, this strategy is included in another strategy definition using the RELRANK_MODULE element.

```
<RELRANK_STRATEGY NAME="Best Price">
   <RELRANK_STATIC NAME="Price" ORDER="ASCENDING"/>
</RELRANK_STRATEGY>
<RELRANK_STRATEGY NAME="MyStrategy">
   <RELRANK_STATIC NAME="Availability" ORDER="DESCENDING"/>
   <RELRANK_EXACT/>
   <RELRANK_MODULE NAME="Best Price"/>
</RELRANK_STRATEGY>
```

RELRANK_NTERMS

The RELRANK_NTERMS element implements the Number of Terms (Nterms) relevance ranking module.

This module assigns a score to each result record based on the number of query terms that the result record matches. For example, in a three-word query, results that match all three words are ranked

above results that match only two words, which are ranked above results that match only one word. For details, see the *Latitude Developer's Guide*.

This module applies only to search modes where the number of results can vary in how many query terms they match. These search modes include matchpartial, matchany, matchallpartial, and matchallany. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_NTERMS EMPTY>
```

Attributes

The RELRANK_NTERMS element has no attributes.

Sub-elements

The RELRANK_NTERMS element has no sub-elements.

Example

In this example, the Nterms module is included in a strategy called NumberOfTerms.

```
<RELRANK_STRATEGY NAME="NumberOfTerms">
   <RELRANK_NTERMS/>
</RELRANK_STRATEGY>
```

RELRANK_NUMFIELDS

The RELRANK_NUMFIELDS element implements the Number of Fields (Numfields) relevance ranking module.

This module ranks results based on the number of fields in the associated search interface in which a match occurs. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_NUMFIELDS EMPTY>
```

Attributes

The RELRANK_NUMFIELDS element has no attributes.

Sub-elements

The RELRANK_NUMFIELDS element has no sub-elements.

Example

This example implements the Numfields relevance ranking module.

```
<RELRANK_STRATEGY NAME="NumFields">
   <RELRANK_NUMFIELDS/>
</RELRANK_STRATEGY>
```

RELRANK_PHRASE

The RELRANK_PHRASE element implements the Phrase relevance ranking module.

This module states that results containing the user's query as an exact phrase, or a subset of the exact phrase, should be considered more relevant than matches simply containing the user's search terms scattered throughout the text. Note that records that have the phrase are ranked higher than records which do not contain the phrase. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_PHRASE EMPTY>
<!ATTLIST RELRANK_PHRASE
    SUBPHRASE        (TRUE | FALSE)   "FALSE"
    APPROXIMATE      (TRUE | FALSE)   "FALSE"
    QUERY_EXPANSION  (TRUE | FALSE)   "FALSE"
>
```

Attributes

The RELRANK_PHRASE element has the following attributes.

SUBPHRASE

If set to TRUE, enables subphrasing, which ranks results based on the length of their subphrase matches.

If set to FALSE (the default), subphrasing is not enabled, which means that results are ranked into two strata: those that matched the entire phrase and those that did not.

APPROXIMATE

If set to TRUE, approximate matching is enabled. In this case, the Phrase module looks at a limited number of positions in each result that a phrase match could possibly exist, rather than all the positions. Only this limited number of possible occurrences is considered, regardless of whether there are later occurrences that are better, more relevant matches.

QUERY_EXPANSION

If set to TRUE, enables query expansion, in which spelling correction, thesaurus, and stemming adjustments are applied to the original phrase. With query expansion enabled, the Phrase module ranks results that match a phrase's expanded forms in the same stratum as results that match the original phrase.

Sub-elements

The RELRANK_PHRASE element has no sub-elements.

Example

This example of the Phrase module enables approximate matching and query expansion, and disables subphrasing.

```
<RELRANK_STRATEGY NAME="PhraseMatch">
   <RELRANK_PHRASE APPROXIMATE="TRUE"
     QUERY_EXPANSION="TRUE" SUBPHRASE="FALSE"/>
</RELRANK_STRATEGY>
```

RELRANK_PROXIMITY

The RELRANK_PROXIMITY element implements the Proximity relevance ranking module.

This module ranks how close the query terms are to each other in a document by counting the number of intervening words. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_PROXIMITY EMPTY>
```

Attributes

The RELRANK_PROXIMITY element has no attributes.

Sub-elements

The RELRANK_PROXIMITY element has no sub-elements.

Example

This example implements a strategy called All that includes the Proximity module.

```
<RELRANK_STRATEGY NAME="All">
   <RELRANK_PROXIMITY/>
   <RELRANK_INTERP/>
   <RELRANK_FIELD/>
</RELRANK_STRATEGY>
```

RELRANK_SPELL

The RELRANK_SPELL element implements the Spell relevance ranking module.

This module ranks matches that do not require spelling correction ahead of spelling-corrected matches. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_SPELL EMPTY>
```

Attributes

The RELRANK_SPELL element has no attributes.

Sub-elements

The RELRANK_SPELL element has no sub-elements.

Example

This example implements a strategy called TrueMatch.

```
<RELRANK_STRATEGY NAME="TrueMatch">
   <RELRANK_SPELL/>
</RELRANK_STRATEGY>
```

RELRANK_STATIC

The RELRANK_STATIC element implements the Static relevance ranking module.

This module assigns a constant score to each result, depending on the type of search operation performed. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_FREQ EMPTY>
<!ATTLIST RELRANK_STATIC
    NAME    CDATA                         #REQUIRED
    ORDER   (ASCENDING|DESCENDING)   #REQUIRED
>
```

Attributes

The RELRANK_STATIC element has the following attributes.

NAME

Specifies the name of a standard or managed attribute that is used for static relevance ranking.

ORDER

Specifies how records should be sorted with respect to the specified standard or managed attribute.

Sub-elements

The RELRANK_STATIC element has no sub-elements.

Example

In this example, the BestPrice strategy consists of the Price managed attribute sorted from lowest to highest.

```
<RELRANK_STRATEGY NAME="BestPrice">
   <RELRANK_STATIC NAME="Price" ORDER="ASCENDING"/>
</RELRANK_STRATEGY>
```

RELRANK_STRATEGIES

A RELRANK_STRATEGIES element contains any number of relevance ranking strategies for an application.

Each strategy is specified in a RELRANK_STRATEGY element.

Format

```
<!ELEMENT RELRANK_STRATEGIES
    ( COMMENT?
    , RELRANK_STRATEGY*
    )
>
```

Attributes

The RELRANK_STRATEGIES element has no attributes.

Sub-elements

The following table provides a brief overview of the RELRANK_STRATEGIES sub-elements.

Sub-element	Brief description
COMMENT	Associates a comment with a parent element and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.
RELRANK_STRATEGY	Contains a list of relevance ranking strategies that affect the order in which search results are returned to a user.

Example

This example shows several strategies grouped under the root element RELRANK_STRATEGIES.

```
<RELRANK_STRATEGIES>
   <RELRANK_STRATEGY NAME="Bestseller Strategy">
     <RELRANK_STATIC NAME="Bestseller" ORDER="DESCENDING"/>
   </RELRANK_STRATEGY>
   <RELRANK_STRATEGY NAME="Electronics Strategy">
     <RELRANK_FIELD/>
     <RELRANK_EXACT/>
     <RELRANK_INTERP/>
     <RELRANK_STATIC NAME="Bestseller" ORDER="DESCENDING"/>
     <RELRANK_STATIC NAME="Product_Name" ORDER="ASCENDING"/>
   </RELRANK_STRATEGY>
</RELRANK_STRATEGIES>
```

RELRANK_STRATEGY

The RELRANK_STRATEGY element contains a list of relevance ranking strategies that affect the order in which search results are returned to a user.

Each sub-element of RELRANK_STRATEGY represents a specific type of strategy. If you want several relevance ranking strategies to affect search result, then the order of the sub-elements, which represent the strategies, is significant. The order of the sub-elements defines the order in which the strategies are applied to the search results. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_STRATEGY (
      RELRANK_STATIC
    | RELRANK_EXACT
    | RELRANK_PHRASE
    | RELRANK_APPROXPHRASE
    | RELRANK_GLOM
    | RELRANK_SPELL
    | RELRANK_FIELD
    | RELRANK_MAXFIELD
    | RELRANK_INTERP
    | RELRANK_FREQ
    | RELRANK_WFREQ
    | RELRANK_NTERMS
    | RELRANK_PROXIMITY
    | RELRANK_FIRST
    | RELRANK_NUMFIELDS
    | RELRANK_MODULE
```

```
    )+>
<!ATTLIST RELRANK_STRATEGY
    NAME    CDATA    #REQUIRED
>
```

Attributes

The RELRANK_STRATEGY element has the following attribute.

NAME

Specifies the name of the strategy.

Sub-elements

The following table provides a brief overview of the RELRANK_STRATEGY sub-elements.

Sub-element	Brief description
RELRANK_STATIC	Assigns a constant score to each result, depending on the type of search operation perform.
RELRANK_EXACT	Groups results into strata based on how well they match the query string, with the highest stratum containing results that match the user's query exactly.
RELRANK_PHRASE	Considers results containing the user’s query as an exact phrase, or a subset of the exact phrase, to be more relevant than matches simply containing the user’s search terms scattered throughout the text.
RELRANK_APPROXPHRASE	Not supported.
RELRANK_GLOM	Ranks single-field matches ahead of cross-field matches.
RELRANK_SPELL	Ranks true matches ahead of spelling-corrected matches.
RELRANK_FIELD	Assigns a score to each result based on the static rank of the dimension or property member of the search interface that caused the document to match the query.
RELRANK_MAXFIELD	Similar to the Field strategy, except it selects the static field-specific score of the highest-ranked field that contributed to the match.
RELRANK_INTERP	A general-purpose strategy that assigns a score to each result document based on the query processing techniques used to obtain the match. Matching techniques considered include partial matching, cross-attribute matching, spelling correction, thesaurus, and stemming matching.
RELRANK_FREQ	Provides result scoring based on the frequency (number of occurrences) of the user's query terms in the result text.
RELRANK_WFREQ	Scores results based on the frequency of user query terms in the result, while weighing the individual query term frequencies for each result by the information content (overall frequency in the complete data set) of each query term.
RELRANK_NTERMS	Assigns a score to each result record based on the number of query terms that the result record matches.

Sub-element	Brief description
RELRANK_PROXIMITY	Ranks how close the query terms are to each other in a document by counting the number of intervening words.
RELRANK_FIRST	Ranks documents by how close the query terms are to the beginning of the document.
RELRANK_NUMFIELDS	Ranks results based on the number of fields in the associated search interface in which a match occurs.
RELRANK_MODULE	Used to refer to other RELRANK elements and compose them into cohesive strategies.

Example

This example presents a ranking strategy called Product_Search_Rank, which itself is composed of multiple strategies.

```
<RELRANK_STRATEGY NAME="Product_Search_Rank">
   <RELRANK_MODULE NAME="IsAvailable"/>
   <RELRANK_FIELD/>
   <RELRANK_PHRASE/>
   <RELRANK_MODULE NAME="BestPrice"/>
</RELRANK_STRATEGY>
```

RELRANK_WFREQ

The RELRANK_WFREQ element implements the Weighted Frequency (Wfreq) relevance ranking module.

This module scores results based on the frequency of user query terms in the result, while weighing the individual query term frequencies for each result by the information content (overall frequency in the complete data set) of each query term. For details, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT RELRANK_WFREQ EMPTY>
```

Attributes

The RELRANK_WFREQ element has no attributes.

Sub-elements

The RELRANK_WFREQ element has no sub-elements.

Example

This example implements a strategy called Term_Freq.

```
<RELRANK_STRATEGY NAME="Term_Freq">
   <RELRANK_WFREQ/>
</RELRANK_STRATEGY>
```

Search_interface elements

The Search_interface elements are used to build and configure search interfaces.

The file's root element is SEARCH_INTERFACE. Search interfaces control record search behavior for groups of standard and managed attributes.

MEMBER_NAME

The MEMBER_NAME element specifies the name of an Endeca standard or managed attribute that is part of a SEARCH_INTERFACE.

For information on search interfaces, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT MEMBER_NAME (#PCDATA)>
<!ATTLIST MEMBER_NAME
    RELEVANCE_RANK    CDATA    #IMPLIED
    SNIPPET_SIZE      CDATA    "0"
>
```

Attributes

The MEMBER_NAME element has the following attributes.

RELEVANCE_RANK

RELEVANCE_RANK is an unsigned integer that specifies the relevance rank of a match on the specified Endeca standard or managed attribute.

SNIPPET_SIZE

The presence of SNIPPET_SIZE enables snippeting for a MEMBER_NAME and the value of SNIPPET_SIZE specifies maximum number of words a snippet can contain. Omitting this attribute or setting its value equal to zero disables snippeting. For more information, see "Using Snippeting in Record Searches" in the *Latitude Developer's Guide*.

Sub-elements

The MEMBER_NAME element has no sub-elements.

Example

In the following example for a search interface named ProductSearch, four Endeca attributes are listed in MEMBER_NAME elements, each with its own relevance rank. A fifth MEMBER_NAME element enables snippeting for the Description attribute.

```
<SEARCH_INTERFACE CROSS_FIELD_BOUNDARY="NEVER"
      CROSS_FIELD_RELEVANCE_RANK="0"
      DEFAULT_RELRANK_STRATEGY="ProductRelRank" NAME="ProductSearch">
   <MEMBER_NAME RELEVANCE_RANK="4">ProductType</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="3">Name</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="2">SalesRegion</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="1">Description</MEMBER_NAME>
   <MEMBER_NAME SNIPPET_SIZE="10">Description</MEMBER_NAME>
</SEARCH_INTERFACE>
```

PARTIAL_MATCH

The PARTIAL_MATCH element specifies if partial query matches should be supported for the SEARCH_INTERFACE that contains this element.

For details about searching and search modes, see the *Latitude Developer's Guide*.

Format

```
<!ELEMENT PARTIAL_MATCH EMPTY>
<!ATTLIST PARTIAL_MATCH
    MIN_WORDS_INCLUDED    CDATA    #IMPLIED
    MAX_WORDS_OMITTED     CDATA    #IMPLIED
>
```

Attributes

The PARTIAL_MATCH element has the following attributes.

MIN_WORDS_INCLUDED

Specifies that search results match at least this number of terms in the search query. This value must be an integer greater than zero. The default value of this attribute is one.

MAX_WORDS_OMITTED

Specifies the maximum number of query terms that may be ignored in the search query. This value must be a non-negative integer. If set to zero or left unspecified, any number of words may be omitted (i.e., there is no maximum). The default value of this attribute is two.

Sub-elements

The PARTIAL_MATCH element has no sub-elements.

Example

In this example, the search interface is subject to partial matching in which at least two of the words in the search query are included, and no more than one is omitted.

```
<SEARCH_INTERFACE CROSS_FIELD_BOUNDARY="ALWAYS"
      CROSS_FIELD_RELEVANCE_RANK="0"
      DEFAULT_RELRANK_STRATEGY="WineRelRank" NAME="WinePartSearch">
   <MEMBER_NAME RELEVANCE_RANK="2">Body</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="1">Description</MEMBER_NAME>
   <PARTIAL_MATCH MAX_WORDS_OMITTED="1" MIN_WORDS_INCLUDED="2"/>
</SEARCH_INTERFACE>
```

SEARCH_INTERFACE

The SEARCH_INTERFACE element is a named collection of Endeca standard attributes and/or managed attributes.

Both standard attributes and managed attributes can co-exist in a SEARCH_INTERFACE. The Endeca attributes in the group are specified in MEMBER_NAME elements.

If a standard attribute or managed attribute is not included in any SEARCH_INTERFACE element, then an implicit SEARCH_INTERFACE element is created with the same name as the standard attribute or managed attribute and that single standard attribute or managed attribute as its only member. The value for the CROSS_FIELD_RELEVANCE_RANK is set to 0.

Format

```
<!ELEMENT SEARCH_INTERFACE
    ( MEMBER_NAME+
    , PARTIAL_MATCH?
    )
>
<!ATTLIST SEARCH_INTERFACE
    NAME                        CDATA        #REQUIRED
    DEFAULT_RELRANK_STRATEGY    CDATA        #IMPLIED
    CROSS_FIELD_RELEVANCE_RANK  CDATA        #IMPLIED
    CROSS_FIELD_BOUNDARY        (ALWAYS
                                |ON_FAILURE
                                |NEVER)      "NEVER"
    STRICT_PHRASE_MATCH         (TRUE|FALSE) #IMPLIED
>
```

Attributes

The SEARCH_INTERFACE element has the following attributes.

NAME

A unique name for this search interface.

DEFAULT_RELRANK_STRATEGY

For record search, a default relevance scoring function assigned to a SEARCH_INTERFACE. For example, if your search interface is called Flavors, the DEFAULT_RELRANK_STRATEGY attribute has the value "Flavors_strategy".

CROSS_FIELD_RELEVANCE_RANK

Specifies the relevance rank score for cross-field matches. The value should be an unsigned 32-bit integer. The default value for CROSS_FIELD_RELEVANCE_RANK is 0.

CROSS_FIELD_BOUNDARY

Specifies when the search engine should try to match search queries across standard attribute/managed attribute boundaries, but within the members of the SEARCH_INTERFACE. If its value is set to ON_FAILURE, then the search engine will only try to match queries across standard attribute/managed attribute boundaries if it fails to find any match within a single standard attribute/managed attribute. If its value is set to ALWAYS, then the engine will always look for matches across standard attribute/managed attribute boundaries, in addition to matches within a standard attribute/managed attribute.

By default, the MDEX Engine will not look across boundaries for matches.

STRICT_PHRASE_MATCH

Specifies that the MDEX Engine should interpret a query strictly when comparing white space in the query with punctuation in the source text. If set to FALSE, partial word tokens connected in the source text by punctuation can be matched to a phrase query where the partial tokens are separated by spaces instead of matching punctuation. The default value of this attribute is TRUE.

Sub-elements

The following table provides a brief overview of the SEARCH_INTERFACE sub-elements.

Sub-element	Brief description
MEMBER_NAME	Specifies the name of a property or dimension that is part of a SEARCH_INTERFACE.
PARTIAL_MATCH	Specifies if partial query matches should be supported for the SEARCH_INTERFACE that contains this element.

Example

This example establishes a search interface called AllFields, which contains four members.

```
<SEARCH_INTERFACE CROSS_FIELD_BOUNDARY="NEVER"
      CROSS_FIELD_RELEVANCE_RANK="0"
      DEFAULT_RELRANK_STRATEGY="All" NAME="AllFields">
   <MEMBER_NAME RELEVANCE_RANK="4">ProductType</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="3">ProductName</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="2">SalesRegion</MEMBER_NAME>
   <MEMBER_NAME RELEVANCE_RANK="1">Description</MEMBER_NAME>
</SEARCH_INTERFACE>
```

Stop_words elements

The Stop_words elements contain words that should be eliminated from a query before it is processed by the MDEX Engine.

Each stop is specified in a STOP_WORD element.

STOP_WORD

The STOP_WORD element identifies words that should be eliminated from a query before it is processed.

Examples of common stop words include the words "the" and "of".

Format

```
<!ELEMENT STOP_WORD (#PCDATA)>
```

Attributes

The STOP_WORD element has no attributes.

Sub-elements

The STOP_WORD element has no sub-elements.

Example

This example shows a common set of stop words.

```
<STOP_WORDS>
   <STOP_WORD>a</STOP_WORD>
   <STOP_WORD>an</STOP_WORD>
   <STOP_WORD>of</STOP_WORD>
```

```
   <STOP_WORD>the</STOP_WORD>
</STOP_WORDS>
```

STOP_WORDS

A STOP_WORDS element specifies the stop words enabled in your application.

Each stop word is represented by a STOP_WORD element.

Format

```
<!ELEMENT STOP_WORDS
    ( COMMENT?
    , STOP_WORD*
    )
>
```

Attributes

The STOP_WORDS element has no attributes.

Sub-elements

The following table provides a brief overview of the STOP_WORDS sub-elements.

Sub-element	Brief description
COMMENT	Associates a comment with a parent element and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.
STOP_WORD	Identifies words that should be eliminated from a query before it is processed.

Example

This example shows a common set of stop words.

```
<STOP_WORDS>
   <STOP_WORD>a</STOP_WORD>
   <STOP_WORD>an</STOP_WORD>
   <STOP_WORD>of</STOP_WORD>
   <STOP_WORD>the</STOP_WORD>
</STOP_WORDS>
```

Thesaurus elements

The Thesaurus elements contain thesaurus entries for your application.

Thesaurus entries provide a means to account for alternate forms of a user's query. These entries provide concept-level mappings between words and phrases. For details, see the *Latitude Developer's Guide*.

THESAURUS

A THESAURUS element contains the term equivalence mappings for an application.

THESAURUS is the root element for all thesaurus entries.

Note that the order of sub-elements within THESAURUS is significant. You should add sub-elements in the order in which they are listed in the format section.

For example, THESAURUS_ENTRY sub-elements appear before THESAURUS_ENTRY_ONEWAY. See the example below.

Format

```
<!ELEMENT THESAURUS
    ( COMMENT?
    , THESAURUS_ENTRY*
    , THESAURUS_ENTRY_ONEWAY*
    )
>
```

Attributes

The THESAURUS element has no attributes.

Sub-elements

The following table provides a brief overview of the THESAURUS sub-elements.

Sub-element	Brief description
COMMENT	Associates a comment with a parent element and preserves the comment when the file is rewritten. This element provides an alternative to using inline XML comments of the form <!-- ... -->.
THESAURUS_ENTRY	Indicates a set of word forms (contained in THESAURUS_FORM elements) that are equivalent.
THESAURUS_ENTRY_ONEWAY	Specifies single-direction equivalency mappings.

Example

This example shows the thesaurus entries for an application.

```
<THESAURUS>
   <THESAURUS_ENTRY>
     <THESAURUS_FORM>france</THESAURUS_FORM>
     <THESAURUS_FORM>french</THESAURUS_FORM>
   </THESAURUS_ENTRY>
   <THESAURUS_ENTRY_ONEWAY>
     <THESAURUS_FORM_FROM>Red wine</THESAURUS_FORM_FROM>
     <THESAURUS_FORM_TO>Merlot</THESAURUS_FORM_TO>
     <THESAURUS_FORM_TO>Shiraz</THESAURUS_FORM_TO>
     <THESAURUS_FORM_TO>Bordeaux</THESAURUS_FORM_TO>
   </THESAURUS_ENTRY_ONEWAY>
</THESAURUS>
```

THESAURUS_ENTRY

The THESAURUS_ENTRY element indicates a set of word forms that are equivalent.

The word forms are contained in THESAURUS_FORM elements. A search for any of these forms (including stemming-matched versions) returns hits for all of the forms.

Format

```
<!ELEMENT THESAURUS_ENTRY (THESAURUS_FORM+)>
```

Attributes

The THESAURUS_ENTRY element has no attributes.

Sub-elements

The following table provides a brief overview of the THESAURUS_ENTRY sub-element.

Sub-element	Brief description
THESAURUS_ENTRY	Indicates a set of word forms that are equivalent.

Example

In this example, the noun and adjective forms of a word are made equivalent.

```
<THESAURUS>
   <THESAURUS_ENTRY>
     <THESAURUS_FORM>france</THESAURUS_FORM>
     <THESAURUS_FORM>french</THESAURUS_FORM>
   </THESAURUS_ENTRY>
</THESAURUS>
```

THESAURUS_ENTRY_ONEWAY

A THESAURUS_ENTRY_ONEWAY element specifies a single-direction mapping.

Searches for any of the "from" forms (THESAURUS_FORM_FROM elements) also return hits for all of the "to" forms (THESAURUS_FORM_TO elements). The other direction is not enabled; that is, searches for the "to" forms do not return results for either the "from" forms or the other "to" forms.

Format

```
<!ELEMENT THESAURUS_ENTRY_ONEWAY
    ( THESAURUS_FORM_FROM
    , THESAURUS_FORM_TO+
    )
>
```

Attributes

The THESAURUS_ENTRY_ONEWAY element has no attributes.

Sub-elements

The following table provides a brief overview of the THESAURUS_ENTRY_ONEWAY sub-elements.

Sub-element	Brief description
THESAURUS_FORM_FROM	Specifies the "from" form in a one-way word mapping.
THESAURUS_FORM_TO	Specifies the "to" form in a one-way word mapping.

Example

In this example, searches for `Red wine` would return hits for `Red wine` as well as for `Merlot`, `Shiraz`, and `Bordeaux`. Since the equivalence is one-way, more specific searches such as `Shiraz` or `Bordeaux` would not return results for the more general concept `Red wine`.

```
<THESAURUS_ENTRY_ONEWAY>
  <THESAURUS_FORM_FROM>Red wine</THESAURUS_FORM_FROM>
  <THESAURUS_FORM_TO>Merlot</THESAURUS_FORM_TO>
  <THESAURUS_FORM_TO>Shiraz</THESAURUS_FORM_TO>
  <THESAURUS_FORM_TO>Bordeaux</THESAURUS_FORM_TO>
</THESAURUS_ENTRY_ONEWAY>
```

THESAURUS_FORM

The THESAURUS_FORM element contains a word form that is used by the THESAURUS_ENTRY element to set an equivalence.

Format

```
<!ELEMENT THESAURUS_FORM (#PCDATA)>
```

Attributes

The THESAURUS_FORM element has no attributes.

Sub-elements

The THESAURUS_FORM element has no sub-elements.

Example

In this example, the noun and adjective forms of a word are made equivalent.

```
<THESAURUS>
   <THESAURUS_ENTRY>
     <THESAURUS_FORM>france</THESAURUS_FORM>
     <THESAURUS_FORM>french</THESAURUS_FORM>
   </THESAURUS_ENTRY>
</THESAURUS>
```

THESAURUS_FORM_FROM

The THESAURUS_FORM_FROM element provides the "from" form within a THESAURUS_ENTRY_ONEWAY element.

Format

```
<!ELEMENT THESAURUS_FORM_FROM (#PCDATA)>
```

Attributes

The THESAURUS_FORM_FROM element has no attributes.

Sub-elements

The THESAURUS_FORM_FROM element has no sub-elements.

Example

In this example, searches for `home theater` would return hits for `home theater` as well as for `stereo` and `television`. Because the equivalence is one-way, more specific searches such as `stereo` or `television` would not return results for the more general concept `home theater`.

```
<THESAURUS_ENTRY_ONEWAY>
   <THESAURUS_FORM_FROM>home theater</THESAURUS_FORM_FROM>
   <THESAURUS_FORM_TO>stereo</THESAURUS_FORM_TO>
   <THESAURUS_FORM_TO>television</THESAURUS_FORM_TO>
</THESAURUS_ENTRY_ONEWAY>
```

THESAURUS_FORM_TO

The THESAURUS_FORM_TO element provides the "to" form within a THESAURUS_ENTRY_ONEWAY element.

Format

```
<!ELEMENT THESAURUS_FORM_TO (#PCDATA)>
```

Attributes

The THESAURUS_FORM_TO element has no attributes.

Sub-elements

The THESAURUS_FORM_TO element has no sub-elements.

Example

In this example, searches for `home theater` would return hits for `home theater` as well as for `stereo` and `television`. Because the equivalence is one-way, more specific searches such as `stereo` or `television` would not return results for the more general concept `home theater`.

```
<THESAURUS_ENTRY_ONEWAY>
   <THESAURUS_FORM_FROM>home theater</THESAURUS_FORM_FROM>
   <THESAURUS_FORM_TO>stereo</THESAURUS_FORM_TO>
   <THESAURUS_FORM_TO>television</THESAURUS_FORM_TO>
</THESAURUS_ENTRY_ONEWAY>
```

Index

A

Add KVPs connector
  configuration properties 151
  configuring for key-value pair loads 117
  enabling SSL 152
  reference details 150
Add Managed Values connector
  adding to graph 124
  configuration properties 153
  configuring for taxonomy loads 125
  enabling SSL 154
  reference details 152
Add/Update Records connector
  adding to graph 56
  configuration properties 149
  configuring for incremental updates 57
  configuring for PDR output 72
  enabling SSL 149
  reference details 148
attribute schema
  configuration input file 62
  managed attributes input file 75
attribute schema load
  about 61
  loading DDRs 75
  loading PDRs 61

B

baseline update, See full index load
Begin Transaction graph 33
BufferOverflow errors, avoiding 170
Bulk Add/Replace Records connector
  adding to graph 46
  configuration properties 146
  configuring for full index load 53
  enabling SSL 147
  reference details 145

C

COMMENT element 175
Commit Transaction graph 34
Common configuration properties for Latitude connectors 165
configuration and schema
  importing and exporting 129
configuration document loads, See index configuration loads
Custom properties, creating 20

D

data types, supported 25
DDRs, loading, See loading DDRs
Delete Data connector
  adding to graph 140
  configuration properties 155
  configuring for delete operation 142
  enabling SSL 155
  reference details 154
  types of delete operations 139
deleting data
  adding components to graph 140
  configuring Delete Data connector 142
  configuring metadata 141
  configuring Reader component 141
  running the graph 143
  source data format 139
Designer GUI overview, LDI 11
DIMNAME element 176
Dimsearch_config
  about 177
  DIMSEARCH_CONFIG element 178
DIMSEARCH_CONFIG element 178

E

Edge component
  configuring for deleting data 141
  configuring for full index load 48
  configuring for loading index configuration 93
  configuring for loading PDRs 64
  configuring for loading taxonomy 126
  configuring for transaction graph 36
Enabled configuration property for Latitude connectors 167
enabling SSL 29
Export Config connector
  configuration properties 157
  configuring in a graph 131
  reference details 156
exporting
  configuration and schema 129
externally managed taxonomies, loading 121

F

full index load
  configuring Bulk Add/Replace Records connector 53
  configuring Edge component 48
  configuring Reader component 47
  creating graph 45

full index load *(continued)*
overview 41
running graph 54
source data format 43

G

Global Configuration Record, loading 101
graph
adding Add Managed Values connector 124
adding Add/Update Records connector 56
adding Bulk Add/Replace Records connector 46
adding Delete Data connector 140
adding UniversalDataReader component 46
adding WebServiceClient component 91
creating empty 45
running for full index load 54
running to add key-value pairs 119
running to delete data 143
running to load incremental updates 58
running to load taxonomy 127

I

Import Config connector
configuration properties 158
configuring in a graph 137
reference details 157
importing
configuration and schema 129
incremental updates
overview 55
running graph 58
using the Add/Update Records connector 57
index configuration loads
about 90
adding components to graph 91
configuring Reader component 92
configuring the Edge component 93
creating graph 91
using WebServiceClient component 99

J

Java heap space errors, avoiding 169

K

key-value pair loads
about 115
configuring Add KVPs connector 117
configuring Edge metadata for graph 118
configuring Reader for graph 116
input format 115
running graph 119
KVP loads, See key-value pair loads

L

Latitude connectors
Add KVPs 150
Add Managed Values 152
Add/Update Records 148
Bulk Add/Replace Records 145
Delete Data 154
Export Config 156
Import Config 157
overview 12
Reset MDEX 159
TransactionRunGraph 161
Visual and Common configuration properties 165
Latitude Data Integrator
about the Server 14
additional documentation 29
creating empty graph 45
creating projects 42
overview of Designer 11
product overview
LDI, See Latitude Data Integrator
loading DDRs
configuring Reader and Edge components 77
overview 75
loading managed attribute metadata
configuring WebServiceClient component 81
loading PDRs
configuring Add/Update Records connector 72
configuring Edge component 64
configuring Reader component 64
overview 61

M

managed attribute name for taxonomy, specifying 125
managed attributes, default values for 28
managed values, loading 121
mdexType Custom properties, creating 20
MEMBER_NAME element 192
metadata
configuring for deleting data 141
configuring for full index load 48
configuring for loading index configuration 93
configuring for loading PDRs 64
configuring for loading taxonomy 126
configuring for transaction graph 36
supported data types 25
multi-assign data
about 44
configuring for Add/Update Records connector 58
configuring for Bulk Add/Replace Records connector 54
errors from misconfiguration 174

O

order of loading data 20

outer transaction
about 31
when to use in the LDI project 18
OutOfMemory errors, avoiding 169

P

PARTIAL_MATCH element 193
PDRs, loading 61
Phase configuration property for Latitude connectors 166
precedence rules
configuration input file 105
configuring Reader component 106
precedence_rules
about 103
primary key
about 44
configuring for Add/Update Records connector 58
configuring for Bulk Add/Replace Records connector 54
project, creating 42
PROP element 176
PROPNAME element 177
PVAL element 177

R

record schema load, See attribute schema load
record spec property, See primary key
Recsearch_config
about 178
RECSEARCH_CONFIG element 179
RECSEARCH_CONFIG element 179
RELRANK_APPROXPHRASE element 180
RELRANK_EXACT element 180
RELRANK_FIELD element 181
RELRANK_FIRST element 181
RELRANK_FREQ element 182
RELRANK_GLOM element 182
RELRANK_INTERP element 183
RELRANK_MAXFIELD element 183
RELRANK_MODULE element 184
RELRANK_NTERMS element 184
RELRANK_NUMFIELDS element 185
RELRANK_PHRASE element 186
RELRANK_PROXIMITY element 187
RELRANK_SPELL element 187
RELRANK_STATIC element 188
Relrank_strategies
about 180
RELRANK_APPROXPHRASE element 180
RELRANK_EXACT element 180
RELRANK_FIELD element 181
RELRANK_FIRST element 181
RELRANK_FREQ element 182
RELRANK_GLOM element 182
RELRANK_INTERP element 183
RELRANK_MAXFIELD element 183
RELRANK_MODULE element 184
Relrank_strategies *(continued)*
RELRANK_NTERMS element 184
RELRANK_NUMFIELDS element 185
RELRANK_PHRASE element 186
RELRANK_PROXIMITY element 187
RELRANK_SPELL element 187
RELRANK_STATIC element 188
RELRANK_STRATEGIES element 188
RELRANK_STRATEGY element 189
RELRANK_WFREQ element 191
RELRANK_STRATEGIES element 188
RELRANK_STRATEGY element 189
RELRANK_WFREQ element 191
Reset MDEX connector
configuration properties 160
reference details 159
Rollback Transaction graph 34

S

Search_interface
about 192
MEMBER_NAME element 192
PARTIAL_MATCH element 193
SEARCH_INTERFACE element 193
SEARCH_INTERFACE element 193
Server, overview of Latitude Data Integrator 14
source data format
attribute schema 62
deleting data 139
full index load 43
incremental updates 55
key-value pair loads 115
managed attribute schema 75
precedence rules 105
taxonomy loads 122
transaction graph 35
SSL enablement 29
Add KVPs connector 152
Add Managed Values connector 154
Add/Update Records connector 149
Bulk Add/Replace Records connector 147
Delete Data connector 155
Export Config connector 157
Import Config connector 159
Reset MDEX connector 161
Transaction RunGraph connector 164
standard attributes, default values for 27
STOP_WORD element 195
Stop_words
about 195
STOP_WORD element 195
STOP_WORDS element 196
STOP_WORDS element 196

T

taxonomy loads
configuring Add Managed Values connector 125

taxonomy loads *(continued)*
configuring Reader component 124
creating graph 123
metadata configuration 126
overview 121
running graph 127
source input file 122
specifying managed attribute name 125
Thesaurus
about 196
THESAURUS element 197
THESAURUS_ENTRY element 198
THESAURUS_ENTRY_ONEWAY element 198
THESAURUS_FORM element 199
THESAURUS_FORM_FROM element 200
THESAURUS_FORM_TO element 200
THESAURUS element 197
THESAURUS_ENTRY element 198
THESAURUS_ENTRY_ONEWAY element 198
THESAURUS_FORM element 199
THESAURUS_FORM_FROM element 200
THESAURUS_FORM_TO element 200
transaction
about 31
performance impact 40
requirements 31
transaction errors, avoiding 172
transaction graph
creating 34
steps input file 35
Transaction RunGraph connector
configuration properties 163
configuring 37
TransactionRunGraph connector 161
transactions
configuring Edge for transaction graph 36
configuring Reader component for transaction graph 36
making existing graphs run within them 32
manually committing 39

U

UniversalDataReader component
adding to full index load graph 46
configuring for DDR input 77
configuring for deleting data 141
configuring for full index load 47
configuring for index configuration input 92
configuring for PDR input 64
configuring for precedence rules 106
configuring for taxonomy loads 124
configuring for transaction graph input 36

V

Visual configuration properties for Latitude connectors 165

W

WebServiceClient component
adding to graph 91
configuring for managed attributes 81
using for index configuration loads 99
workspace.prm
MDEX_TRANSACTION_ID 25
parameters specific to the MDEX Engine 25
specifying an outer transaction ID 19

X

XML elements
COMMENT 175
DIMNAME 176
PROP 176
PROPNAME 177
PVAL 177