Oracle Endeca Platform Services

Relationship Discovery Guide

Version 6.1.3 • June 2012

Contents

Copyright and disclaimer

Preface

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

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

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

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

About this guide

This guide describes the tasks involved in creating an Endeca Relationship Discovery application.

Who should use this guide

This guide is intended for developers responsible for creating an Endeca Relationship Discovery implementation.

Conventions used in this guide

This guide uses the following typographical conventions:

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

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

Contacting Oracle Support

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

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

Chapter 1

Introduction to Term Discovery

This section provides overviews of the Endeca Term Discovery and Endeca Cluster Discovery features.

Overview of Relationship Discovery

The Endeca Relationship Discovery feature provides you with the ability to identify and extract key relationships in documents, including documents consisting of unstructured text.

This guide describes how to implement the Endeca Term Discovery and Endeca Cluster Discovery features, which are two major components of the Relationship Discovery solution. The third major component, Endeca Entity Discovery, is not documented in this guide; for information on this feature, contact your Endeca representative.

Note: The implementation of Term Discovery and Cluster Discovery requires that you must have purchased a license for Relationship Discovery. The license package includes documentation that describes how to enable the feature. Contact your Endeca representative if you are not certain whether you have a Relationship Discovery license.

The Term Discovery and Cluster Discovery components of Relationship Discovery have the capability to:

- Extract salient terms (noun phrases) from documents.
- Provide a scoring mechanism for the extracted terms, which determines whether a given term is retained. Note that term scores are for internal use by the Endeca software and therefore not exposed.
- Generate relevant terms on queries.
- Generate clusters of relevant terms.

Note that Term Discovery is currently supported only for English and French. All documentation in this guide applies to both versions, unless otherwise noted.

Overview of Term Discovery

Term Discovery is the feature that extracts salient terms from source documents.

Term Discovery can be thought of as a two-part process:

1. Extracting terms from source documents (unstructured or structured) and scoring them according to their relevancy. The scored terms are mapped to an Endeca dimension, called a *Term Discovery dimension* in this guide.
2. Presenting terms relevant to the current navigation state.

Extracting terms from documents

Term extraction is the process of tagging an Endeca record with a list of its relevant terms. A *term* represents a concept mentioned in the record's source document, and is typically a noun phrase. The noun phrase consists of one or more nouns, potentially with adjoining words. A relevant term is a term that bears information for a document relative to the rest of the corpus.

During the term extraction process, term variants found in documents are stemmed for comparison and aggregation, but the final representation of the term uses the dominant form (most frequent variant). Using the dominant form allows the recovery of the preferred representation (singular/plural case, capitalization) of proper nouns and brand names.

Term extraction is performed by the Data Foundry via a Java manipulator pipeline component that uses the Endeca TermExtractor class. The terms are extracted into a user-specified property on the Endeca record. The property is then mapped (via a property mapper) to a dimension. Such a dimension is called a Term Discovery dimension in this guide.

Configuration information on term extraction is given in Chapter 2 ("Configuration Guidelines for Term Extraction").

Maximum size of extracted terms

A noun phrase consists of a noun (or a sequence of nouns) with any associated modifiers. The modifiers are limited to adjectives, adjective phrases, or nouns used as adjectives.

Programmatically, each word in a noun phrase is called a token. An extracted noun phrase can have a maximum of 5 tokens; each token is limited to 200 characters. Therefore, a valid noun phrase has a maximum size of 1,000 characters.

The maximum size of a sentence in a document is 1,000 tokens (words). If the term extractor cannot determine the sentence boundaries of text in the document, it splits the text into blocks of 1,000 tokens and then performs text extraction on the blocks. In addition, the following WARN message will be entered in the term extraction log:

```
Sentence boundaries could not be found for text beginning
with tokens t1 t2 t3 t4 t5
```

where `t1` through `t5` are the first 5 tokens of the problematic text.

The term extractor treats invalid noun phrases as follows:

- If a noun phrase has more than 5 tokens, only the last 5 tokens are retained. For example, with a 7-token noun phrase, the first 2 tokens are completely ignored by the term extractor and the last 5 tokens are retained.
- Tokens that are over 200 characters are ignored.
- If a noun phrase includes an overly-long token, that token is ignored, and the precedent and antecedent tokens are treated as separate noun phrases. For example, assume a 5-token noun phrase. Token 3 is an overly-long token and the others are valid. In this case, Token 3 will be ignored and the term extractor will return 2 noun phrases: the first noun phrase will consist of Tokens 1 and 2, and the second will consist of Tokens 4 and 5.

Presenting relevant terms

Relevant terms are the most frequent terms available in the Term Discovery dimension. These terms are returned from the documents in the current result set. All of the terms in the set belong to the same Term Discovery dimension.

The Term Discovery dimension must have these two attributes:

- It must be a flat dimension (that is, a dimension that does not contain hierarchies).
- It must not be a hidden dimension. (Configuring it as a hidden dimension will disable the Cluster Discovery feature.)

Relevant terms are returned by the Endeca MDEX Engine as dimension value refinements. Programmatically, relevant terms are `DimVal` objects. Therefore, application developers can use the same Endeca Presentation API methods on relevant terms that can be used on normal dimension value refinements. For example, they can be returned sorted using any ranking behavior supported for dimension value refinements.

For more information on displaying relevant terms, see the “Displaying refinements” topic.

Overview of Cluster Discovery

A cluster is a collection of relevant terms, providing a grouping of Endeca records that share these common terms.

All of the terms (which are dimension values) must come from the same dimension, which must be a Term Discovery dimension. Clusters can be generated only if the Term Discovery dimension is available for refinements. So not only can this dimension not be hidden, it must also be available from the navigation states for which clusters are desired. Your application must therefore have this dimension globally available (rather than having it available only when triggered by precedence rules).

The following features apply to the clusters:

- The Endeca MDEX Engine performs dynamic clustering. That is, when a user navigates the clustering tree, it is reclustered at any selection, allowing users to zoom into their data to practically any level.
- There is no limit in the number of records that can be clustered.
- Each cluster is represented by a list of terms, which provide to the user what is known as *information scent*: the user is instantly aware of what each cluster contains (that is, the user can quickly understand the implied content of the clustered records).
- All clusters are designed to maximize two metrics: coherence (each cluster has only closely related records) and distinctiveness (two different clusters will have different records).
- Each cluster has high recall. A match partial technique is typically used on cluster selection, maximizing the number of semantically related records that are returned.

Multiple clusters can be generated from the same dimension. You can configure the maximum number of clusters that can be generated by using the Clusters tab on the Dimensions editor. This tab also allows you to set several parameters for cluster generation. For details, see Chapter 3 (“Configuration Guidelines for Clustering”).

Programmatically, a cluster is a `Supplement` object that accompanies the result of a navigation query. For details on the object’s properties and how to display them, see the “Displaying clusters” topic in Chapter 5.

Chapter 2

Configuration Guidelines for Term Extraction

This topic describes some guidelines for configuring term extraction.

Term extraction workflow

This topics gives an overview of the workflow of the term extractor.

Term extraction consists of three steps, each of which is optional:

1. Candidate Term Identification — Identify all terms that are candidates for a given document and then extract those terms. This step is omitted if terms are being extracted from pre-tagged records (see the “Term filtering with pre-tagged records” topic in Chapter 6).
2. Corpus-level Filtering — Globally filter the extracted terms to determine a controlled vocabulary for the corpus. This involves identifying terms that should be removed corpus-wide (and in step 3 are removed from the set of tags on each record).
3. Record-level Filtering — Determine, for each record, what are the most relevant terms for it from the controlled vocabulary. This involves identifying terms that should be removed from an individual record (independent of terms that should be removed from the entire corpus, but possibly using corpus-level information) and subsequently removing these terms from the tags on the record.

Note that steps 2 and 3 remove the terms that are judged (by their score or by their presence on the exclude list) to be of low information value. The tagged terms on each record are a result of step 3.

This section provides configuration requirements for the term extraction modules. You supply the configuration parameters as pass-through name/value pairs to the Java manipulator.

Configuration for the main term extraction module

The com.endeca.edf.termextractor.TermExtractor class is the main module for the term extraction framework.

You supply this class name in the CLASS_NAME attribute of the Java manipulator. This module also requires configuration parameters that are supplied with PASS_THROUGH elements in the Java manipulator. These parameters are listed in the following table, with details in later sections.

PASS_THROUGH Element	Configuration Value
RECORD_SPEC_PROP_NAME	The source property that is mapped to the Endeca record specifier property. Mandatory. No default.
OUTPUT_PROP_NAME	The source property to use as the destination for tagged terms. Mandatory. No default.
UPDATE_MODE	The type of data update to perform: `STATELESS` (the default), `STATEFUL`, or `PARTIAL`. Optional.
MAX_INPUT_RECORDS	Integer that sets the maximum number of records to be processed. Optional. Defaults to all records processed.

Record specifier property name

The RECORD_SPEC_PROP_NAME pass-through specifies a source record property that is mapped to the Endeca record specifier property in the implementation.

The RECORD_SPEC_PROP_NAME pass-through is mandatory.

To find out the name of the record specifier source property:

1. In Developer Studio, use the Properties view to find out the record specifier property for the implementation. The Record Spec column shows this information.
2. In the Property Mapper editor, use the Mappings dialog box to find out which source property is mapped to the Endeca record specifier property.

Use the name from step 2 for the RECORD_SPEC_PROP_NAME pass-through.

Terms output property

The OUTPUT_PROP_NAME pass-through specifies the property that will hold the tagged terms (if any) for each Endeca record.

The OUTPUT_PROP_NAME pass-through is mandatory. If the property does not already exist in the implementation, then the term extractor will create it. After term extraction, this property is typically mapped to an Endeca dimension via the property mapper.

Update mode

The UPDATE_MODE pass-through specifies which type of data update is being performed by the pipeline.

The UPDATE_MODE pass-through is optional. The three values for this pass-through are `STATELESS`, `STATEFUL`, and `PARTIAL`. Note that if this pass-through is omitted, the term extractor performs a `STATELESS` update.

STATELESS mode

`STATELESS` is analogous to a baseline update and performs the following actions:

1. Extracts terms from all records.
2. Performs corpus-level and record-level filtering on all records.
3. Emits all records.

4. Does not create state files.

STATEFUL mode

STATEFUL performs the following actions:

1. Extracts terms from new records only.
2. Performs corpus-level and record-level filtering on all records (i.e., both previously processed records and new records).
3. Emits all records (i.e., both previous and new records).
4. Creates term state files.

For information on the STATEFUL mode, see the "Using STATEFUL mode for tuning" topic in Chapter 6.

PARTIAL mode

PARTIAL is analogous to a partial update and performs the following actions:

1. Extracts terms from new records only.
2. Performs record-level filtering on new records only. Corpus-level filtering is not performed at all, so the previous corpus information is left as-is.
3. Emits new records only.
4. Does not create state files (i.e., previous term state files are left as-is).

For details on performing PARTIAL updates with a Term Discovery pipeline, see the "Partial updates for term extraction" topic in Chapter 6.

Notes on update modes

Keep the following notes in mind when using the update modes:

- STATELESS mode is recommended for sites that perform baseline updates exclusively.
- STATEFUL mode is recommended for sites that implement partial updates. That is, the baseline update pipeline will use STATEFUL mode while the partial update pipeline will use PARTIAL mode.
- The STATELESS and STATEFUL modes do not need pre-existing state files in order to run.
- The PARTIAL mode does require the existence of the term state files. Therefore, a STATEFUL update must be performed before a PARTIAL update.

Maximum number of input records

The MAX_INPUT_RECORDS pass-through specifies the maximum number of input records to be processed by Forge.

The MAX_INPUT_RECORDS pass-through is optional. This pass-through is intended for development purposes and should be omitted in a production environment so that all records are processed.

Note that the Java manipulator will respect the Forge `-n` flag. The difference is that MAX_INPUT_RECORDS affects the input of records while `-n` affects the output (emission) of records. That is, with the `-n` flag, all source records are read in, but only `-n` records are emitted; with the MAX_INPUT_RECORDS pass-through, only the specified number of records are read in (and emitted).

Configuration for candidate term identification

A set of pass-throughs is available to configure which terms are candidates for term extraction.

The following configuration parameters are used when performing candidate term identification for terms:

PASS_THROUGH Element	Configuration Value
TEXT_PROP_NAME	Source property to use as the source text for term extraction. Mandatory. No default.
ALL_TERMS_OUTPUT_PROP_NAME	Source property to use as the destination for all terms on a record. Mandatory. No default.
INPUT_TERM_PROP_NAME	Source property to use as the source text for pre-tagged records. Optional. No default.
LANG_PROP_NAME	Source property containing the language ID. Optional. No default.
LANG	Language ID that determines whether the English (en) or French (fr) version of the term extractor will be used. Optional, but its use is recommended. Default is the first language in the product configuration file.

Source input text property

The TEXT_PROP_NAME pass-through specifies which pre-existing property on the input data record will be used as the source for term extraction.

The TEXT_PROP_NAME pass-through is mandatory. Note that the value for this pass-through specifies a source property, not an Endeca property. This property will then be mapped to an Endeca dimension.

If you want to extract terms from multiple properties, a pipeline component must combine the text from the multiple properties into one text property. The name of that text property is then used for the TEXT_PROP_NAME pass-through.

All-terms destination property

The ALL_TERMS_OUTPUT_PROP_NAME pass-through specifies the property which gets all terms on a record that pass corpus-level filtering.

The ALL_TERMS_OUTPUT_PROP_NAME pass-through is mandatory. The property gets all terms on a record that pass corpus-level filtering regardless of whether they pass record-level filtering.

If the property does not already exist in the implementation, the term extractor will create it. When mapped to an Endeca property, record searches can be performed on this all-terms property (assuming it is configured to be searchable).

The all-terms property is used for search purposes. For each record, the term extractor finds all of the terms in the corpus-wide vocabulary that occur in that document (regardless of their relevance for that document) and puts them in the all-terms property. By using this property for searches, stemming of the terms can be performed.

For example, if the terms "search engine" and "search engines" appear in the corpus, they will be normalized to the dominant form (e.g., as "search engine"). But if you want to find all records that

contain either variant, you cannot use Phrase search against the body text, because Phrase search does not locate stemmed variants. Instead, Term Discovery ensures that both the dimension value name and the term stored in the all-terms property is the dominant form.

The terms in the all-terms property are separated by using **sep** as the delimiter. The term extractor makes the separator by doing `(sep)+` until the separator is not a substring of any term in the corpus. Therefore, the separator may be **sep**, **sepsep**, **sepsepsep**, and so on. For example, an article on Aldera, Spain might produce the following all-terms property (named P_AllTerms) on the record (in the example, **sepsep** is the separator for the property):

```
P_AllTerms: district sepsep Spain sepsep south coast sepsep
     coast sepsep town sepsep Aldera sepsep province sepsep
     Romans sepsep decline sepsep station sepsep hills sepsep
     Heracles sepsep temple sepsep colonies sepsep Tiberius
```

Because of the widespread use of this separator, you should add it to the stop word list. (Note that this is the application's stop word list, not the term exclude list.) Before doing so, first determine which form is the separator. Run the corpus at least once to find what the separator is and then set that separator as a stop word. For example, if "sep" is a valid term in the corpus, then it is likely that **sepsep** will the separator. Thus, you would add "sepsep" (but not "sep") to the stop word list. Then, periodically monitor the corpus to make sure the separator has not changed.

Input term property

The INPUT_TERM_PROP_NAME pass-through is used for a corpus that contains pre-tagged records.

Pre-tagged records are source records with pre-existing terms that were generated by non-Endeca software. In this scenario, you do not want to extract new terms from the documents but do want to perform corpus-level and/or record-level filtering on the pre-tagged terms.

This pass-through can be used in conjunction with the TEXT_PROP_NAME pass-through to combine the pre-existing terms with the new extracted terms. For details on working with pre-tagged records, see the "Term filtering with pre-tagged records" topic in the *Advanced Term Discovery Topics* section.

Language specification of input records

Two pass-throughs set the language ID of input records on a global and per-record basis.

You can use the LANG and LANG_PROP_NAME pass-throughs to specify the global language ID and the per-record language ID of the input records. The language ID is case insensitive for both pass-throughs (for example, you can specify either `EN` or `en` for English).

Note that the LANG_PROP_NAME value takes precedence, and if not present, the value of LANG is used as the language of the record.

Both pass-throughs are optional and both can be specified in a Java manipulator.

LANG pass-through

The LANG pass-through specifies the language ID to use on a global basis. Currently, you can specify either `EN` (or `en`) to run the English version of the text extractor or `FR` (or `fr`) to run the French version.

If you do not specify this pass-through, the global language ID defaults to the first language specified in the product configuration file (named ProductConfig.xml). Although this pass-through is optional, it is recommended that you use it to explicitly set the global language ID.

LANG_PROP_NAME pass-through

The LANG_PROP_NAME pass-through specifies the name of the record property that contains the language ID for that record. If you do not specify this pass-through, the language ID for each record will default to the value of the LANG pass-through. For example, if the value for LANG is `EN`, then the term extractor will assume that all the records are in English.

If you do specify the LANG_PROP_NAME pass-through, the term extractor will evaluate each record as follows:

- If the value of the LANG_PROP_NAME property matches the LANG setting, then terms are extracted from the record in that language.
- If the value of the LANG_PROP_NAME property does not match the LANG setting, then terms are not extracted from the record. That is, the record is ignored for purposes of term extraction, but the record is otherwise processed by Forge. For example, if the value of LANG is `FR` and the value of the LANG_PROP_NAME property is `EN` for a given record, the terms extractor will ignore that record.
- If the value of the LANG_PROP_NAME property is null or the record does not contain the LANG_PROP_NAME property, the term extractor will assume that the language ID of the record is the same as the LANG setting and therefore will attempt to extract terms from the record in that language.

If you have documents in multiple languages, the LANG_PROP_NAME pass-through is useful to ensure that only records in the desired language (the LANG setting) are processed by the term extractor.

Configuration for corpus-level filtering

A set of pass-throughs is available to configure how corpus-level filtering is done.

The following configuration parameters are used for performing corpus-level filtering:

PASS_THROUGH Element	Configuration Value
CORPUS_MIN_RECS	Integer specifying the minimum number of records in which a term must appear in order to be considered. Default is `0`; minimum recommended value is `2`.
CORPUS_MAX_RECS	Integer specifying the maximum number of records in which a term must appear in order to be considered. Default is unlimited.
CORPUS_MIN_COVERAGE	Double specifying the minimum fraction of the corpus that must contain the term. Default is Double.NEGATIVE_INFINITY; useful range is `0-1`; recommended value is `0.00005`.
CORPUS_MAX_COVERAGE	Double specifying the maximum fraction of the corpus that must contain the term. Default is Double.POSITIVE_INFINITY; useful range is `0-1`; recommended value is `0.2`.
CORPUS_MIN_INFO_GAIN	Minimum global informativeness score a term must have to be considered. Default is Double.NEGATIVE_INFINITY.
CORPUS_MAX_INFO_GAIN	Maximum global informativeness score a term must have to be considered. Default is Double.POSITIVE_INFINITY.
CORPUS_REGEX_KEEP	String that is a regular expression for terms that should be kept. No default.

PASS_THROUGH Element	Configuration Value
CORPUS_REGEX_SKIP	String that is a regular expression for terms that should be discarded. No default.
CORPUS_DEBUG	If set to `true`, detailed information about corpus-level scoring is written to the logs. Default is `false`.

Minimum and maximum occurrences in records

Two pass-throughs set the minimum and maximum number of records in a term can appear before it is discarded.

The CORPUS_MIN_RECS parameter determines the minimum number of records in which a term can appear before it can be considered. If a term appears in fewer records than the CORPUS_MIN_RECS setting, then the term is discarded.

Setting a value less than 2 is not recommended, as it useless for clustering. In addition, a value of 2 or higher means that singlets are eliminated.

Singlets are terms that appear in only one record. Singlets typically occur very frequently in a corpus, but cannot be used for clustering, which uses cross-record statistics. Therefore, eliminating singlets reduces memory use and computation time. Minimum occurrences should be set to at least 2 to remove singlets; higher values will require a term to appear on more records. Note that large document sets have more term redundancy and can have this parameter set to higher values, such as 3 or higher.

The CORPUS_MAX_RECS parameter sets the maximum number of records in which a term can appear. If it appears in more records than the set value, then the term is discarded.

You can use both pass-throughs to create a window. For example, assume you have set these values:

```
CORPUS_MIN_RECS=5
CORPUS_MAX_RECS=20
```

The result would be that a term would be retained only if it appeared in at least 5 records but no more than 20 records.

Minimum and maximum coverage settings

Two pass-throughs set the minimum and maximum percentage of records that must contain a term.

The coverage pass-throughs correspond to the fraction of the records in the corpus which contain at least one occurrence of the given term:

- CORPUS_MIN_COVERAGE sets the minimum fraction of the corpus that must contain the term.
- CORPUS_MAX_COVERAGE sets the maximum fraction of the corpus that must contain the term.

For example, if you want to keep only those terms that appear in between 5% and 25% of the corpus, you would use these settings:

```
CORPUS_MIN_COVERAGE=0.05
CORPUS_MAX_COVERAGE=0.25
```

The useful range for both pass-throughs is 0-1, in which 1 is 100% of the corpus.

Threshold for the global informativeness of terms

Two pass-throughs set the scoring threshold for the global informativeness of a term.

The CORPUS_MIN_INFO_GAIN parameter sets the minimum scoring threshold when measuring the global informativeness (info_gain) of a term. The useful range for CORPUS_MIN_INFO_GAIN is minus infinity to plus infinity, although most terms tend to fall in the -5.0...+5.0 range. The terms with negative information gain are likelier to contribute more noise than signal. Eliminating these globally uninformative terms saves considerable memory and query compute time. The minimum info_gain setting can be increased to require higher global informativeness—or, conversely, decreased to allow more terms to be retained. Setting CORPUS_MIN_INFO_GAIN to 0 is usually adequate. Setting this parameter to values higher than 1.0-2.0 can dramatically reduce the number of terms.

The CORPUS_MAX_INFO_GAIN parameter sets the maximum scoring threshold for the global informativeness of a term. If the scoring for a term exceeds this threshold, the term is discarded.

Using regular expressions

Two pass-throughs allow you to use regular expressions to retain or discard terms.

The term extractor supports two pass-throughs that are used for matching character sequences against patterns specified by regular expressions:

- CORPUS_REGEX_KEEP specifies a regular expression that a term must match in order to be retained.
- CORPUS_REGEX_SKIP specifies a regular expression that a term must not match in order to be retained.

You can use either or both pass-throughs. If both are used, then a term must pass both tests in order to be retained (that is, if the term satisfies one pass-through but not the other, the term is discarded).

The term extractor implements Sun's `java.util.regex` package to parse and match the pattern of the regular expression. Therefore, the supported regular-expression constructs are the same as those in the documentation page for the `java.util.regex.Pattern` class at this URL:

http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

This means that among the valid constructs you can use are:

- Escaped characters, such **\t** for the tab character.
- Character classes (simple, negation, range, intersection, subtraction). For example, **[^abc]** means match any character except a, b, or c, while **[a-zA-Z]** means match any upper- or lower-case letter.
- Predefined character classes, such as **\d** for a digit or **\s** for a whitespace character.
- POSIX character classes (US-ASCII only), such as **\p{Alpha}** for an alphabetic character, **\p{Alnum}** for an alphanumeric character, and **\p{Punct}** for punctuation.
- Boundary matchers, such as **^** for the beginning of a line, **$** for the end of a line, and **\b** for a word boundary.
- Logical operators, such as **X|Y** for either X or Y.

For a full list of valid constructs, see the `Pattern` class documentation page at the URL listed above.

The following is an example of a useful regular expression that uses the POSIX **\p{Alnum}** construct:

```
^\p{Alnum}[\p{Alnum}\.\-' ]+$
```

When used with the CORPUS_REGEX_KEEP pass-through, this regular expression will retain only terms that have at least two characters, starts with an alphanumeric character, and contains only alphanumeric, period, dash, apostrophe, and space characters. (The apostrophe is to retain terms such as `O'Malley`).

The following is a another example that is fairly restrictive:

```
[^A-Za-z0-9\-\. ]
```

When used with the CORPUS_REGEX_SKIP pass-through, this regular expression will retain only terms that consist of alphanumeric, dash, period, and space characters.

Enabling debugging information for corpus-level filtering

The CORPUS_DEBUG pass-through enables the logging of debugging information.

The CORPUS_DEBUG pass-through enables the term extractor to write detailed information about the corpus-level filtering scores it assigns to terms. Temporarily setting this pass-through to `true` will help you to tune corpus-level filtering. For details, see the "Using corpus-filtering logging statistics" topic in Chapter 6.

Configuration for record-level filtering

Two pass-throughs are available to configure record-level filtering for term extraction.

This table lists the configuration parameters that are used for performing record-level filtering. Note that both pass-throughs are optional.

PASS_THROUGH Element	Configuration Value
RECORD_FRACT_OF_MEDIAN	Double that sets the minimum scoring fraction of the median to use. `1.1` is the recommended value; `0.0` is the default.
RECORD_NTERMS	Sets a limit on the maximum number of terms that are tagged on a record. Default is to have no limit.

Specifying a scoring threshold

The RECORD_FRACT_OF_MEDIAN sets a scoring threshold for record-level filtering.

When the Term Discovery software decides which terms will be tagged on a record, it uses a scoring method in which terms that are more frequent in this document than across the corpus are considered to be more relevant and thus are retained. Filtering occurs on the document-by-document basis; in other words, each term is considered for inclusion or exclusion separately for each document in which it occurs.

The distribution of scores for terms on a single record typically has very few high-scoring terms, followed by a long, gently-sloped plateau of marginally informative terms, with a sudden drop-off of few uninformative terms.

The RECORD_FRACT_OF_MEDIAN value lets you set a scoring threshold for the plateau; only terms that score above this threshold are kept. RECORD_FRACT_OF_MEDIAN should be set to a value that expresses the threshold as a fraction of the median score for terms on the document.

The recommended threshold is `1.1` (i.e., 10% higher than the median), which will keep only the highly-informative terms. Higher values will tend to increase precision (the terms that are kept are more likely to be relevant) but decrease recall (more likely to lose relevant terms). The default value of this threshold is `0.0`, which allows all terms through.

Limiting the number of terms per record

The RECORD_NTERMS pass-through sets a limit on the maximum number of terms that are tagged on a record.

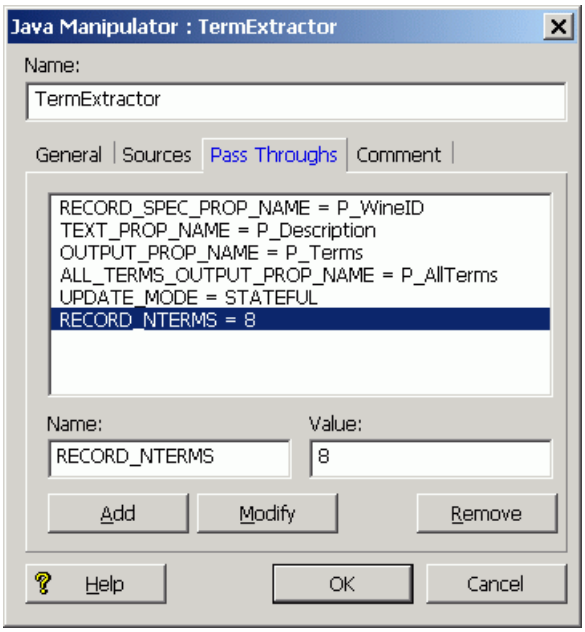
You can use the RECORD_NTERMS pass-through to implement one of two strategies to limit the number of terms that are tagged on records:

- Set an absolute upper limit.
- Establish a cut-off window.

You cannot mix both strategies. In both strategies, the Term Discovery software determines which terms have the highest relevance for that record. Note that this pass-through is recommended mainly for collections that have large documents.

Setting a hard limit

To set an absolute upper limit, use the RECORD_NTERMS pass-through with only one integer value. Use this version of the pass-through when you are certain about the number of terms you want per record and can therefore set a hard limit. In this example, RECORD_NTERMS is set to a value of 8:

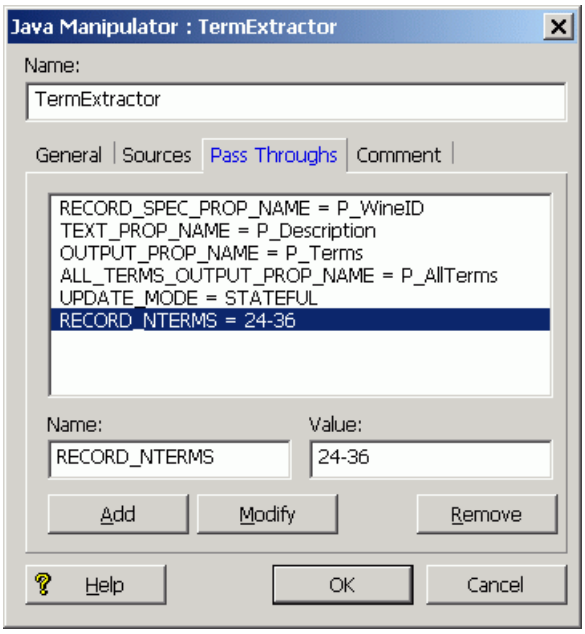


Using this setting, Term Discovery will determine which are the eight most relevant terms for this record and tag the record with them.

Establishing a cut-off window

To establish a cut-off window, use the RECORD_NTERMS pass-through with a range of two integers, which sets the lower and upper limits of a cut-off window. This windowing strategy establishes a window that will be scanned for an optimal cut-off. This cut-off is where term informativeness drops off most precipitously. Use this strategy when you want Term Discovery to be sensitive to actual term informativeness rather than just using a hard limit.

In the following example, the value for RECORD_NTERMS is the range of 24-36 terms:



You can think of the term range as providing a fuzzy neighborhood to be used instead of a hard limit. For example, instead of RECORD_NTERMS having a hard limit of 32, you can set it to a range of 24-36. This range establishes a window where a record can have a minimum of 24 terms and a maximum of 36 terms. The Term Discovery algorithm will determine the optimal cut-off within that window for each record.

For example, assume the above settings and also assume that 40 terms were extracted from Record A and also from Record B:

- For Record A, the optimal cut-off for the terms might be after term 26 (because of a sharp drop-off in relevancy for terms 27-40). Therefore, Record A will have 26 terms tagged onto it.
- For Record B, the optimal cut-off for its set of terms might be after term 30. In this case, Record B will have 30 tagged terms.

When using the range version of this pass-through, keep the following in mind:

- The lowest recommended value for the lower limit is around 10. The reason is that the scores of the top terms scores usually differ noticeably, and the largest score drop-off is likely to be found at the setting for the lower limit. Thus, if the lower is less than 10, you should expect it to behave like the hard-limit version of RECORD_NTERMS, which is misleading.
- The value for the upper limit should not be much larger than the value for the lower limit. If the difference is too much, the number of terms assigned to each particular record will be essentially random (within the cut-off window). The only way to have this number of terms relatively consistent is to use a lower- and upper-limit pair that are not too far from each other.

Best practices for term filtering

This topic presents a best practices list for term extraction.

The best practices lists include the pass-through name and a recommended (best practices) value. If a recommended value is not given, then the default value is also the recommended one.

The two important things to keep in mind are:

- When tuning corpus-level filtering, the number of documents is the main consideration.

- When tuning record-level filtering, the size of the text property is the main consideration.

Two tuning aids that may helpful are described in the "Tuning aids for the filtering parameters" topic in Chapter 6.

Corpus-level filtering best practices

CORPUS_MIN_RECS

Recommendation: Values of less than 2 are not recommended in general, since they allow terms that are seen only once in the entire corpus. If clustering is used, this value MUST be set to at least 2. Note that this parameter works similarly to CORPUS_MIN_COVERAGE: terms that are seen less frequently than in CORPUS_MIN_RECS are discarded, as are terms that are seen in less than CORPUS_MIN_COVERAGE `*(number of documents in the corpus)`.

CORPUS_MAX_RECS

Recommendation: As a general rule of thumb, this pass-through does not have to be used. If your number of records can change (say, via partial updates), Oracle recommends that you not use CORPUS_MAX_RECS, because the statistics will change with the changed number of records. In this case, you may want to use the CORPUS_MAX_COVERAGE pass-through instead.

CORPUS_MIN_COVERAGE

Recommendation: The useful range is 0-1. A value of 0.00005 is a good compromise, because the term extractor will retain a term if it has been seen in at least one document out of 20,000.

This value will change with the nature of the data set. For example, a site with a data set with a lot of topical diversity (such as news) can reduce this value and allow terms with lower coverage (however, one out of any hundred thousand is probably the smallest reasonable value). If memory use is an issue, you should increase this value.

CORPUS_MAX_COVERAGE

Recommendation: The useful range is 0-1. A value of 0.2 (which is 20% of the documents) is a good compromise. If a term is seen in more than one out of five documents (i.e., 20%), it is probably too broad to be useful. If terms that are tagged onto documents seem too generic, this number should be turned down. As with CORPUS_MIN_COVERAGE, turning this number down, even slightly, should free memory.

CORPUS_REGEX_KEEP

Recommendation: A useful regular expression for terms to keep is:

```
^\p{Alnum}[\p{Alnum}\.\-' ]+$
```

This retains terms that have at least two characters, start with an alphanumeric character, and includes only alphanumerics, spaces, periods, dashes, and single quotes.

Note: Each term must both match CORPUS_REGEXP_KEEP and not match CORPUS_REGEXP_SKIP to be retained.

CORPUS_REGEX_SKIP

Recommendation: Use this pass-through only if you are certain of the format of the terms you want to discard.

CORPUS_MIN_INFO_GAIN and CORPUS_MAX_INFO_GAIN

Recommendation: Begin by setting CORPUS_MIN_INFO_GAIN to 0. Do not set CORPUS_MAX_INFO_GAIN initially. Tune the other term extraction pass-throughs. Then, run a data set (or a subset) with CORPUS_DEBUG set to `true`, which will print the list of terms that passed all

the selection criteria. You can use this information to adjust the selection criteria, which may include adjusting the CORPUS_MIN_INFO_GAIN and using the CORPUS_MAX_INFO_GAIN pass-throughs.

If fewer generic terms are desired, increase the value of CORPUS_MIN_INFO_GAIN in small increments (0.5 or 1.0). If more generic terms are desired, decrease this value. CORPUS_DEBUG can be used to select a particular value of CORPUS_MIN_INFO_GAIN.

CORPUS_DEBUG

Recommendation: Set this pass-through to `true` only when you are tuning the filtering parameters; otherwise, do not use it. For details on the logging entries it produces, see the "Using corpus-filtering logging statistics" topic in Chapter 6.

Record-level filtering best practices

RECORD_NTERMS and RECORD_FRACT_OF_MEDIAN

Recommendation: The use of these pass-throughs depends on the length of text in the text property that contains candidate terms. The two scenarios considered here are properties with either short text or long text.

For short text (such the P_Description property in the wine data set shipped with the sample reference implementation), the recommendation is to not use these pass-throughs, which will keep all the terms.

For long text (such as news sites or sites with long articles), use the range version of RECORD_NTERMS to set however many terms per record you want, say a range of 16-24 or, if more is wanted, a range of 24-30. (Keep in mind that the lower limit should be greater than 10 and the upper limit should not be much large than the lower limit.) Set RECORD_FRACT_OF_MEDIAN to 1.1 for relatively small documents, 1.2 for larger documents, and 1.5 for very large documents.

Minimal term extraction configuration

The minimal term extraction configuration consists of four pass-throughs.

The four pass-throughs for a minimal term extraction configuration are the following:

- RECORD_SPEC_PROP_NAME
- TEXT_PROP_NAME
- OUTPUT_PROP_NAME
- ALL_TERMS_OUTPUT_PROP_NAME

This configuration will run as follows:

- A baseline update (STATELESS mode) will be performed.
- No corpus- or record-level filtering will be performed, but the exclude list (if it exists) will be processed. As a result, all terms in the corpus will be extracted and all of them (with the exception of the terms on the exclude list) will be tagged onto records.
- The expected language of the documents will be the first language specified in the product configuration file.

This configuration is the most permissive for term extraction. Although most sites will prefer to perform some level of corpus and record filtering, this minimal configuration may be useful with small data sets that have a closely-related set of noun phrases in their documents.

Format of the source data

Terms can be extracted from either structured and unstructured source data.

In general, data and content acquisition will typically be used to retrieve the records used by the term extractor. In particular, crawling (via the Endeca Web Crawler or the Endeca CAS Server) is a viable source of content for term extraction.

The term extractor utilizes Natural Language Parsing (NLP). In order to maximize the accuracy of NLP output, the input must be as clean as possible and as similar to natural language as the original data allows. The following list provides some recommendations about how to pre-process unstructured text that is fed to the term extractor:

- Remove anything from the property that is sent to the term extractor that is not the main contents of the document. For example, when dealing with news articles, it is a good idea to remove bylines, copyright disclaimers, and so forth. When dealing with Web pages, the task is noticeably harder, because the navigational elements, page headers and footers, guestbook links, ads, etc., all have to be removed. In such extreme cases, one suggestion is to retain long sequences of sentences with correct sentence-terminating punctuation that does not have many major HTML tags (H1, P, HR, DIV, SPAN, etc.) embedded: meaningful text is likely to satisfy this requirement, and items such as menus, ads, and page elements are likely to fail it.
- Remove anything that is not natural language text. This includes HTML tags, other markup, and long sequences of non-alpha characters (e.g., long sequences of dashes used as delimiters). Links to images, URLs (that might be used in plain text, outside of HTML tags), and anything that is not in grammatically correct language should be stripped. The same caveat applies to sequence of characters that are too long to be meaningful terms. The term extractor will report and skip those overlong noun phrases. However, it is useful to detect these upstream of the term extractor because their presence might indicate sections of the documents that should be removed.
- Punctuation should be correctly spaced, especially when stripping HTML or adding sentence-terminating punctuation. A sentence terminator is correctly interpreted only if it is followed by a space. For example:

  ```
  Look at this.<IMG a.gif>Or this.
  ```

  should not be converted to:

  ```
  Look at this.Or this.
  ```

  but instead to:

  ```
  Look at this. Or this.
  ```

- Convert non-sentence text into sentences. If, for example, a useful section of the document is written as a list or a table (that is, separated with <LI> or <TD> tags), it is recommended to terminate such entries with periods (or semicolons, depending on context), if they are not so terminated to begin with.
- Merge text fields, if needed. For example, if the document title is a separate property, it is useful to append it to the main text property (terminating it with punctuation, if possible).
- Use the correct case for capitalized and non-capitalized text. NLP relies on capitalization to make correct part-of-speech judgments. If capitalization is not available, it makes a best guess and this guess is better when dealing with lower-case text than with all upper-case text. If the document text is in all upper case, it is advisable to convert it to all lower-case (or, possibly, all lower case with initial capitalization for the first word in every sentence). This will improve NLP quality.
- Use correct spelling in Web pages, especially blogs. For documents written in informal language, it is recommended that simple pattern replacement be done on most frequent terms (for example, "u" should be replaced with "you").

As mentioned in the second item above, the term extractor will discard overly-long noun phrases and issue warning messages, such as this example from the Forge log:

```
While processing Record '42710': Overlong noun phrase ending
in 'sturdy white featuring simple pear flavor'
```

The example shows a problem with text in Record 42710.

Whenever possible, the text should be cleansed in the source data. However, you can add record manipulators to the pipeline to perform pre-processing cleanup.

You can also use the CORPUS_REGEX_KEEP and CORPUS_REGEX_SKIP pass-throughs in the Java manipulator to control which extracted terms are kept or discarded. For details on how to construct regular expressions with these pass-throughs, see the "Using regular expressions" topic of this chapter.

Chapter 3

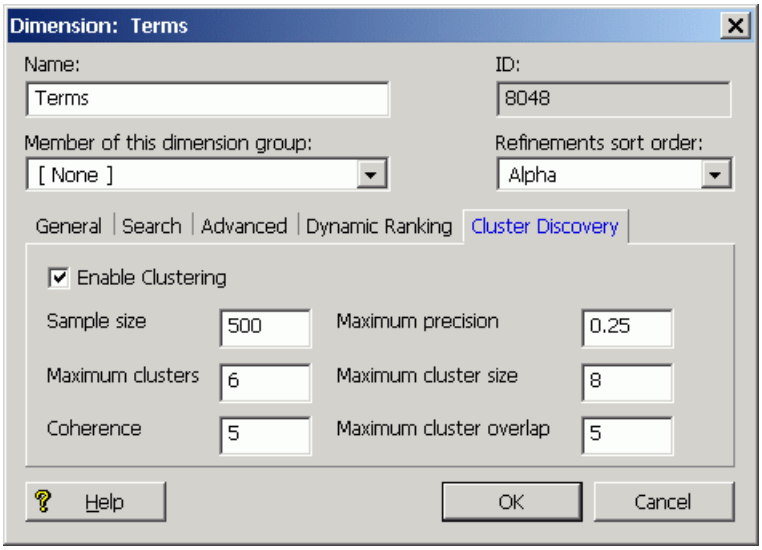
Configuration Guidelines for Clustering

This section describes some guidelines for configuring term extraction.

Configuration UI for clusters

You use the Dimension editor in Developer Studio to configure clustering for a Term Discovery dimension.

In the Dimension editor, the Cluster Discovery tab is where you set the configuration parameters:



The next two topics provide descriptions of the parameters and guidelines for tuning them.

Clustering parameter descriptions

This topic describes the meanings of the parameters on the Cluster Discovery tab.

The list below describes the clustering parameters on the Cluster Discovery tab, including their range of values (with defaults) and recommended values. The next topic in this section explains the reasoning behind the recommended values and provides an order in which these parameters should be adjusted.

Sample size

Description: Clustering is done by examining term distribution across a sample of the result set. This parameters governs how many records are sampled from the navigation state. Clustering processing time and memory consumption are both roughly linear with this number; thus, lowering the value results in smaller memory consumption and faster turnaround. However, statistical errors are likely to occur when the sample size is small. Setting this value higher will overcome statistical errors for data sets where fewer terms are tagged onto each record.

Range: Integer, 50-2000 (default: 500)

Recommended value: 500

Maximum clusters

Description: This parameter limits the number of clusters that will be generated by the MDEX Engine.

Range: Integer, 2-10 (default: 10)

Recommended value: 6

Coherence

Description: This parameter governs the decision of whether a set of terms is coherent enough to form a cluster (that is, each cluster should have only closely related records). Low values are permissive (i.e., not demanding much coherence) and will result in fewer larger clusters. High values are strict and will result in more smaller clusters. The average value is recommended.

Range: Integer, 0-10 (default: 5)

Recommended value: 5

Maximum precision

Description: Terms that are extracted from sampled records are filtered by their precision **p** (where **p** = number of sampled records that this term is tagged onto divided by the number of all sampled records). Terms that have too high a value of **p** are likely to be the search term (or be synonymous with it) or be too general to make for a good clustering term. If you use the recommended tuning values of the term extractor, each term is tagged to only roughly 1/3 of the records that contain this term in the text, which means that the search term, if present, will have **p** of roughly 0.33 (more or less stringent tuning of the term extractor will change this value). There usually is a gap in the values of **p** between the search term and the more useful terms, which start at approximately **p** = 0.25 and less.

Range: Float, 0.0 - 1.0 (default: 1.0)

Recommended value: 0.25

Maximum cluster size

Description: This parameter sets the maximum number of terms in a cluster. Each cluster will have at least 2 terms. Because of the match-partial cluster selection mechanism, the more terms there are in the cluster, the (potentially) higher its coverage will be. On the other hand, the clusters that are too large take up too much space to display and take too long for users to read.

Range: Integer, 2 - 10 (default: 10)

Recommended value: 8

Maximum cluster overlap

Description: If two clusters overlap (that is, if the record sets that each cluster maps to overlap), then the smaller one (as measured by the estimated size of the record set it maps to) can be removed, depending on how big this overlap is. This parameter dictates the overlap above which the smaller cluster is removed.

Clusters which overlap by more than this value will be removed. Thus, the default setting of 10 means that clusters that overlap by more than 10 out of 10 records will be removed. Since this is impossible, this means that setting of 10 will disable cluster overlap filtering, which is most extreme level of coarseness for this filter. Tuning this parameter down will make the cluster overlap more and more fine-grained. Thus, a value of 9 will remove only the clusters that greatly overlap; setting it to the recommended value of 5 will remove only clusters overlapping half-way or so (remember that the overlap is merely estimated). Setting this parameter to lower values (less than 5) will make overlap filtering quite sensitive and will remove clusters which overlap even by a small amount. Note that clusters that do not overlap at all will never be filtered.

Range: Integer, 0-10 (default: 10)

Recommended value: 5

Tuning strategy for clusters

This topic provides guidelines for tuning the clustering parameters.

The guidelines for the clusters tuning strategy include initial values (to be used for the first trial clustering run) and recommended values. The tuning process will involve changing the parameters from their initial values toward their recommended values, with certain variation dependent on the properties of the particular data set and the application needs.

In general, the tuning strategy involves starting with the parameters at a permissive setting and then gradually decreasing the value. You tune the parameters by observing their impact simultaneously on the results for several different queries (no query or node 0; broad queries; narrow queries; single-term query; multi-term query). In other words, you should avoid tuning the parameters based on a specific query.

The following procedure is intended as a tool for gradual tuning, as it allows you to observe the effect of changing the parameters on several different queries at once. Use the suggested order, as it maps to the order in which these parameters impact the clustering algorithm, from upstream to downstream.

1: Number of records sampled from the navigation state

Recommended value: 500

Initial value: 500

Strategy: Start with the parameter set to 500, and increase it if you see that the terms at the bottom of your related terms list (terms 100-120 or so) are seen in fewer than 3 records.

Note that the recommended value of 500 is for data sets with 20 or more terms tagged onto each record. Use a higher value for data sets with fewer terms per record. If an average record has only 2 to 3 terms per record, set this value to 2000. A good rule of thumb for this value is: when the 120 most frequent terms are sampled for clustering, the 120th term should be present in at least 3 records. If it is present in fewer, this setting should be increased.

2: Maximum refinement precision

Recommended value: 0.25

Initial value: 1.0

Strategy: Start with this value set to 1.0 (no precision filtering). Try several different queries and pick a level of top useful precision that separates useful terms from the frequent but uninformative ones. Note that, typically, only the values between 0.05 and 0.5 will be useful.

3: Maximum number of terms per cluster

Recommended value: 6 - 8

Initial value: 10

Strategy: Start with a value of 10 to see all the terms that are getting into clusters. Reduce the value until the clusters are small enough to fit into whatever real estate your UI provides. Using a value of 2 is not recommended. Note that the cluster coverage (and recall) are reduced when the number of terms is reduced.

4: Cluster Coherence

Recommended value: 5

Initial value: 5

Strategy: Start with the default value of 5. If you see undesirable cluster splintering (several clusters that seem to map to the same semantic areas), this value should be decreased; on the other hand, if the cluster set is missing some semantic areas, this value should be increased. Note that it is acceptable to have several overlapping clusters remaining after tuning this value, because they will be removed in the next step.

5: Maximum cluster overlap

Recommended value: 5

Initial value: 10

Strategy: Start with a value of 10, then decrease this parameter until the desired number of overlapping clusters remains (i.e., in some cases, depending on customer needs, some cluster overlap can be retained, particularly if the smaller cluster is an especially coherent one).

6: Maximum number of clusters

Recommended value: 6

Initial value: 10

Strategy: Start with a value of 10 to see all the available clusters after all the other settings had been applied. Reduce this number if you still see more clusters than permitted by the available UI space.

Chapter 4

Creating a Term Discovery Application

This section describes how to create and configure a Relationship Discovery application.

Term Discovery application workflow

This section gives an overview of the major tasks in building a Term Discovery application.

The general steps in building the application are:

1. Create and configure the appropriate Endeca properties and dimensions for term extraction and clustering, such as the record specifier property, the all-terms property, and the Term Discovery dimension.
2. Create the record adapters for the term extraction pipeline.
3. Create the Java manipulator that performs the term extraction.
4. Add other pipeline components.
5. Run the pipeline.

Note that this section does not describe how to modify the front-end of the application. That information is in Chapter 5, "Building the Front End of the Term Discovery Application."

Configuring the required dimension and properties

The Endeca dimension and properties needed by the Relationship Discovery components must be created and configured.

The tasks are:

- Designate an Endeca property to be the record specifier property for the application.
- Configure the Term Discovery dimension.
- Configure the all-terms property.
- Create a partial-match search interface.

These tasks are described in the following topics.

Designating the record specifier property

One of the Endeca properties must be designated as the record specifier property.

This task assumes that an Endeca property already exists that can qualify to be the record specifier property. The property must meet the following requirements:

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

Keep in mind that only one property in the project may be designated as the record specifier property.

You must designate a record specifier property because the RECORD_SPEC_PROP_NAME pass-through (for the term extraction Java manipulator) needs to use this property for record identification purposes.

To designate a record specifier property:

1. In the Project tab of Developer Studio, double-click **Properties**.
2. From the Properties view, select a property and click **Edit**. The Property editor is displayed.
3. In the General tab, check **Use for Record Spec**.
4. Click **OK**. The Properties view is redisplayed.
5. From the File menu, choose **Save**.

Configuring the Term Discovery dimension

An Endeca dimension must be configured as the Term Discovery dimension.

The extracted terms are mapped to an Endeca dimension, called a Term Discovery dimension. The Term Discovery dimension must have these two attributes:

- It must be a flat dimension (that is, a dimension that does not contain hierarchies).
- It must not be a hidden dimension.

To configure the Term Discovery dimension:

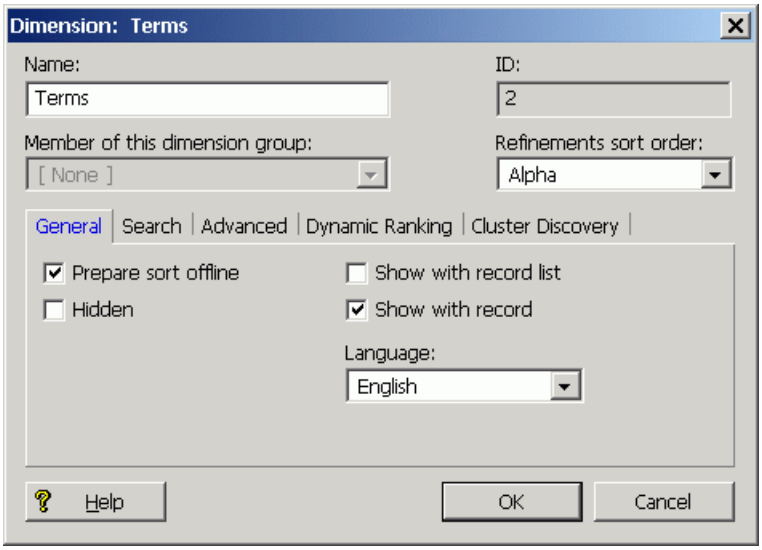
1. In the Project tab of Developer Studio, double-click **Dimensions**.
2. Create a dimension to be the Term Discovery dimension.
3. In the Dimensions editor, configure the Term Discovery dimension, using the following recommended values (DR refers to the Dynamic Ranking tab):

Tab/Field	Recommended Value
none/Name	*any valid dimension name*
none/ID	*system generated*
none/Member of this dimension group	**None**
none/Refinements sort order	**Alpha**
General/Prepare sort offline	Checked
General/Hidden	Unchecked
General/Show with record list	*as required by the application*
General/Language	Either **Default**, **English**, or **French**
Search/Search hierarchy for dimension search	Checked
Search/Enable record search	Checked
Search/Search hierarchy for record search	Unchecked

Tab/Field	Recommended Value
Search/Enable wildcard search	*as required by the application*
Advanced/Primary	Unchecked
Advanced/Enable for rollup	Unchecked
Advanced/Compute refinement statistics	*as required by the application*
Advanced/Collapsible dimension threshold	*leave text field empty*
Advanced/Multiselect	Either **None** or **And** (do not select **Or**)
DR/Enable dynamic ranking	Checked
DR/Maximum dimension values to return	**10**
DR/Sort dimension values	**Alphabetically**
DR/Generate "More..." dimension value	Unchecked
Cluster Discovery/all fields	See Chapter 3 ("Configuration Guidelines for Clustering").

4. When finished, click **OK**.
5. From the File menu, choose **Save**.

The resulting Dimension editor should look similar to this example:



Configuring the all-terms property

The purpose of the all-terms property is for users to perform record searches against the extracted terms, in order to increase recall.

The ALL_TERMS_OUTPUT_PROP_NAME pass-through is used to create the property value. The property should then be mapped to an Endeca property, with a name of your choosing.

To configure the all-terms property:

1. In the Project tab of Developer Studio, double-click **Properties**.

2. Create a property to be the all-terms property.
3. Configure the property, using the following recommended configuration values:

Option	Description
none/Name	*any valid property name*
none/Type	**Alpha**
General/Prepare sort offline	Checked
General/Rollup	Unchecked
General/Enable for record filters	*as required by the application*
General/Use for record spec	Unchecked
General/Show with record list	*as required by the application*
General/Show with record	*as required by the application*
General/Language	Either **Default**, **English**, or **French**
Search/Enable record search	Checked
Search/Enable positional indexing	Checked
Search/Enable wildcard search	*as required by the application*

4. When finished, click **OK**.
5. From the File menu, choose **Save**.

Keep in mind that the all-terms property uses a form of the string **sep** as a separator between the terms. For example, the string **sepsep** may be the separator. It is recommended that you add this separator to the stop word list. (Note that this is the application's stop word list, not the term exclude list.)

Creating a partial-match search interface

You can create a search interface for the all-terms property.

Although not required, it is a good idea that you create a search interface for the all-terms property. The MatchPartial search mode should be configured for the interface, thus allowing matches on subsets of queries.

To create a partial-match search interface:

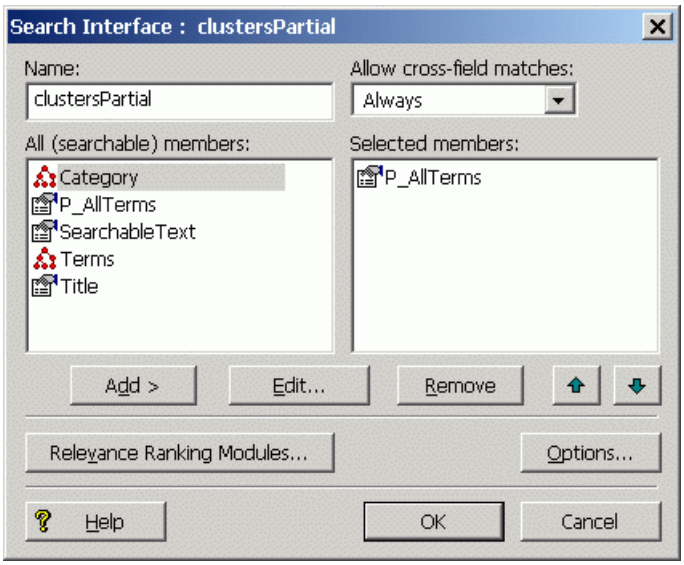
1. In the Project tab of Developer Studio, double-click **Search Interfaces**.
2. To begin creating the search interface, click **New**.
3. In the main dialog of the New Search Interface editor, enter a name for the interface.
4. In the Allow Cross-field drop-down, select **Always**.
5. From the All (searchable) members panel, select the all-terms property and add it to the Selected members box.
6. Click the Relevance Ranking Modules button and add the **NTerms** (Number of Terms) module.
7. Click the Options button to open the Search Interface Options editor. In the editor, configure the following recommended values:

Field	Recommended Value
Customize partial match settings	Checked

Field	Recommended Value
Match at least ... words	2
Omit at most ... words	0

8. Save the configured interface.

The following is an example of a partial-match search interface named **clustersPartial** that was created for the all-terms property named **P_AllTerms**.



Creating the Term Discovery pipeline

This section describes how to create and configure a Term Discovery pipeline using Developer Studio.

The pipeline is used for baseline updates. For instructions on creating a pipeline for partial updates, see the "Partial Updates for Term Extraction" topic in Chapter 6.

The goal of this section is to describe the pipeline components that are specific to Term Discovery, in particular, the Java manipulator. Therefore, components that are common to all pipelines (dimension server, property mapper, indexer adapter, and so on) are omitted for simplicity.

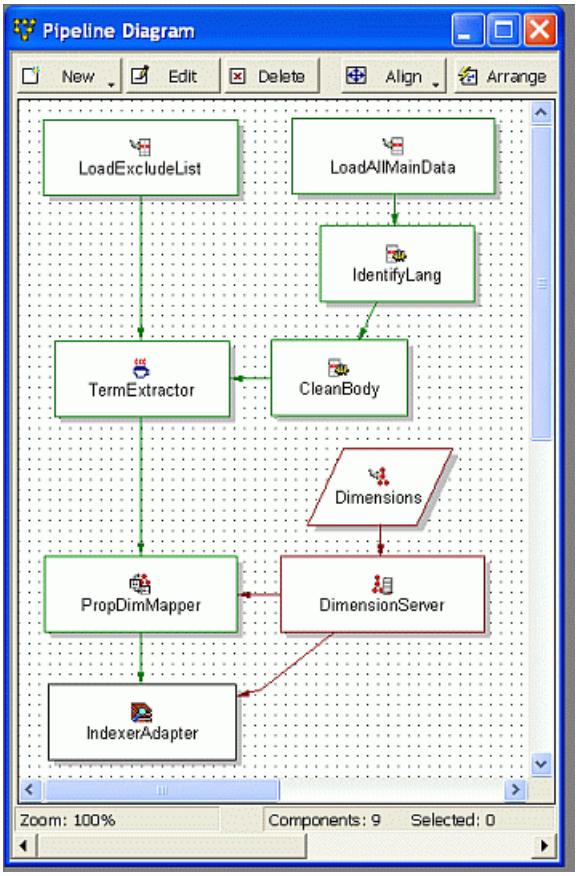
The pipeline for your specific implementation such as a record manipulator to pre-process records, and perhaps another one to post-process the records. For example, if you are crawling a Web site, you will probably include a record manipulator to strip the records of HTML code before the terms are extracted from the records.

The high-level overview of the procedure to construct a Term Discovery pipeline using Developer Studio is:

1. Create a record adapter to read the source records.
2. Create a record adapter to read in the exclude list.
3. Optionally, create other components to pre-process the incoming records.
4. Create a Java manipulator to perform the term extraction process.
5. Create a property mapper to map source properties to Endeca properties and dimensions.

6. Create an indexer adapter. Because there is nothing unique about an indexer adapter for a Term Discovery pipeline, the process of creating this component is not described in this guide.

Here is how the sample Term Discovery pipeline looks in the Developer Studio Pipeline Diagram.



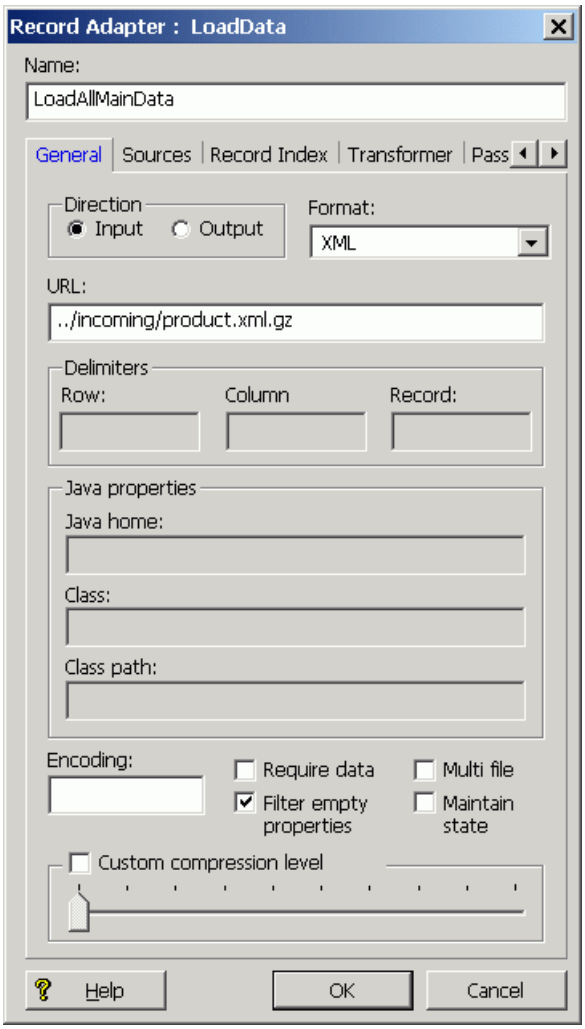
Creating the record adapter for source records

The pipeline needs at least one record adapter to read in the source records.

The actual configuration of the record adapter depends on the data format of the source data. However, the record adapter does not need any special configuration for a Term Discovery pipeline, because the term extraction process begins after the source records are read in.

The General tab is configured as shown in the illustration below. The other tabs have the default settings. The Developer Studio on-line help includes detailed information on how to create and configure the record adapter.

The record adapter for the sample pipeline uses an XML format and looks like this:



Creating the record adapter for the exclude list

If you are using an exclude list, the pipeline needs a record adapter to read in the list.

The exclude list contains a set of terms that will be removed from the final list of extracted terms. Excludes are compared against the canonical and all raw forms of a term; if it matches any, the term will be excluded. This is equivalent to canonicalizing the exclude term.

The excluded terms can be stored in any source file that can be read by a record adapter. For example, the terms can reside in a database and be read with a JDBC record adapter. The sample reference uses a text file that is read in with a Delimited record adapter.

The exclude list should be stored in the Forge input directory (that is, the directory where the source record data set is stored). If you are using the Endeca Application Controller, the `Incoming Directory` (as provisioned in Endeca Workbench for the Forge component) is the mandatory location for the exclude list.

The format rules for the excluded terms list are as follows:

- One (and only one) header must be used.
- The header name can be any name. The sample reference uses EXCLUDE as the header name.

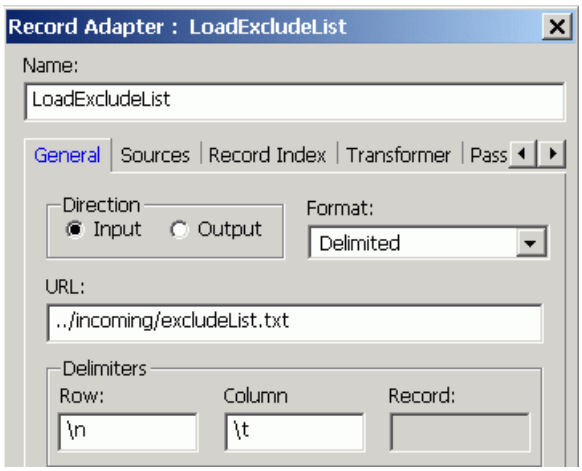
- If you are using a Delimited record adapter, each excluded term must be delineated by the column and row delineators that are configured in the record adapter.
- The exclude list is case insensitive.
- Apostrophes can be included as necessary. For example, enter `i'm` if you want that term to be excluded.

The list is processed after all terms have been extracted from the records.

In this brief example, the names of the weekdays will be excluded from the list of extracted terms:

```
EXCLUDE
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

The excluded terms list is read in by its own record adapter. The record adapter in the sample pipeline uses a Delimited format, and is identical to the source data record adapter, except for the name and URL path, as shown in this partial illustration of the adapter:



Adding pre-processing components

You can add pipeline components that pre-process incoming records.

Although it is not mandatory, you may need to add one or more pipeline components that pre-process the incoming records before they are passed on to the Java manipulator for term extraction. The sample pipeline uses a record manipulator, named CleanBody, to remove HTML coding from incoming HTML documents. This data cleansing is necessary so that the term extractor does not extract HTML tags.

The CleanBody record manipulator is used as the record source for the Java manipulator described in the next section. The record manipulator produces a property named **body** which is used as the source property for the TEXT_PROP_NAME pass-through in the Java manipulator.

The CleanBody record manipulator is described in Appendix A.

Configuring the Java manipulator

A Java manipulator must be configured for the term extraction class.

A Java manipulator component uses one or more Java modules to manipulate source records as part of Forge's data processing. You can use multiple Java manipulators in the pipeline.

The Java manipulator in a Term Discovery pipeline typically performs these major term extraction tasks:

- Extracts terms (noun phrases) from source records.
- Filters out unwanted terms (based on the exclude list).
- Calculates per-record scores for the terms.
- Performs corpus-level filtering, which determines how informative a term is in respect to the other records in the corpus.

The configuration attributes of a Java manipulator are described below.

Creating the Java manipulator

You use Developer Studio to create and configure a Java manipulator.

To create a Java manipulator:

1. From the Pipeline Diagram in Developer Studio, click **New**.
2. Select **Java Manipulator**. The New Java Manipulator editor is displayed.
3. In the Name field, enter a name. The name must be unique among the pipeline components.
4. Fill in the appropriate fields on the General, Sources, and Pass Throughs tabs. See the following sections for details on these tabs.
5. Optionally, you can use the Comment tab to enter description or other comment about this component.
6. Click **Ok**.

Configuring the General tab

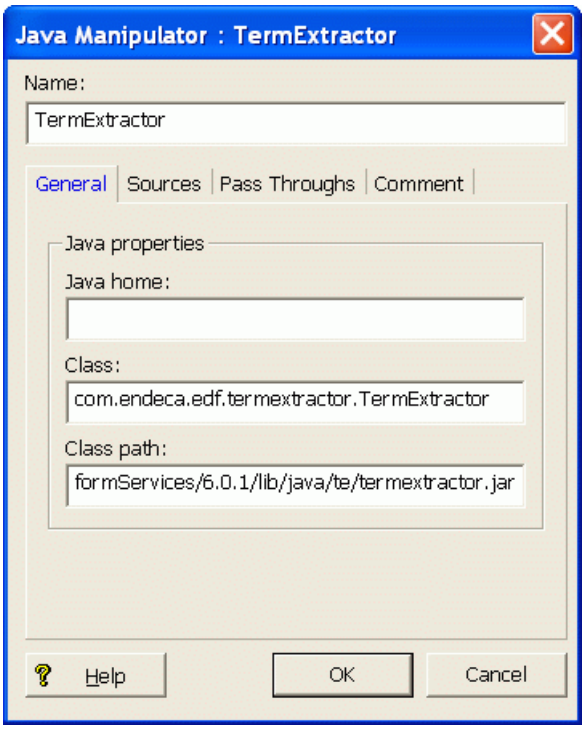
Use the General tab to configure these Java property attributes:

- **Java home**. Optional. Specifies the location of the Java runtime engine (JRE). If this attribute is not specified, Forge tries to obtain the location by using the following sequence:
  1. The argument to the Forge `--javaHome` flag.
  2. The `ENDECA_ROOT/j2sdk` directory, which is installed as part of the Endeca Platform Services package.
  3. The JAVA_HOME environment variable.
- **Class**. Mandatory. Specifies the name of the Java class that will be used by this Java manipulator. Use this class for term extraction:

  ```
  com.endeca.edf.termextractor.TermExtractor
  ```
- **Class path**. Mandatory. Specifies the absolute or relative path to the JAR file that contains the class specified by the Class attribute. The JAR file must contain the class and all other classes it depends on. The following example points to the location of the `termextractor.jar` (which contains the TermExtractor class):

  ```
  /endeca/PlatformServices/6.1.3/lib/java/te/termextractor.jar
  ```

The following is an example of the General tab:



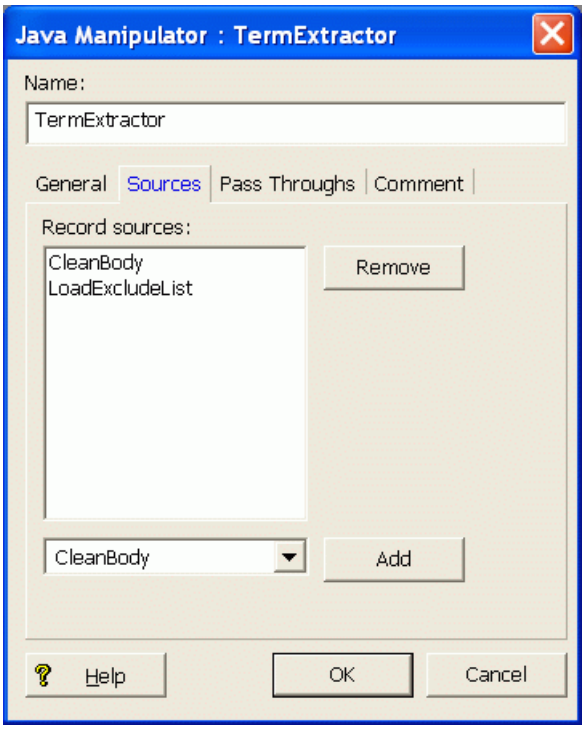
Configuring the Sources tab

Use the Sources tab to specify which other component in the pipeline is providing records to this Java manipulator. You can specify multiple record sources.

A Java manipulator used for term extraction typically uses two record sources: one for the source records and the other for the exclude list. If two record source inputs are configured, the first input is used for source records and the second is for the exclude list.

Note: Make sure that the record sources are configured in the proper order, with the source record source being first. If they are reversed, the TermExtractor class in the Java manipulator will throw an exception and the Forge process will fail. Developer Studio arranges the sources in alphabetical order; therefore, you may have to rename them so that they are displayed in the correct order.

In the following example, the CleanBody record manipulator is considered to be the record data source while the LoadExcludeList source is the exclude list.



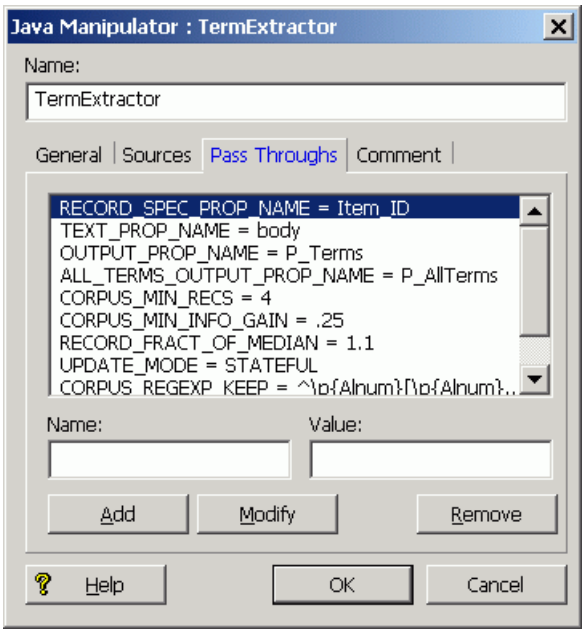
Configuring the Pass Throughs tab

The Pass Throughs tab is used to send configuration-specific information to the Java classes being executed by the Java manipulator. Descriptions of the pass-through name/value pairs for the Java classes are provided in the "Configuration Guidelines for Term Extraction" section.

To add a pass-through name/value pair:

1. In the Java Manipulator editor, click the **Pass Throughs** tab
2. In the Name field, enter the name of the pass-through.
3. In the Value field, enter the value for the named pass-through.
4. Click **Add**.
5. Repeat steps 2-4 as necessary.
6. Click **Ok**.

The following is an example of a populated Pass Throughs tab:



Configuring other components

The Term Discovery pipeline will have other components that are common to all pipelines.

Among these other components are a property mapper. When configuring the property mapper, you must map the property specified in the OUTPUT_PROP_NAME pass-through to the Term Discovery dimension and the ALL_TERMS_OUTPUT_PROP_NAME property to an Endeca property, such as P_AllTerms.

The configurations of other components, such as an indexer adapter, are not described in this guide because there are no requirements that are specific to a Term Discovery pipeline. For more information on pipeline components, see the *Platform Services Forge Guide*.

Running the Term Discovery pipeline

You can run the pipeline with either the Endeca Application Controller (EAC) or control scripts.

Baseline updates for Term Discovery applications are supported under the Endeca Application Controller. No special configuration is needed for Term Discovery applications. See the *Oracle Endeca Workbench Administrator's Guide* for details on provisioning your application to Endeca Workbench and uploading the instance configuration from Developer Studio to Endeca Workbench (this makes the application known to the EAC Central Server.)

For information on using control scripts with term extraction pipelines, see the *Endeca Control System Guide*.

Chapter 5

Building the Front End of the Term Discovery Application

This section provides information on how to change your front-end application to display terms and clusters.

Files to be changed

You can modify existing JSP files and add a new file.

The examples are based on the following JSP files that are shipped with the `endeca_jspref` reference implementation:

- `constants.jsp`
- `controller.jsp`
- `nav_controls.jsp`

In addition, a new `nav_clusters.jsp` file is provided in Appendix A as an example of rendering clusters.

Adding global constants

The `constants.jsp` file sets global variables that can be used in any other page in the application.

It is a good practice to add global variables for handling the displaying of the terms dimension and the clusters. Doing so will allow you to easily change display characteristics in one central file.

The following Java code is an example of a section that can be added to the constants.jsp file:

```
// Dimension name of the Term Discovery dimension.
// Make it null if you do NOT want term processing.
private static final String relTermsDimName = "Terms";

//Display name of the Term Discovery dimension.
private static final String relTermsDisplayString = "Term Discovery";

//The rootId of the Term Discovery dimension.
//Make it -1 if you do NOT want terms processing.
private static final long relTermsRootId = 2;
```

```
// Handling for the all-terms property.
// Ignored if String is null.
private static final String P_AllTerms = "P_AllTerms";

// if true, a More... link shows after the term's short list.
// Should be false in production.
private static final boolean showTermsMore = false;

// If true, TD dimension and property show in record display
// Should be false in production.
private static final boolean showTermsInRecord = false;
```

The code sets up the following variables. These constants will be used in most of the JSP pages listed above.

Global Variable	Purpose
relTermsDimName	Sets the name of the Endeca dimension for Term Discovery.
relTermsDisplayString	Sets the name that will be displayed in application pages for the Term Discovery dimension.
relTermsRootId	Sets the root ID of the Term Discovery dimension.
P_AllTerms	Sets the name of the Endeca property that contains all the terms.
showRelTermsMore	Sets whether a More... link is shown for terms.

Setting refinements in the controller file

The `controller.jsp` module is the entry point into the Endeca application.

The controller file receives the browser request from the application server, formulates the query, and sends the query to the MDEX Engine.

The following code should be added to the `controller.jsp` file:

```
if (relTermsRootId >= 0 && usq.containsNavQuery()
   && request.getParameter("Ne") == null) {
      DimValIdList dvl = new DimValIdList();
      dvl.addDimValueId(relTermsRootId);
      usq.setNavExposedRefinements(dvl);
}
```

The code can be added after the `ENEQuery` object named `usq` is created.

The code first tests that three conditions are true:

- The `relTermsRootId` global variable is set to a value greater than 0.
- The `ENEQuery.containsNavQuery()` method determines that the current query is a navigation query.
- The request does not include the `Ne` URL parameter, which determines which dimension navigation refinements are exposed.

If all three conditions are true, then the code performs three actions:

1. Creates an empty `DimValIdList` object.
2. Uses the `DimValIdList.addDimValueId()` method to add the ID of the Term Discovery dimension to the list.

3. Uses the `ENEQuery.setNavExposedRefinements()` method to set the refinements that are exposed/returned with the navigation query.

In step 3, the dimension ID of the Term Discovery dimension is given as the argument, which means that this dimension will be the parent of the refinements that should be returned with the query. The returned refinements will be the terms.

Displaying refinements

The terms in a Term Discovery dimension are returned as dimension value refinements.

These refinements are displayed according to the coding in the `nav_controls.jsp` file. Because a number of changes are required to handle the Term Discovery refinements, the modified file is included in Appendix A.

Displaying clusters

The information for each cluster is returned from the MDEX Engine in a `Supplement` object that accompanies the result of a navigation query.

The `Supplement` objects are encapsulated in a `SupplementList` object.

Cluster properties

Each cluster (`Supplement` object) contains a `PropertyMap` object.

The `PropertyMap` object in turn contains the following cluster-related properties (as key/value pairs):

Key Name	Value
Dgraph.SeeAlsoCluster	The name of the Term Discovery dimension from which this cluster was generated.
ClusterRank	The zero-based rank of this cluster (0 for the first cluster, 1 for the second cluster, and so on).
NTerms	A number indicating the number of terms in this cluster.
Term_i (where I is 0, 1 ... NTerms-1)	The term at rank I within this cluster. Note that there will be as many Term_i entries as there are terms in the cluster.

JSP code for displaying clusters

You can add a new `nav_clusters.jsp` file for rendering clusters.

Appendix A of this guide includes a new `nav_clusters.jsp` file that can be used as a template for rendering cluster contents. This file should be included at the end of the `nav_controls.jsp` file. Note that it is recommended that you display clusters only if there are two or more.

The highlights of this file are as follows:

1. Get all the `Supplement` objects from the `Navigation` object, which will be encapsulated in a `SupplementList` object (note that the list can also include non-cluster `Supplement` objects):

```
SupplementList navsups = nav.getSupplements();
```

2. Within a For loop (labeled supLoop), get a `Supplement` object and then get that object's properties:

```
Supplement sup = (Supplement)navsups.get(I);
PropertyMap propsMap = sup.getProperties();
```

3. Get the value of the `Dgraph.SeeAlsoCluster` property:

```
String clustersPropName = (String)propsMap.get("DGraph.SeeAlsoCluster");
```

4. Test the value returned from the `Dgraph.SeeAlsoCluster` property. If it is null, then this `Supplement` object is not a cluster and we should loop back to step 2; if the value is non-null, then this is a cluster and we continue to step 5:

```
if (clustersPropName != null)
```

5. If this is the first discovered cluster, then the `clustersOn` variable will be set to false. Therefore, first display the Cluster Discovery header and then set the `clustersOn` variable to true (so that the header will not be displayed again):

```
if (!clustersOn) {
   // display title
   ...
   clustersOn = true;
}
```

6. Get the ranking (`ClusterRank` property) of the cluster and the number of terms (`NTerms` property) it contains. The returned string values are then transformed to integers:

```
String rankString = (String)propsMap.get("ClusterRank");
String nTermsString = (String)propsMap.get("NTerms");
int rank;
int nTerms;
try {
   rank = Integer.parseInt(rankString);
   nTerms = Integer.parseInt(nTermsString);
}
```

7. Within a For loop, retrieve the terms from the `NTerms` property and format them by separating them with commas and spaces. The list of terms will then be:

```
StringBuffer termsSB = new StringBuffer();
StringBuffer termsSBSpace = new StringBuffer();
for (int iTerm = 0; iTerm < nTerms; ++iTerm) {
   String term = (String)propsMap.get("Term_"+iTerm);
   ...
   if (termsSB.length() != 0) {
      termsSB.append(", ");
      termsSBSpace.append(" ");
   }
   termsSB.append(term);
   termsSBSpace.append('"').append(term).append('"');
}
```

8. For a given cluster, begin to create a `UrlENEQuery` request (using the current request), in case the user wants to click on that cluster. Also get the current navigation searches (as an `ERecSearchList`) to determine if the cluster selection is already active in the searches.

```
UrlENEQuery newq = new UrlENEQuery(request.getQueryString(),"UTF-8");
ERecSearchList searches = newq.getNavERecSearches();
```

9. Create a new search (an `ERecSearch` object), using the `clusterPartial` search interface as the search key, the list of related terms (in the `clusterSpace` variable) as the terms for the search, and mode matchpartial as the search option (which specifies `MatchPartial` as the search mode).

```
ERecSearch newSearch = new ERecSearch("clustersPartial",
   clusterSpace, "mode matchpartial");
```

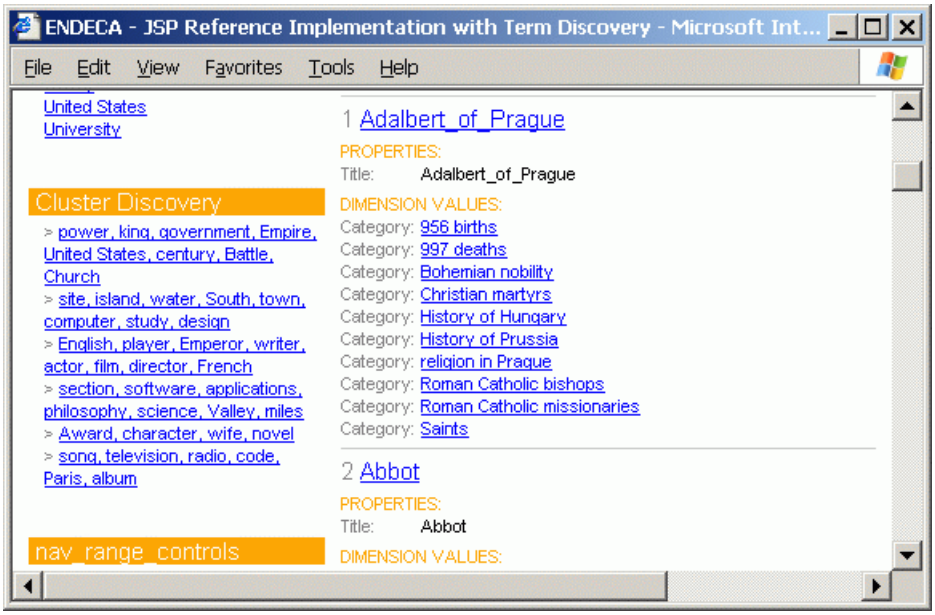
10. Test whether the current navigation searches are null or do not contain the new search (from step 9). If the test is true, then the new search can be added to the `UrlENEQuery` request; if it is false, do not add the new search because the cluster selection is already active.

```
if (searches == null || !searches.contains(newSearch)) {
   ...
   searches.add(newSearch);
   newq.setNavERecSearches(searches);
   ...
}
```

11. Loop back to step 2 to get another `Supplement` object. The loop is done when all the objects in the `SupplementList` have been retrieved.
12. Display the clusters, which are stored as a list of strings in the `clusterStrings` variable. The `clusterUrls` variable is a list of the cluster URLs:

```
for (int I = 0; I < clusterStrings.size(); ++I) {
   %><tr><td><img src="b.gif" width=10></td>
   <td width="100%"><font face="arial" size="1" color="gray">
   > <a href="<%= clusterUrls.get(I) %>">
   <font face="arial" size="1" color="blue">
   <%= clusterStrings.get(I) %></font></a>
   </td></tr><%
}
```

The following is an abbreviated example of the JSP reference implementation showing the clusters rendered by the `nav_clusters.jsp` file. Clicking on a cluster link will execute the partial match query built by steps 8-10.



Clustering overlap properties

Clustering overlap information is also returned by the MDEX Engine.

The `PropertyMap` object (in the cluster `Supplement`) also includes the following set of properties that provide clustering overlap information.

Key Name	Value
Dgraph.SeeAlsoClusterOverlaps	The name of the Term Discovery dimension from which this cluster was generated.
NClusters	A number indicating the number of clusters that were returned by the MDEX Engine.
Cluster_i (where I is 0, 1, ... NClusters-1)	The cluster overlap numbers for a given cluster. Note that the cluster number (the I value) corresponds to the `ClusterRank` value in the `DGraph.SeeAlsoCluster` object.

These properties provide a square matrix that has the cluster overlap numbers. In the matrix, `number(I, j)` is the estimated number of records (from the records sampled from the navigation states) that are covered by both cluster I and cluster j.

Note that from the definition it follows that diagonal numbers (I, I) have the estimated number of records covered by each particular cluster. These diagonal numbers tend to decrease, because of the way that the Cluster Discovery software sorts clusters (by decreasing estimated coverage).

This information can be used in application-specific ways, for example, by an application page that presents a graphical depiction of the clusters.

Displaying records and dimension refinements

Records and refinements from a Term Discovery dimension are displayed like other dimensions.

There is no difference in displaying refinement dimension values from a Term Discovery dimension than from regular dimensions. Information on displaying refinements is found in the *Endeca MDEX Engine Basic Development Guide*, in the chapter titled “Working with Dimensions.”

Likewise, the process of displaying Endeca records generated from Term Discovery refinements is the same as with any Endeca record. For details, see the chapter titled “Working with Endeca Records” in the *Endeca MDEX Engine Basic Development Guide*.

Chapter 6

Term Discovery Advanced Topics

This section discusses advanced topics for Term Discovery applications.

Partial updates for term extraction

Partial updates can be performed in Term Discovery implementations.

As with other types of implementations, performing partial updates in a Term Discovery implementation requires a second pipeline that includes an update adapter. Full details on constructing partial update pipelines are in the *Endeca MDEX Engine Partial Updates Guide*. This section will describe in detail only those parts that are unique to the term extraction pipeline.

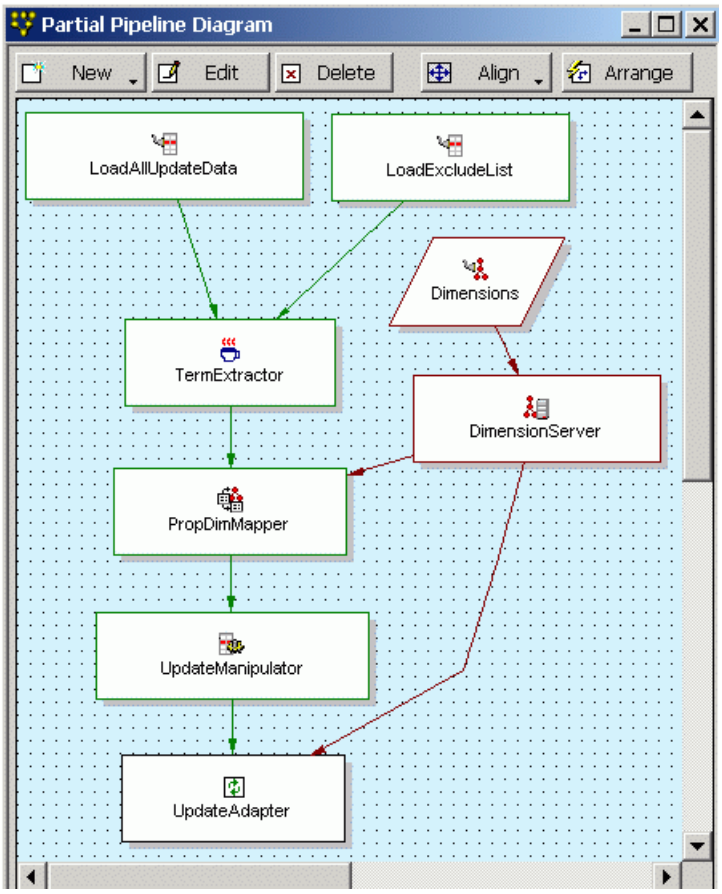
Term extraction prerequisites for partial updates

Term Discovery implementations have additional requirements for partial updates.

The *Endeca MDEX Engine Partial Updates Guide* describes the requirements for the baseline and partial update pipelines. The following additional prerequisites apply to a Term Discovery application performing partial updates:

- In the baseline pipeline, the Java manipulator must have an UPDATE_MODE pass-through with the STATEFUL setting.
- In the partial update pipeline, the Java manipulator must have an UPDATE_MODE pass-through with the PARTIAL setting.
- Both pipelines must use the same name for their Java manipulators. The reason is that the baseline pipeline (with the STATEFUL setting) produces state files whose names are derived from the name of the Java manipulator. The partial update Java manipulator, in turn, also uses its name to look for the term state files produced by the baseline Java manipulator. Thus, both names must be identical.
- The partial update pipeline does not perform corpus-level filtering. Therefore, you do not have to add corpus filtering pass-throughs to the Java manipulator.
- The Java manipulator pass-throughs should be the same in both pipelines (other than the corpus filtering and UPDATE_MODE pass-throughs).

The sample partial update pipeline, shown below, is used to illustrate the various components:



Record adapters for partial updates

The partial update pipeline also has a source record adapter and an exclude record adapter.

The record adapter for the incoming source records is configured the same as the source adapter for the baseline pipeline. The configuration for this record adapter is described in the topic "Creating the record adapter for source records", in the section "Creating the Term Discovery pipeline" of Chapter 4.

The record adapter for the exclude list (which is optional) is also identical to the one described in the topic "Creating the record adapter for the exclude list". Note that the same exclude list is being used as the one in the baseline pipeline.

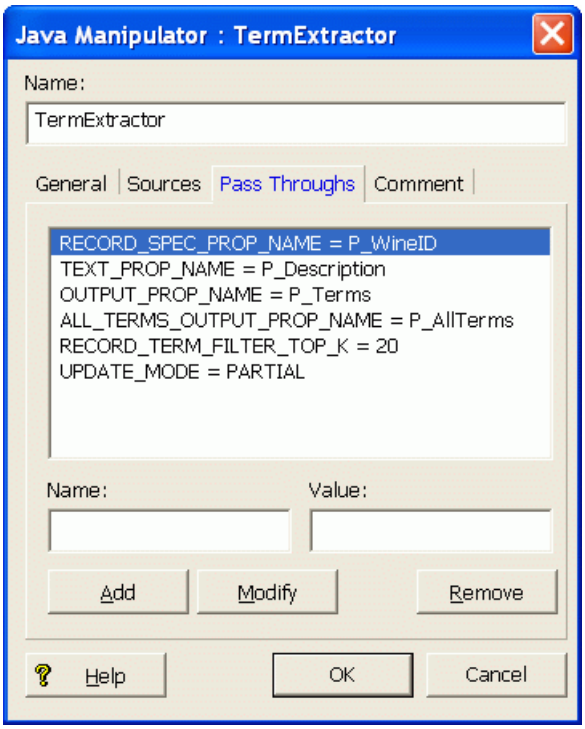
Java manipulator for partial updates

A Java manipulator for term extraction is also used in the partial update pipeline.

The Java manipulator in the sample partial update pipeline is identical to the one in the baseline pipeline, with the exceptions previously noted. In particular, make sure of the following:

- The name of this Java manipulator must be the same as the one in the baseline pipeline.
- This Java manipulator must have an UPDATE_MODE pass-through with the PARTIAL setting.

The Pass Throughs tab of the Java manipulator should look similar to this example:



Term filtering with pre-tagged records

The Term Discovery software can apply filtering to documents that have already been tagged.

The use case described in this scenario involves a corpus in which the documents are tagged with pre-existing terms that were generated and maintained by an external process (i.e., not by the Endeca term extraction software). The goal is to apply corpus-level and record-level filtering to these terms, just as though they had been identified as candidates by the term extractor itself.

There are three variations of this use case:

- Filtering only of pre-tagged terms: No new terms are extracted from the records, but you want filtering to be performed on the pre-tagged terms.
- Uniform filtering on both sets of terms: The term extractor extracts new terms and combines them with the pre-tagged terms. The same filters are applied to both sets of terms equally.
- Filtering only one of the sets of terms: New terms are extracted from the records by the Endeca term extractor. Only the newly-extracted terms are filtered, but the pre-tagged terms are not. After filtering, both sets of terms are combined into one output property and tagged on the records.

Important: In all cases, the pre-tagged source property must contain only one term as its value. Any given record can have multiple instances of this property.

Creating the instance implementation

The back-end implementation for pre-tagged records varies according to the use case.

Creating the instance implementation (pipeline and dimension/property configurations) for these use cases is very similar to that described in Chapter 4. Only one Java manipulator is needed for each case and everything should be done in Developer Studio.

To create the instance implementation for pre-tagged records:

1. In Developer Studio, configure the record specifier property. You can also create an all-terms property if you wish.
2. Configure the related terms dimension, including the clustering parameters.
3. In the Pipeline Diagram, create a record adapter for the source data.
4. Optionally, you can use an exclude list to ensure that unwanted terms are excluded. If so, create a record adapter for the exclude list. (Note that an exclude list can be used with pre-existing terms.)
5. Create and configure the Java manipulator, using the configuration values in the tables in the following three sections. Note that you can also configure the ALL_TERMS_OUTPUT_PROP_NAME and the UPDATE_MODE pass-throughs, according to the application's needs.
6. In the property mapper component, map the OUTPUT_PROP_NAME property to the related terms dimension that you created in step 2. If you created an all-terms property, map that also.

You then run the instance implementation in an Endeca Application Controller environment, as described in the "Running the Term Discovery pipeline" topic in Chapter 4.

The next topics describe the configuration settings for the Java manipulator pass-throughs.

Filtering only pre-existing terms

This use case assumes that the source records have already been tagged with terms.

In this use case, you want to perform corpus- and/or record-level filtering on these pre-tagged terms. However, you do not want the term extractor to extract any more terms from the records.

The Java manipulator that will perform filtering on the pre-tagged terms should have the following configuration values for the pass-throughs:

PASS_THROUGH Element	Configuration Value
RECORD_SPEC_PROP_NAME	Set to the name of the record specifier property.
TEXT_PROP_NAME	Set to the name of a non-existent property, so that no new terms are extracted.
INPUT_TERM_PROP_NAME	Set it to the name of the source property containing the pre-tagged terms.
OUTPUT_PROP_NAME	Set it to the name of the property that is the destination for the pre-existing terms on the Endeca record.
corpus-level pass-throughs	As required by the application. For details, see the "Configuration for corpus-level filtering" topic in Chapter 2.
record-level pass-throughs	As required by the application. For details, see the "Configuration for record-level filtering" topic in Chapter 2.

Filtering both sets of terms uniformly

This use case assumes that you want to perform term extraction on a data set that already has pre-tagged terms.

In this use case, you want to combine both sets of terms and have the same filtering applied to them.

The Java manipulator that will perform uniform filtering on both sets of terms should have the following configuration values for the pass-throughs:

PASS_THROUGH Element	Configuration Value
RECORD_SPEC_PROP_NAME	Set to the name of the record specifier property.
TEXT_PROP_NAME	Set to the name of the source text property from which new terms will be extracted.
INPUT_TERM_PROP_NAME	Set to the name of the source property containing the pre-tagged terms.
OUTPUT_PROP_NAME	Set to the name of the property that is the destination for both newly-tagged terms and pre-existing terms.
corpus-level pass-throughs	As required by the application. For details, see the "Configuration for corpus-level filtering" topic in Chapter 2.
record-level pass-throughs	As required by the application. For details, see the "Configuration for record-level filtering" topic in Chapter 2.

When the term extractor runs, both newly-extracted terms and pre-tagged terms are output to the same OUTPUT_PROP_NAME property. As a result, the same corpus- and record-filtering is applied to all terms.

Filtering only the new terms

This use case assumes that new terms are extracted from the records by the Endeca term extractor, but are not immediately combined with the pre-tagged terms.

In this use case, the newly-extracted terms are not combined at first with the pre-tagged terms. Instead, only the newly-extracted terms are filtered, and the pre-tagged terms are not. After filtering, both sets of terms are combined into one output property and tagged on the Endeca records.

The Java manipulator that will perform this type of filtering should have the following configuration values for the pass-throughs:

PASS_THROUGH Element	Configuration Value
RECORD_SPEC_PROP_NAME	Set to the name of the record specifier property.
TEXT_PROP_NAME	Set to the name of the source text property from which new terms will be extracted.
OUTPUT_PROP_NAME	Set to the name of the source property containing the pre-tagged terms. That is, the destination for the new terms will be the same property as the pre-existing terms.
corpus-level pass-throughs	As required by the application. For details, see the "Configuration for corpus-level filtering" topic in Chapter 2.

PASS_THROUGH Element	Configuration Value
record-level pass-throughs	As required by the application. For details, see the "Configuration for record-level filtering" topic in Chapter 2.

Note that the INPUT_TERM_PROP_NAME pass-through is not used. As a result, the pre-existing terms are not filtered, but are used as-is.

When the term extractor runs, only the newly-extracted terms are filtered with the corpus- and record-level pass-throughs. The filtered terms are then output to the OUTPUT_PROP_NAME property, which is the name of the property with the pre-existing terms. As a result, the same corpus- and record-filtering is applied to all terms. Note that if duplicate values are created for the OUTPUT_PROP_NAME property, they are removed by the property mapper.

Tuning aids for the filtering parameters

This section discusses two tuning aids that will help you when you are tuning the parameters for the corpus- and record-level pass-throughs.

The two tuning aids are:

- STATEFUL update mode
- Corpus-verbose logs

Using STATEFUL mode for tuning

You can use STATEFUL mode to tune the values you set for the corpus-level and record-level pass-throughs in the Java manipulator.

Before you begin, make sure the baseline pipeline has the UPDATE_MODE pass-through set to STATEFUL mode.

The general procedure for using STATEFUL mode for tuning is:

1. Generate a baseline update, index the records, and start the MDEX Engine.
2. Run searches against the P_AllTerms property and check the quality of the clusters.
3. Adjust the corpus-level and/or record-level parameters.
4. Add a MAX_INPUT_RECORDS pass-through set to 0 (zero) to the Java manipulator.
5. Generate another baseline update. The update will be much faster because no terms will be extracted. However, a full corpus- and record-level filtering operation will be performed.
6. Repeat the above steps (except step 4) until you are satisfied with the results.

When you finish, be sure to remove the MAX_INPUT_RECORDS pass-through so all your source records will be processed.

Using corpus-filtering logging statistics

The CORPUS_DEBUG pass-through is helpful for generating debugging information.

The CORPUS_DEBUG pass-through enables the term extractor to log detailed information about the scores it assigns to terms. Temporarily setting this pass-through will help you to tune corpus-level filtering.

The log entries contain four fields of information:

Log Entry	Information
term	The term (noun phrase) that was extracted and filtered.
count	The number of documents in which this term occurs at least once. You can use the RECORD_NTERMS pass-through to set a limit on the number of documents in which a term can occur.
coverage	The coverage score, which is a percentage of all the corpus documents in which this term was found. You can use the CORPUS_MIN_COVERAGE and CORPUS_MAX_COVERAGE pass-throughs to adjust the percentage.
info_gain	the info_gain score, which is a measure of the global informativeness of the term. The CORPUS_MIN_INFO_GAIN and CORPUS_MAX_INFO_GAIN pass-throughs will affect this score.

An example of a term log entry is:

```
term: airport count: 60 coverage: 0.04 info_gain: 1.614589
```

In this example, the term `airport` was found in 60 documents, which is 0.04 (4%) of the 1500-document corpus, and a global informativeness score of 1.614589 was given to the term.

Examining the term extractor logs

This section provides an overview of the term extractor logs.

In addition to the logs, the section also describes how to control the size of the JVM heap (based on errors in the logs) and information on the state files produced by the term extractor.

Term extractor logs

The term extractor writes its log entries to the Forge log.

The term extractor log entries are of these default types:

- INFO
- WARN
- ERROR

`INFO` entries are log messages that indicate information that may be of interest to the user; for example, what the component is doing.

`WARN` entries are warning messages that indicate that processing will continue, but that problems may exist with the source data or the instance configuration. For example, if a noun phrase contains more than five tokens, a warning message similar to this one will warn you that the noun phrase is too long:

```
While processing Record '52755': Overlong noun phrase ending
in 'tannic Shiraz showing spicy oak flavors'
```

`ERROR` messages indicate serious problems that prevent successful processing of the source data. For example, providing an invalid pass-through to the Java manipulator will result in an error like this:

```
(AdapterRunner): Bad value for option UPDATE_MODE: PARTAL
com.endeca.edf.adapter.AdapterException: Bad value for option UPDATE_MODE:
 PARTAL
```

```
  at com.endeca.edf.termextractor.TermExtractor.configure(TermExtractor.ja¬
va:1805)
  at com.endeca.edf.termextractor.TermExtractor.execute(TermExtractor.ja¬
va:1644)
  at com.endeca.edf.adapter.AdapterRunner.main(AdapterRunner.java:204)
```

In this example, the user incorrectly entered "PARTAL" (instead of "PARTIAL") for the UPDATE_MODE pass-through. As a result, an AdapterException was thrown by the Java manipulator.

If you are using the Endeca Application Controller, you can view the Forge log from the EAC Admin Console page in Endeca Workbench.

The Java manipulator also creates a log using this naming format:

```
Edf.Pipeline.RecordPipeline.JavaManipulator.jmname.log
```

where *jmname* is the name of the Java manipulator. The contents of this log are essentially the same as the term extractor entries in the Forge log. This log serves as a backup log in case of a problem with Forge. Note that this log cannot be viewed from Endeca Workbench.

Increasing the JVM heap size

You can increase the size of the JVM heap to fix out-of-memory errors.

The Java manipulator runs under the control of the JVM (Java virtual machine) on your computer. If the Java manipulator requires a large amount of memory, it is possible that the JVM will begin to throw OutOfMemoryError instances when attempting to instantiate objects. The following ERROR log entry is an example:

```
Exception in thread "main" java.lang.Error: Error was thrown at record:
13352
at com.endeca.edf.termextractor.TermExtractor
  $ExtractifiedRecordHandlerSource.produceNextEr
  (TermExtractor.java:711)
...
at com.endeca.edf.adapter.AdapterRunner.main
  (AdapterRunner.java:221)
Caused by: java.lang.OutOfMemoryError: Java heap space
```

As the final line of the log entry indicates, the problem was caused because the Java heap space is too small.

Java has options that help control how much memory it uses:

- `-Xmx` sets the maximum memory heap size.
- `-Xms` sets the minimum memory heap size.

The default size for these values is measured in bytes. Append the letter `K` (or `k`) to the value to indicate kilobytes, `M` (or `m`) to indicate megabytes, and `G` (or `g`) to indicate gigabytes.

You then use the Forge `--javaArgument` flag to pass in the JVM heap size setting to Forge, as in this example that sets the JVM heap size to a maximum of one gigabyte:

```
--javaArgument -Xmx1G
```

Make sure that the heap size is not larger than the available free RAM on your system.

Note that if you want to pass in multiple Java options, you must use a separate `--javaArgument` flag for each option.

Location of term extraction state files

When run in STATEFUL mode, the term extractor produces several state files that are required to support partial updates.

The term extractor state files are stored in directories that use this format:

```
jmname_cs
jmname_ers
```

where *jmname* is the name of the Java manipulator that produced the state file.

For example, if MyTermExtractor is the name of the Java manipulator, then one of the resulting state file will be:

```
MyTermExtractor_cs
```

Two other files are also created (one level above the state directories) with the following name format:

```
jmname_rtss
jmname_ers.count
```

By using the name of the Java manipulator as part of the state files and directories, multiple term extraction components can be used in the pipeline without the problem of one state file overwriting another component's state files.

All term extraction state files are written to the default state directory of Forge. On the EAC Admin Console page (in Endeca Workbench), you can specify another location for the state directory of Forge.

Appendix A

Term Discovery Sample Files

This appendix section contains two JSP files for the UI and a sample record manipulator.

Modified nav_controls.jsp file

This sample `nav_controls.jsp` file is used for the Relationship Discovery UI.

The following `nav_controls.jsp` file has been modified to display refinements from the Terms Discovery dimension. Added or modified code is highlighted in bold face.

```
<%--------------------------------------------------------------------
DESCRIPTION:
This module displays basic, standard navigation controls.  It
is mainly used for debugging purposes and as a starting point for
no-frills navigation solutions. It only displays refinement dimensions,
so this module should be used in conjunction with nav_breadcrumbs.

Copyright (C) 2008 by Endeca Technologies - COMPANY CONFIDENTIAL
--------------------------------------------------------------------%>

<table border="0" cellspacing="0" cellpadding="0" width="100%">
  <tr><td colspan="2" bgcolor="orange"><font face="arial" size=2 col¬
or="white">
   nav_controls:<font></td></tr>
  <tr><td colspan="2"><img src="b.gif" height=3></td></tr>
  <%
  // Get refinement dimension groups
  DimGroupList refDimensionGroups = nav.getRefinementDimGroups();
  // Get descriptor dimensions
  DimensionList descDimensionsNC = nav.getDescriptorDimensions();
  // Output message if no refinement options left
  if (refDimensionGroups.size() == 0) {
    %>
    <tr><td colspan="2"><font face="arial" size=3 color="orange">
    <i>No Additional Query<br>Parameters Available</i></font></td></tr>
    <%
  }
  // Output message if no refinement options have been made
  else if (descDimensionsNC.size() == 0) {
    %>
    <tr><td colspan="2"><font face="arial" size=3 color="orange">
    <i>Query Parameters:</i></font></td></tr>
    <tr><td colspan="2"><img src="b.gif" height=5></td></tr>
```

```
  <%
}
// Header if additional query parameters available
else {
  %>
  <tr><td colspan="2"><font face="arial" size=3 color="orange">
  <i>Additional Query Parameters:</i></font></td></tr>
  <tr><td colspan="2"><img src="b.gif" height=5></td></tr>
  <%
}
// Loop over dimension groups
for (int i=0; i<refDimensionGroups.size(); i++) {
  // Get dimension group object
  DimGroup dg = (DimGroup)refDimensionGroups.get(i);
  //  If group is explicit (not default group), display group
  if (dg.isExplicit()) {
    %>
    <tr><td colspan="2"><img src="b.gif" height="10"></td></tr>
    <tr><td colspan="2"><font face="arial" size="2" color="#999999">
    <%= dg.getName() %></font></td></tr>
    <%
  }
// Loop over dimensions in group
  for (int j=0; j<dg.size(); j++) {
    // Get dimension object
    Dimension dim = (Dimension)dg.get(j);
    // Get root for dimension
    DimVal root = dim.getRoot();
    // Get id of root
    long rootId = root.getId();
    // special handling for Term Discovery dimension
    final boolean isRelTerms = dim.getName().equals(relTermsDimName);
    // Get refinement list for dimension
    DimValList refs = dim.getRefinements();
    // Create request to expose dimension values
    UrlGen urlg = new UrlGen(request.getQueryString(), "UTF-8");
    urlg.removeParam("D");
    urlg.removeParam("Dx");
    urlg.removeParam("sid");
    urlg.removeParam("in_dym");
    urlg.removeParam("in_dim_search");
    urlg.addParam("sid",(String)request.getAttribute("sid"));
    // Expand dimension
    if (!isRelTerms && refs.size() == 0) {
      urlg.addParam("Ne",Long.toString(rootId)+
      (relTermsRootId>= 0? " "+Long.toString(relTermsRootId):""));
    }
    // Close dimension
    else {
      urlg.removeParam("Ne");
    }
    String url = CONTROLLER+"?"+urlg;
    // Display dimension (open row here, close later)
    if (!isRelTerms) {
      %>
      <tr><td colspan="2"><a href="<%= url %>">
      <font face="arial" size="2" color="#444444">
      <%= dim.getName() %></font></a>
    }
    else {
      %>
```

```
      <tr><td colspan="2" bgcolor="orange">
      <font face="arial" size=2 color="white">
       <%=relTermsDisplayString %><font></td></tr>
      <%
    }

    // Get intermediate list for dimension
    DimValList ints = dim.getIntermediates();
    // Loop over intermediate list
    for (int k=0; k < ints.size(); k++) {
      // Get intermediate dimension value
      DimVal intermediate = ints.getDimValue(k);
      // Display intermediate
      %><font face="arial" size="2" color="#444444"> >
      <%= intermediate.getName() %></font><%
    }
    // Close nav row
    %></td></tr><%
    String refinementsColor = "blue";
    Set activeDiscTerms = new HashSet();
    if (isRelTerms) {
      String ntk = (String)request.getParameter("Ntk");
      String ntx = (String)request.getParameter("Ntx");
      if (ntk != null && ntk.equals(P_AllTerms) &&
         "mode matchall".equals(ntx))
      {
        String discTerm = (String)request.getParameter("Ntt");
        if (discTerm != null) {
          if (discTerm.length() >= 3 &&
             discTerm.charAt(0) == '"' &&
             discTerm.charAt(discTerm.length()-1) == '"')
          {
            // remove quotes
            discTerm = discTerm.substring(1, discTerm.length()-1);
          }
          activeDiscTerms.add(discTerm);
        }
      }
    } // if (isRelTerms)

  %><%
    // Loop over refinement list
    for (int k=0; k < refs.size(); k++) {
      // Get refinement dimension value
      DimVal ref = refs.getDimValue(k);
      // Get properties for refinement value
      PropertyMap pmap = ref.getProperties();
      // Get dynamic stats
      String dstats = "";
      if (pmap.get("DGraph.Bins") != null) {
        dstats = " ("+pmap.get("DGraph.Bins")+")";
      }
  %><%
      // Create request to select refinement value
      urlg = new UrlGen(request.getQueryString(), "UTF-8");
      boolean displayRefinement=true;
      if (isRelTerms &&
         (!ref.getName().equals("More...") || !showRelTermsMore))
      {
        if (ref.getName().equals("More...") ||
           ENEQueryToolkit.isImplicitRefinement(dim, ref) ||
```

```
            activeDiscTerms.contains(ref.getName()))
        {
          displayRefinement = false;
        }
        else {
          urlg.addParam("Ntk",P_AllTerms);
          urlg.addParam("Ntt","\""+ref.getName()+"\"");
          urlg.addParam("Ntx","mode matchall");
          urlg.removeParam("No");
          urlg.removeParam("Nao");
          urlg.removeParam("Nty");
          urlg.removeParam("D");
          urlg.removeParam("Dx");
          urlg.removeParam("sid");
          urlg.removeParam("in_dym");
          urlg.removeParam("in_dim_search");
          urlg.addParam("sid",(String)request.getAttribute("sid"));
          url = CONTROLLER+"?"+urlg;
        }
    %><%
    } else {
        // If refinement is navigable, change the Navigation parameter
        if (ref.isNavigable()) {
          urlg.addParam("N",
           (ENEQueryToolkit.selectRefinement(nav,ref)).toString());
          urlg.addParam("Ne",Long.toString(rootId)+
           (relTermsRootId>= 0? " "+Long.toString(relTermsRootId):""));
    }
    // If refinement is non-navigable, change only the
    // exposed dimension parameter
    // (Leave the Navigation parameter as is)
    else {
      urlg.addParam("Ne",Long.toString(ref.getId())+
        (relTermsRootId>= 0? " "+Long.toString(relTermsRootId):""));
    }
    urlg.removeParam("No");
    urlg.removeParam("Nao");
    urlg.removeParam("Nty");
    urlg.removeParam("D");
    urlg.removeParam("Dx");
    urlg.removeParam("sid");
    urlg.removeParam("in_dym");
    urlg.removeParam("in_dim_search");
    urlg.addParam("sid",(String)request.getAttribute("sid"));
    url = CONTROLLER+"?"+urlg;
  }
    // Display refinement
    if (displayRefinement) {
      %>
      <tr><td><img src="b.gif" width=10></td>
      <td width="100%"><a href="<%= url %>">
      <font face="arial" size="1" color="<%=refinementsColor%>">
      <%= ref.getName() %></font></a>
      <font face="arial" size="1" color="gray"><%= dstats %>
      </font></td></tr>
      <%
    }
    } // end of: Loop over refinement list
  } // end of: Loop over dimensions in group
  //  If group is explicit (not default group), display spacer
  if (dg.isExplicit()) {
```

```
      %>
      <tr><td colspan="2"><img src="b.gif" height="10"></td></tr>
      <%
    }

  } // end of: Loop over dimension groups
  %>
  <tr><td colspan="2"><img src="b.gif" height=30></td></tr>
</table>

<%-- Display Clusters Controls --%>
<%@ include file="nav_clusters.jsp" %>

<%-- Display Range Filter Controls --%>
<%@ include file="nav_range_controls.jsp" %>
```

New nav_clusters.jsp file

This `nav_clusters.jsp` sample file is used to render clusters that are generated by the Cluster Discovery feature.

This file should be included in the `nav_controls.jsp` file.

```
<%-----------------------------------------------------------------------
DESCRIPTION:
This module demonstrates the Cluster Discovery feature.
It displays clusters received as Supplemental Objects, makes them
selectable, and, upon selection, generates a search based on
the selected clusters.

This module is included in the nav_controls.jsp module.

Copyright (C) 2008 by Endeca Technologies - COMPANY CONFIDENTIAL
-----------------------------------------------------------------------%>
<%
// Get supplemental list
SupplementList navsups = nav.getSupplements();
boolean clustersOn = false;

// lazily allocated:
List<String> clusterUrls = null;
List<String> clusterStrings = null;

// Loop over cluster supplemental objects
supLoop:
for (int i = 0; i < navsups.size(); ++i) {
  // Get individual see also object
  Supplement sup = (Supplement)navsups.get(i);
  // Get property map
  PropertyMap propsMap = sup.getProperties();

  String clustersPropName = (String)propsMap.get("DGraph.SeeAlsoCluster");

  if (clustersPropName != null) {
    if (!clustersOn) {
      // display title
      %>
      <table border="0" cellspacing="0" cellpadding="0" width="100%">
```

```
      <tr><td colspan="5" bgcolor="orange">
      <font face="arial" size=2 color="white">
      Cluster Discovery</font></td></tr>
      <tr><td colspan="5"><img src="b.gif" height=3></td></tr>
      <%
      clustersOn = true;
    } // end of if !clustersOn
    String rankString = (String)propsMap.get("ClusterRank");
    String nTermsString = (String)propsMap.get("NTerms");
    int rank;
    int nTerms;
    try {
      rank = Integer.parseInt(rankString);
      nTerms = Integer.parseInt(nTermsString);
    } catch (NumberFormatException e) {
    // add code here to log error
      continue supLoop;
    } // end of catch

    StringBuffer termsSB = new StringBuffer();
    StringBuffer termsSBSpace = new StringBuffer();
    for (int iTerm = 0; iTerm < nTerms; ++iTerm) {
      String term = (String)propsMap.get("Term_"+iTerm);
      if (term == null) {
        // add code to log error
        continue supLoop;
      } // end of if termsSB.length
      termsSB.append(term);
      termsSBSpace.append('"').append(term).append('"');
    } // end of for terms
    String clusterX = termsSB.toString();
    String clusterSpace = termsSBSpace.toString();
    // Create request to follow a cluster selection
    // (unless cluster selection already active in the searches)
    UrlENEQuery newq = new UrlENEQuery(request.getQueryString(), "UTF-8");

    ERecSearchList searches = newq.getNavERecSearches();
    ERecSearch newSearch = new ERecSearch("clustersPartial",
       ERecSearch newSearch = new ERecSearch("clustersPartial",

    if (searches == null || !searches.contains(newSearch)) {
      if (searches == null)
        searches = new ERecSearchList();
      if (clusterUrls == null) {
        clusterUrls = new ArrayList<String>();
        clusterStrings = new ArrayList<String>();
      } // end of if clusterUrls
      searches.add(newSearch);
      newq.setNavERecSearches(searches);
      UrlGen hostportq = new UrlGen("", "UTF-8");
      hostportq.addParam("eneHost",
         (String)request.getAttribute("eneHost"));
      hostportq.addParam("enePort",
         (String)request.getAttribute("enePort"));
      String url = CONTROLLER + "?" + hostportq.toString() +
         "&" + UrlENEQuery.toQueryString(newq, "UTF-8");
      clusterUrls.add(url);
      clusterStrings.add(clusterX);
    } // end of if searches
  } // end of if clusterPropName != null
} // end of if clusterPropName != null
```

```
if (clusterStrings != null && clusterStrings.size() > 1) {
  // display clusters only if at least 2.
  for (int i = 0; i < clusterStrings.size(); ++i) {
    %>
    <tr><td><img src="b.gif" width=10></td>
    <td width="100%"><font face="arial" size="1" color="gray">
    > <a href="<%= clusterUrls.get(i) %>">
    <font face="arial" size="1" color="blue">
    <%= clusterStrings.get(i) %></font></a>
    </td></tr>
    <%
  } // end of for clusterStrings
} // end of if clusterStrings

if (clustersOn) {
  // close table
  %>
  <tr><td colspan="2"><img src="b.gif" height=30></td></tr>
  </table>
  <%
} // end of if clusterOn
```

Sample record manipulator for HTML documents

This sample record manipulator cleans HTML documents.

This sample record manipulator, named CleanBody, uses several expressions (including PERL and PARSE_DOC) to remove HTML coding from incoming HTML documents. This cleansing is necessary so that the term extractor can extract terms that do not include HTML tags.

The logic behind cleaning the body text is as follows:

1. Create a property called `body_missing`.
2. If a property named `body` exists, remove the property named `body_missing`.
3. Create a property named `body.mimetype` with value "text/html".
4. Create a property named `body.encoding` with value "utf8".
5. Run the PARSE_DOC expression from `body[.*]` to create a property named `text`.
6. Remove all the `body[.*]` properties.
7. Rename the text property to `body`.

The resulting body property is then used as the source property for the TEXT_PROP_NAME pass-through for the Java manipulator.

Code for the sample record manipulator

```
<RECORD_MANIPULATOR FRC_PVAL_IDX="TRUE" NAME="CleanBody">
  <COMMENT>Provides expressions to clean HTML body text.</COMMENT>
  <RECORD_SOURCE>LoadAllMainData</RECORD_SOURCE>
  <EXPRESSION LABEL="" NAME="CREATE" TYPE="VOID" URL="">
    <EXPRNODE NAME="PROP_NAME" VALUE="body.missing"/>
    <EXPRESSION LABEL="" NAME="CONST" TYPE="STRING" URL="">
      EXPRNODE NAME="VALUE" VALUE="true"/>
    </EXPRESSION>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="IF" TYPE="VOID" URL="">
```

```
    <EXPRESSION LABEL="" NAME="PROP_EXISTS" TYPE="INTEGER" URL="">
      <EXPRNODE NAME="PROP_NAME" VALUE="body"/>
    </EXPRESSION>
    <EXPRESSION LABEL="" NAME="REMOVE" TYPE="VOID" URL="">
      <EXPRNODE NAME="PROP_NAME" VALUE="body.missing"/>
    </EXPRESSION>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="IF" TYPE="VOID" URL="">
    <EXPRESSION LABEL="" NAME="PROP_EXISTS" TYPE="INTEGER" URL="">
      <EXPRNODE NAME="PROP_NAME" VALUE="body.missing"/>
    </EXPRESSION>
    <EXPRESSION LABEL="" NAME="CREATE" TYPE="VOID" URL="">
      <EXPRNODE NAME="PROP_NAME" VALUE="body"/>
      <EXPRESSION LABEL="" NAME="IDENTITY" TYPE="PROPERTY" URL="">
        <EXPRNODE NAME="PROP_NAME" VALUE="lp"/>
      </EXPRESSION>
    </EXPRESSION>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="CREATE" TYPE="VOID" URL="">
    <EXPRNODE NAME="PROP_NAME" VALUE="body.mimetype"/>
    <EXPRESSION LABEL="" NAME="CONST" TYPE="STRING" URL="">
      <EXPRNODE NAME="VALUE" VALUE="text/html"/>
    </EXPRESSION>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="CREATE" TYPE="VOID" URL="">
    <EXPRNODE NAME="PROP_NAME" VALUE="body.encoding"/>
    <EXPRESSION LABEL="" NAME="CONST" TYPE="STRING" URL="">
      <EXPRNODE NAME="VALUE" VALUE="utf8"/>
    </EXPRESSION>
  </EXPRESSION>

  <EXPRESSION LABEL="" NAME="PERL" TYPE="VOID" URL="">
    <EXPRBODY>
<![CDATA[my @bodies = get_props_by_name("body");
foreach my $body (@bodies) {
  my $value = $body->value;
  # Replace HTML block elements with end-of-sentence punctuation.
  $value =~
  s/<\/?(?:H[123456])|(?:[OU]L)|(?:LI)|(?:D[LDT])|(?:PRE)|(?:DIV)|
  (?:NOSCRIPT)|(?:BLOCKQUOTE)|(?:FORM)|(?:TABLE)|(?:TH)|(?:TD)|(?:HR)|
  (?:FIELDSET)|(?:ADDRESS)>/. /g;
  $body->value($value);
}
replace_props("body", @bodies);]]>
    </EXPRBODY>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="PARSE_DOC" TYPE="VOID" URL="">
    <EXPRNODE NAME="FILE_PATH" VALUE="FALSE"/>
    <EXPRNODE NAME="PARSE_META" VALUE="FALSE"/>
    <EXPRNODE NAME="MIMETYPE_PROP" VALUE="body.mimetype"/>
    <EXPRNODE NAME="ENCODING_PROP" VALUE="body.encoding"/>
    <EXPRNODE NAME="BODY_PROP" VALUE="body"/>
    <EXPRNODE NAME="TEXT_PROP" VALUE="text"/>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="REMOVE" TYPE="VOID" URL="">
    <EXPRNODE NAME="PROP_NAME" VALUE="body"/>
    <EXPRNODE NAME="PROP_NAME" VALUE="body.missing"/>
    <EXPRNODE NAME="PROP_NAME" VALUE="body.mimetype"/>
    <EXPRNODE NAME="PROP_NAME" VALUE="body.encoding"/>
  </EXPRESSION>
  <EXPRESSION LABEL="" NAME="RENAME" TYPE="VOID" URL="">
```

```
    <EXPRNODE NAME="OLD_NAME" VALUE="text"/>
    <EXPRNODE NAME="NEW_NAME" VALUE="body"/>
  </EXPRESSION>

  <EXPRESSION LABEL="" NAME="PERL" TYPE="VOID" URL="">
    <EXPRBODY>
    <![CDATA[my @bodies = get_props_by_name("body");
foreach my $body (@bodies) {
  my $value = $body->value;
  # Move filing date to a separate property
  if ($value =~ s/Filed at (.*)//) {
    my $filed = $1;
    add_props(new Zinc::PropVal("filed", $filed));
  }
  # Remove leading newlines
  $value =~ s/^(?:\r?\n)+//;
  # Remove byline
  $value =~ s/^([a-zA-Z,. ]+ \([A-Z]+\)) ?-- ?//;
  # Add sentence endings to things that look like lists
  $value =~ s/([^.])(?:\r?\n)+([0-9])/$1. \n$2/g;
  $body->value($value);
}
replace_props("body", @bodies);]]>
    </EXPRBODY>
  </EXPRESSION>
</RECORD_MANIPULATOR>
```

Index

L

M

N

O

P

R

S

T

U