Oracle Endeca Commerce

MDEX Engine Advanced Development Guide

Version 6.4.0 • December 2012

ORACLE®

ENDECA

Contents

Copyright and disclaimer

Preface

The Oracle Endeca 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 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 how to develop an Oracle Endeca Commerce implementation that includes more advanced features. Advanced features may include using internationalized source data, EQL, bulk export operations, or query debugging.

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

For basic development tasks, see the *MDEX Engine Basic Development Guide.*

Who should use this guide

This guide is intended for developers who are building Endeca applications and would like to use advanced features.

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*.

Part 1

Advanced Record Features

- *Controlling Record Values with the Select Feature*
- *Using the Endeca Query Language*
- *Record Filters*
- *Bulk Export of Records*

Chapter 1

Controlling Record Values with the Select Feature

This section describes how to use the Select feature for selecting specific keys (Endeca properties and/or dimensions) from the data so that only a subset of values is returned for Endeca records in a query result set.

About the Select feature

Your application can return record sets based on specific keys.

A set of Endeca records is returned with every navigation query result. By default, each record includes the values from all the keys (properties and dimensions) that have record page and record list attributes. These attributes are set with the **Show with Record** (for record page) and **Show with Record List** (for record list) checkboxes, as configured in Developer Studio.

However, if you do not want all the key values, you can control the characteristics of the records returned by navigation queries by using the Select feature.

The Select feature allows you to select specific keys (Endeca properties and/or dimensions) from the data so that only a subset of values will be transferred for Endeca records in a query result set. The Select functionality allows the application developer to determine these keys dynamically, instead of at Dgraph startup. This selection will override the default record page and record list fields.

A Web application that does not make use of all of the properties and dimension values on a record can be more efficient by only requesting the values that it will use. The ability to limit what fields are returned is useful for exporting bulk-format records and other scenarios. For example, if a record has properties that correspond to the same data in a number of languages, the application can retrieve only the properties that correspond to the current language. Or, the application may render the record list using tabs to display different sets of data columns (e.g., one tab to view customer details and another to view order details without always returning the data needed to populate both tabs).

This functionality prevents the transferring of unneeded properties and dimension values when they will not be used by the front-end Web application. It therefore makes the application more efficient because the unneeded data does not take up network bandwidth and memory on the application server.

The Select feature can also be used to specifically request fields that are not transferred by default.

Configuring the Select feature

No system configuration is required for the Select feature.

In other words, no instance configuration is required in Developer Studio and no Dgidx or Dgraph flags are required to enable selection of properties and dimensions. Any existing property or dimension can be selected.

URL query parameters for Select

There is no Select-specific URL query parameter.

A query for selected fields is the same as any valid navigation query. Therefore, the Navigation parameter (`N`) is required for the request

Selecting keys in the application

With the Select feature, the Web application can specify which properties and dimensions should be returned for the result record set from the navigation query.

The specific selection method used by the application depends on whether you have a Java or .NET implementation.

Java selection method

Use the `ENEQuery.setSelection()` method for Java implementations.

For Java-based implementations, you set the selection list on the `ENEQuery` object with the `setSelection()` method, which has this syntax:

```
ENEQuery.setSelection(FieldList selectFields)
```

where *selectFields* is a list of property or dimension names that should be returned with each record. You can populate the `FieldList` object with string names (such as "P_WineType") or with Property or Dimension objects. In the case of objects, the `FieldList.addField()` method will automatically extract the string name from the object and add it to the `FieldList` object.

During development, you can use the `ENEQuery.getSelection()` method (which returns a `FieldList` object) to check which fields are set.

The `FieldList` object will contain a list of Endeca property and/or dimension names for the query. For details on the methods of the `FieldList` class, see the Endeca Javadocs for the Presentation API.

> 🖉 **Note:** The `setSelection()` and `getSelection()` methods are also available in the `UrlENEQuery` class.

Java Select example

The following is a simple Java example of setting an Endeca property and dimension for a navigation query. When the `ENEQueryResults` object is returned, it will have a list of records that have been

tagged with the P_WineType property and the Designation dimension. You extract the records as with any record query.

```
// Create a query
ENEQuery usq = new UrlENEQuery(request.getQueryString(),"UTF-8");
// Create an empty selection list
FieldList fList = new FieldList();
// Add an Endeca property to the list
fList.addField("P_WineType");
// Add an Endeca dimension to the list
fList.addField("Designation");
// Add the selection list to the query
usq.setSelection(fList);
// Make the MDEX Engine query
ENEQueryResults qr = nec.query(usq);
```

.NET selection property

Use the `ENEQuery.Selection()` property for Java implementations.

In a .NET application, the `ENEQuery.Selection` property is used to get and set the `FieldList` object. You can add properties or dimensions to the `FieldList` object with the `FieldList.AddField` property.

Note: The `Selection` property is also available in the `UrlENEQuery` class.

.NET selection example

The following is a C# example of setting an Endeca property and dimension for a navigation query.

```
// Create a query
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");
// Create an empty selection list
FieldList fList = new FieldList();
// Add an Endeca property to the list
int i = fList.AddField("P_WineType");
// Add an Endeca dimension to the list
i = fList.AddField("Designation");
// Add the selection list to the query
usq.Selection = fList;
// Make the MDEX Engine query
ENEQueryResults qr = nec.query(usq);
```

Chapter 2

Using the Endeca Query Language

This section describes how to use the Endeca Query Language, which allows you to create various types of record filters when making navigation queries for record search.

About the Endeca Query Language

The Endeca Query Language (EQL) contains a rich syntax that allows an application to build dynamic, complex filters that define arbitrary subsets of the total record set and restrict search and navigation results to those subsets.

Besides record search, these filters can also be used for dimension search. EQL is available as a core feature of Oracle Endeca Guided Search with the capabilities listed in the next section, "Basic filtering capabilities". In addition, Record Relationship Navigation (RRN) (described in the topic "Record Relationship Navigation module") is available as an optional module that extends the MDEX Engine capability.

Basic filtering capabilities

You can use EQL to create an expression that can filter on different features.

These include:

- Dimension values
- Specific property values
- A defined range of property values (range filtering)
- A defined range of geocode property values (geospatial filtering)
- Text entered by the user (record search)

The language also supports standard Boolean operators (`and`, `or`, and `not`) to compose complex expressions. In addition, EQL requests can be combined with other Endeca features, such as spelling auto-correction, Did You Mean suggestions, and the sorting parameters (`Ns` and `Nrk`). Details on these interactions are provided in "Endeca Query Language and other features."

Record Relationship Navigation module

The Record Relationship Navigation (RRN) module is an optional module that is intended for use with complex relational data.

Only customers entitled to the new module can pose queries that join records at query time and navigate based on the connected relationships.

This module is intended for sites that have different types of records, in which properties in one record type have values that ultimately refer to properties in another record type. For example, an Author record type can have an author_bookref property with the ID of a Book record type. In this case, you can leave the records uncombined (when the pipeline is run) and then have the MDEX Engine apply a relationship filter among the record types with an RRN request.

Among the benefits of query-time relationship filters are:

- Reduced memory footprint: With no need to combine different types of records in the pipeline, this will reduce the memory footprint of the MDEX Engine, allowing more data to fit into a single engine.
- Reduced application complexity: With the MDEX Engine handling the data relationships, custom application logic will be greatly simplified.
- Improved performance: RRN improves query performance by removing the need to query the MDEX Engine multiple times, thereby reducing the data being transferred over the network.

For details on constructing these types of requests, see "Record Relationship Navigation queries."

Endeca Query Language syntax

The following EBNF grammar describes the syntax for EQL filter expressions.

```
RecordPath ::= Collection "(" ")" "/" RecordStep
Collection ::= FnPrefix? "collection"
FnPrefix ::= "fn" ":"
RecordStep ::= "record" Predicate?
Predicate ::= "[" Expr "]"
Expr ::= OrExpr
OrExpr ::= AndExpr ("or" AndExpr)*
AndExpr ::= NotExpr ("and" NotExpr)*
NotExpr ::= PrimaryExpr | (FnPrefix? "not" "(" Expr ")")
PrimaryExpr ::= ParenExpr | TestExpr
ParenExpr ::= "(" Expr ")"
TestExpr ::= ComparisonExpr | FunctionCall
FunctionCall ::= TrueFunction | FalseFunction | MatchesFunction
TrueFunction ::= FnPrefix? "true" "(" ")"
FalseFunction ::= FnPrefix? "false" "(" ")"
MatchesFunction ::= "endeca" ":" "matches" "(" "." ","
 StringLiteral "," StringLiteral ( "," StringLiteral ( ","
 StringLiteral ( "," (TrueFunction | FalseFunction) )? )? )?
 ")"
ComparisonExpr ::= LiteralComparison | JoinComparison
 | RangeComparison | GeospatialComparison
 | DimensionComparison
EqualityOperator ::= "=" | "!="
LiteralComparison ::= PropertyKey EqualityOperator Literal
JoinComparison ::= PropertyKey "=" PropertyPath
RangeComparison ::= PropertyKey RangeOperator NumericLiteral
GeospatialComparison ::= "endeca" ":" "distance" "("
 PropertyKey "," "endeca" ":" "geocode" "(" NumericLiteral ","
 NumericLiteral ")" ")" (">" | "<") NumericLiteral
DimensionComparison ::= DimensionKey EqualityOperator
 (DimValById | DimValPath) "//" "id"
DimValById ::= "endeca" ":" "dval-by-id" "(" IntegerLiteral ")"
DimValPath ::= Collection "(" "" "dimensions" "" ")"
 ("/" DValStep)*
```

```
DValStep ::= ("*" | "dval") "[" "name" "=" StringLiteral "]"
DimensionKey ::= NCName
PropertyPath ::= RecordPath "/" PropertyKey
PropertyKey ::= NCName
RangeOperator ::= "<" | "<=" | ">" | ">="
Literal ::= NumericLiteral | StringLiteral
NumericLiteral ::= IntegerLiteral | DecimalLiteral
StringLiteral ::= '"' ('""' | [^"])* '"'
IntegerLiteral ::= [0-9]+
DecimalLiteral ::= ([0-9]+ "." [0-9]*) | ("." [0-9]+)
```

The EBNF uses these notations:

- + means 1 or more instances of a component
- ? means 0 or 1 instances
- * means 0 or more instances

The EBNF uses the same Basic EBNF notation as the W3C specification of XML, located at this URL:
http://www.w3.org/TR/xml/#sec-notation

Also, note these important items about the syntax:

- Keywords are case sensitive. For example, "endeca:matches" must be specified in lower case, as must the and and or operators.
- The names of keywords are not reserved words across the Endeca namespace. For example, if you have a property named collection, its name will not conflict with the name of the collection() function.
- To use the double-quote character as a literal character (that is, for inclusion in a string literal), it must be escaped by prepending it with a double-quote character.

These and other aspects of EQL will be discussed further in later sections of this section.

Negation operators

EQL provides two negation operators.

As the EBNF grammar shows, EQL provides two negation operators:

- The **not** operator
- The **!=** operator

An example of the **not** operator is:

```
collection()/record[not(Recordtype = "author")]
```

An example of the **!=** operator is:

```
collection()/record[Recordtype != "author"]
```

Although both operators look like they work the same, each in fact may return a different record set. Using the above two sample queries:

- The **not** operator example returns any record which does not have a Recordtype property with value "author" (including records which have no Recordtype properties at all).
- The **!=** operator returns only records which have non-"author" Recordtype property values. This operator excludes records which have no Recordtype properties.

The small (but noticeable) difference in the result sets may be a useful distinction for your application.

Using negation on properties

EQL supports filtering by the absence of assignments on records. By using the **not** operator, you can filter down to the set of records which do *not* have a specific property assignment.

For example:

```
collection()/record[author_id]
```

returns all records *with* the "author_id" property, while:

```
collection()/record[not (author_id)]
```

returns all records *without* the "author_id" property.

NCName format for properties and dimensions

With a few exceptions (noted when applicable), the names of Endeca properties and dimensions used in EQL requests must be in an NCName format.

(This restriction does not apply to the names of non-root dimension values or to the names of search interfaces.) The names are also case sensitive when used in EQL requests.

The NCName format is defined in the W3C document Namespaces in XML 1.0 (Second Edition), located at this URL: *http://www.w3.org/TR/REC-xml-names/#NT-NCName*

As defined in the W3C document, an NCName must start with either a letter or an underscore (but keep in mind that the W3C definition of Letter includes many non-Latin characters). If the name has more than one character, it must be followed by any combination of letters, digits, periods, dashes, underscores, combining characters, and extenders. (See the W3C document for definitions of combining characters and extenders.) The NCName cannot have colons or white space.

Take care when creating property names in Developer Studio, because that tool allows you to create names that do not follow the NCName rules. For example, you can create property names that begin with digits and contain colons and white space. Any names which do not comply with NCName formatting will generate a warning when running your pipeline.

The property must also be explicitly enabled for use with record filters (not required for record search queries). Dimension values are automatically enabled for use in record filtering expressions, and therefore do not require any special configuration.

URL query parameters for the Endeca Query Language

The MDEX Engine URL query parameters listed in this topic are available to control the use of EQL requests.

- `Nrs` - The `Nrs` parameter specifies an EQL request that restricts the results of a navigation query. This parameter links to the Java `ENEQuery.setNavRecordStructureExpr()` method and the .NET `ENEQuery.NavRecordStructureExpr` property. The `Nrs` parameter has a dependency on the `N` parameter, because a navigation query is being performed.
- `Ars` - The `Ars` parameter specifies an EQL request that restricts the results of an aggregated record query. This parameter links to the Java `ENEQuery.setAggrERecStructureExpr()` method and the .NET `ENEQuery.AggrERecStructureExpr` property. The `Ars` parameter has a dependency on the `A` parameter, because an aggregated record query is being performed.
- `Drs` - The `Drs` parameter specifies an EQL request that restricts the set of records considered for a dimension search. Only dimension values represented on at least one record satisfying the filter are returned as search results. This parameter links to the Java

ENEQuery.setDimSearchNavRecordStructureExpr() method and the .NET ENEQuery.Dim¬
SearchNavRecordStructureExpr property. The Drs parameter has a dependency on the D
parameter.

These parameters (including the EQL expression) must be URL-encoded. For example, this query:

```
collection()/record[Recordtype = "author"]
```

should be issued in this URL-encoded format:

```
collection%28%29/record%5BRecordtype%20%3D%20%22author%22%5D
```

However, the examples in this chapter are not URL-encoded, in order to make them easier to
understand.

Making Endeca Query Language requests

The collection() function is used to query the MDEX Engine for a set (that is, a collection) of
Endeca records, based on an expression that defines the records you want.

EQL allows you to make the following types of requests, all of which begin with the collection()
function:

- Property value query
- Record Relationship Navigation query
- Dimension value query
- Record search query
- Range filter query

The basic syntax for the collection() function is:

```
fn:collection()/record[expression]
```

The fn: prefix is optional, and for the sake of brevity will not be used in the examples in this chapter.
The /record step indicates that Endeca records are being selected. The expression argument
(which is called the predicate) is an EQL expression that filters the total record set to the subset that
you want. The predicate can contain one or more collection() functions (multiple functions are
nested).

Issuing the collection() function without a predicate (that is, without an expression) returns the
total record set because the query is not filtering the records. This query is therefore the same as
issuing only an N=0 navigation query, which is a root navigation request.

The following sample query illustrates the use of the collection() function with the Nrs parameter:

```
controller.jsp?N=0&Nrs=collection()/record[book_id = 8492]
```

Because EQL is a filtering language, it does not have a built-in sorting option. Therefore, an EQL
request returns the record set using the MDEX Engine default sort order. You can, however, append
a URL sorting parameter, such as the Ns parameter or the Nrk, Nrt, Nrr, and Nrm set of parameters.
For more information on the interaction with other Endeca features, see "Endeca Query Language
and other features."

Property value queries

Property value queries (also called literal comparison queries) return those records that have a property
whose value on the records is equal to a specified literal value.

The syntax for this type of query is:

```
collection()/record[propertyName = literalValue]
```

where:

- `propertyName` is the NCName of an Endeca property that is enabled for record filters. Dimension names are not supported for this type of query.
- `literalValue` is a number (either integer or floating point) or a quoted string literal. Numbers are not quoted. For a record to be returned, the value of `literalValue` must exactly match the value of `propertyName`, including the case of the value for quoted string literals. Wildcards are not supported, even if the property has been enabled for wildcard search.

Because it is a predicate, the expression must be enclosed within square brackets. Expressions can be nested.

Note that you can use one of the negation operators described in the "Negation operators" topic.

Examples

The first example illustrates a simple property comparison query:

```
collection()/record[Recordtype = "author"]
```

This query returns all records that have a property named `Recordtype` whose value is "author". If a `Recordtype` property on a record has another value (such as "editor"), then that record is filtered out and not returned.

The second example illustrates how to use the `and` operator:

```
collection()/record[author_nationality = "english"
 and author_deceased = "true"]
```

This query returns all `Author` records for English writers who are deceased.

Record Relationship Navigation queries

EQL allows you to issue a request against normalized records, using record-to-record relationship filter expressions. These types of requests are called Record Relationship Navigation (RRN) queries.

If you have different record types in your source data, you can keep the records uncombined by using a Switch join in your pipeline. Then, by issuing an RRN query, the MDEX Engine can apply a relationship filter to the records at query time. Depending on how you have tagged the properties on the records, an RRN query can return records of only one type or of multiple types.

For example, assume that you want to have three record types (`Author` records, `Book` records, and `Editor` records). To define the record type, all the records have a property named `Recordtype` (the actual name does not matter). `Author` records have this property set to "author", `Book` records have it set to "book", and `Editor` records use a value of "editor". In your pipeline, you use a Switch join to leave those records uncombined. You can then filter `Book` records via relationship filters with `Author` and `Editor` records, but the returned records (and their dimension refinements) will be `Book` records only. This means that any other query parameters apply only to the record type that is returned.

> **Note:** You must configure the MDEX Engine in order to enable RRN. This capability is an optional module that extends the MDEX Engine. Endeca customers who are entitled by their license to use RRN can find instructions on the Endeca Support site. Contact your Endeca representative if you need to obtain an RRN license.

Record Relationship Navigation query syntax

This topic describes the syntax for RRN queries.

The basic syntax for an RRN query is:

```
collection()/record[propertyKey1 = recordPath/propertyKey2]
```

where:

- `propertyKey1` is the NCName of an Endeca property on a record type to be filtered. The resulting records will have this property.
- `recordPath` is one or more `collection()/record` functions.
- `propertyKey2` is the NCName of an Endeca property on another record type that will be compared to `propertyKey1`. Records that satisfy the comparison will be added to the returned set of records.

The forward slash (`/`) character is required between `recordPath` and `propertyKey2` because `propertyKey2` is a `property` step.

There are two ways to differentiate RRN queries from other types of EQL requests:

- RRN queries have a `collection()/record` function on the right side of the comparison operator in the predicate.
- They include a property step.

The following example illustrates a basic relationship filter query:

```
collection()/record[author_bookref =
 collection()/record[book_year = "1843"]/book_id]
```

In this example, the `author_bookref` is a property of `Author` records, which means that `Author` records are returned. These records are filtered by the `book_year` and `book_id` properties of the `Book` records. The `author_bookref` property is a reference to the `book_id` property (which is being used as the property step). Therefore, the query returns `Author` records for authors who wrote books that were published in 1843. There is an inner `collection()/record` function (which uses `book_year` as its property key) on the right side of the comparison expression.

The above query example is shown in a linear format. The query can also be made in a structured format, such as the following:

```
collection()/record
[
 author_bookref = collection()/record
 [
  book_year = "1843"
 ]
 /book_id
]
```

This structured format will be used for most of the following examples, as it makes it easier to parse the query.

Relationship filter expressions work from the inside out (that is, the inner expressions are performed before the outer ones). In this example, the MDEX Engine first processes the `Book` records to come up with a set of `Book` records that have the `book_year` property set to "1843". Then the `book_id` property values of the `Book` records provide a list of IDs that are used as indices to filter the `Author` records (that is, as comparisons to the `author_bookref` property).

Record Relationship Navigation query examples

This topic contains examples of RRN queries.

The following examples assume that you have three record types in your source data: `Author` records, `Book` records, and `Editor` records. While all records have several properties, the `Author` and `Book` records have these properties that establish a relationship between the record types:

- `Author` records have an `author_bookref` property that references the `book_id` property of `Book` records. In addition, `Author` records have an `author_editorref` property that references the `editor_id` property of an `Editor` record.
- `Book` records have a `book_authorref` property that references the `author_id` property of `Author` records.

Using these cross-record reference properties, an RRN query can apply relationship filters between the record types.

RRN relationship filter examples

These examples illustrate how to build relationship filter queries.

The user may first issue a query for `Editor` records for an editor named Jane Smith who works in the city of Boston:

```
collection()/record
[
 editor_name = "Jane Smith"
 and
 editor_city = "Boston"
]
```

The query is then modified for `Author` records:

```
collection()/record
[
 author_editorref = collection()/record
 [
  editor_name = "Jane Smith"
  and
  editor_city = "Boston"
 ]
 /editor_id
]
```

The query returns all `Author` records filtered by the results of the `Editor` records. That is, the `Author` records are filtered by the `editor_id` property of the `Editor` records (which are referenced by the `author_editorref` property in the `Author` records).

The next example query returns books by American authors:

```
collection()/record
[
 book_authorref = collection()/record
 [
  author_nationality = "american"
 ]
 /author_id
]
```

The next example query returns all books by authors who are still alive:

```
collection()/record
[
 book_authorref = collection()/record
 [
```

```
 author_deceased ="false"
 ]
 /author_id
]
```

The next example query combines the two previous examples, and also illustrates the use of the `or` operator. Both inner `collection()/record` functions use the `author_id` property value results as indices for the `Book` records.

```
collection()/record
[
 book_authorref = collection()/record
 [
  author_nationality = "american"
 ]
 /author_id
 or
 book_authorref = collection()/record
 [
  author_deceased="false"
 ]
 /author_id
]
```

The next example query returns the books written by authors who have had those books published in a hard-cover format.

```
collection()/record
[
 book_authorref=collection()/record
 [
  author_bookref=collection()/record
  [
   book_cover = "hard"
  ]
  /book_id
 ]
 /author_id
]
```

The next query example extends the previous one by returning the books written by authors who have published hard-cover books and have worked with an editor named "Jane Smith". The query also shows how to apply relationship filters among all three record types.

```
collection()/record
[
 book_authorref=collection()/record
 [
  author_bookref=collection()/record
  [
   book_cover="hard"
  ]
  /book_id
  and
  author_editorref=collection()/record
  [
   editor_name="Jane Smith"
  ]
  /editor_id
 ]
 /author_id
]
```

In the final example, this powerful query returns all books written by the author of a play titled "The Island Princess" (which was written by English playwright John Fletcher) and also all books that were written by authors who co-wrote books with Fletcher. The result set will include plays that were written either by Fletcher or by anyone who has ever co-authored a play with Fletcher.

```
collection()/record
[
 book_authorref = collection()/record
 [
  author_bookref = collection()/record
  [
   book_authorref = collection()/record
   [
    author_bookref = collection()/record
    [
     book_title = "The Island Princess"
    ]
    /book_id
   ]
   /author_id
  ]
  /book_id
 ]
 /author_id
]
```

Dimension value queries

Dimension value queries allow you to filter records by dimension values. The dimension value used for filtering can be any dimension value in a flat dimension or in a dimension hierarchy.

Rules for the naming format of the dimension value are as follows:

- For a root dimension value (which has the same name as the dimension), the name must be in the NCName format.
- For a non-root dimension value (such as a leaf), the name does not have to be in the NCName format.

There are two syntaxes for using dimension values to filter records, depending on whether you are specifying a dimension value path or an explicit dimension value node.

Querying with dimension value paths

The syntax described in this topic specifies a dimension value path to the `collection()/record` function.

The path can specify just the root dimension value, or it can traverse part or all of a dimension hierarchy. The query will return all records that are tagged with the specified dimension value and with descendants (if any) of that dimension value.

Use the following steps to construct a dimension value path:

1. The path must start with the NCName of the dimension from which the dimension values will be filtered. Tthe dimension name is not quoted and is case sensitive:`[dimName = collection("di¬ mensions")`

2. It must then be followed by a slash-separated step specifier that uses the dval keyword (or the *
   keyword, both are interchangeable) and the name of the root dimension value, which is the same
   name as the dimension name. The name is case sensitive and must be within double quotes:
   `/dval[name = "dvalName"]` or `/*[name = "dvalName"]`.
3. Optionally, you can use one or more slash-separated step specifiers to specify a path to a dimension
   value descendant. These step specifiers use the same syntax as described in the previous step.
   Names of descendant dimension values do not have to be in the NCName format.
4. The dimension value path must be terminated with `//id`. The `//id` path terminator specifies that
   the path be extended to any descendants of the last specified dvalName dimension value. The
   resulting syntax is: `collection()/record[dimName = collection("dimen¬`
   `sions")/dval[name = "dvalName"]//id`.

Note that you can use one of the negation operators described in the "Negation operators" topic.

Query examples using dimension value paths

The examples in this topic illustrate how to construct dimension value paths using EQL syntax rules.

The examples use the Genre dimension, which has this hierarchy:

```
Genre
 Fiction
  Classic
   Literature
  Science-fiction
 Non-fiction
```

The Fiction dimension value has two descendants (Classic and Science-fiction), while the Non-fiction
dimension value has no descendants (that is, it is a leaf dimension value).

The first example query is made against the dimension named Genre (the `dimName` argument). It
uses one step specifier for the root dimension value (also named Genre). The query returns all records
that are tagged with the Genre dimension value, including all its descendants (such as the Classic
dimension value).

```
collection()/record
[
 Genre = collection("dimensions")/dval[name="Genre"]//id
]
```

The next example query uses two step specifiers in the predicate. The dimension value path begins
with the dimension name (Genre), followed by the root dimension value name (also Genre), and finally
the Fiction child dimension value. The query returns all records that are tagged with the Fiction
dimension value, including its three descendants (Classic, Literature, and Science-fiction). Records
tagged only with the Non-fiction dimension value are not returned because it is not a descendant of
Fiction.

```
collection()/record
[
 Genre = collection("dimensions")/dval[name="Genre"]
  /dval[name="Fiction"]//id
]
```

The next example query uses three step specifiers to drill down to the Classic dimension value, which
is a descendant of Fiction. The query returns all records that are tagged with the Classic dimension

value or its Literature descendant. The example also shows the use of * (instead of dval) in the step specifier.

```
collection()/record
[
 Genre =collection("dimensions")/*[name="Genre"]
  /*[name="Fiction"]/*[name="Classic"]//id
]
```

The final example shows how you can use the `or` operator to specify two dimension value paths. The query returns records tagged with either the Science-fiction or Non-fiction dimension values. Using the `and` operator in place of the `or` operator here would return records tagged with both the Science-fiction and Non-fiction dimension values.

```
collection()/record
[
 Genre = collection("dimensions")/dval[name="Genre"]
  /dval[name="Fiction"]/dval[name="Science-fiction"]//id
 or
 Genre = collection("dimensions")/dval[name="Genre"]
  /dval[name="Non-fiction"]//id
]
```

Querying with dimension value IDs

You can also query dimension value paths using the numerical ID of a dimension value, rather than its name.

In this case, the query returns records that are tagged with this dimension value and all of its descendants (if any). This syntax does not use the "dimensions" argument to the `collection()` function, but it does use the `endeca:dval-by-id()` helper function, as follows:

```
collection()/record[dimName = endeca:dval-by-id(dimValId)//id]
```

where:

- `dimName` is the NCName of the dimension from which the dimension values are filtered. The dimension name is not quoted and is case sensitive.
- `dimValId` is the ID of the dimension value on the records that you want returned. `dimValId` can be any dimension value in the dimension and is not quoted.
- `//id` is the path terminator that specifies that the path be extended to any descendants of `dim¬ ValId`.

You can also use the `and` or `or` operators, as shown in the second example below. You can also use one of the negation operators described in the "Negation operators" topic.

Examples

The first query example selects records that are tagged with either the dimension value whose ID is 9 or its descendants.

```
collection()/record
[
 Genre = endeca:dval-by-id(9)//id
]
```

The next query example uses an `or` operator to select records that are tagged with either dimension value 8 (or its descendants) or dimension value 11 (or its descendants).

```
collection()/record
[
 Genre = endeca:dval-by-id(8)//id
 or
 Genre = endeca:dval-by-id(11)//id
]
```

Record search queries

The `endeca:matches()` function allows a user to perform a keyword search against specific properties or dimension values assigned to records. (Record search queries are also called text search queries.)

The resulting records that have matching properties or dimension values are returned, along with any valid refinement dimension values. The search operation returns results that contain text matching all user search terms (that is, the search is conjunctive by default). To perform a less restrictive search, use the `matchMode` argument to specify a match mode other than `MatchAll` mode. Wildcard terms (using the * character) can be specified if the search interface or property is configured for wildcards in Developer Studio.

Note the following about record search queries:

- The text search is case insensitive, including phrase search.
- Properties must be enabled for record search (in Developer Studio). Records with properties that are not enabled for record search will not be returned in this type of query.
- For wildcard terms, properties must be enabled for wildcard search.

The syntax for a record search query is:

```
collection()/record[endeca:matches(., "searchKey", "searchTerm", "matchMode",
 "languageId", autoPhrase)]
```

The meanings of the arguments are as follows:

Argument	Meaning
.	Required. The period is currently the only valid option. The period is the XPath context item, which is the node currently being considered (that is, the node to apply the function to). In effect, the context item is the record to search.
searchKey	Required. The name of an Endeca property or search interface which will be evaluated for the search. The name must be specified within a set of double quotes. Property names must use the NCName format and must be enabled for record search. Search interface names do not have to use the NCName format.
searchTerm	Required. The term to search for (which may contain multiple words or phrases). Specify the search term within a pair of double quotes. Phrase searches within `searchTerm` must be enclosed within two pairs of double quotes in addition to the pair enclosing the entire `searchTerm` entry. (This

Argument	Meaning
	is because a pair of double quotes is the XPath escape sequence for a single double quote character within a string literal.)
	For example, in "Melville ""Moby Dick"" hardcover", the phrase "Moby Dick" is enclosed in two pairs of double quotes: these yield a single escaped pair which indicates a phrase search for these words. In another example, """Tiny Tim""", the outermost pair of double quotes delimits the full `searchTerm` value, while the two inner pairs yield a single escaped pair to indicate a phrase search.
	Note: To enable EQL parsing, use straight double-quote characters for double quotes (rather than typographer's double quotes, which the EQL parser does not accept).
`matchMode`	Optional. A match mode (also called a search mode) that specifies how restrictive the match should be. The match mode must be specified within a set of double quotes.
	The valid match modes are `all` (MatchAll mode; perform a conjunctive search by matching all user search terms; this is the default), `partial` (MatchPartial mode; match some of the search terms), `any` (MatchAny mode; results need match only a single search term), `allpartial` (MatchAllPartial mode; first use MatchAll mode and, if no results are returned, then use MatchPartial mode), `allany` (MatchAllAny mode; first use MatchAll mode and, if no results are returned, then use MatchAny mode), and `partialmax` (MatchPartialMax mode; first use MatchAll mode and, if no results are returned, then return results matching all but one term, and so on). For details on match modes, see the *Basic Development Guide*.
	Note: MatchBoolean is not supported, because EQL has its own powerful set of query composition features such as the `and`, `or`, and `not` operators.
`languageId`	Optional. A per-query language ID, such as "fr" for French. The ID must be specified within a set of double quotes. For a list of valid language IDs, see the topic "Using language identifiers." The default language ID is the default for the MDEX Engine.
`autoPhrase`	Optional. A TrueFunction or FalseFunction that sets the option for automatic-phrasing query re-write. The default is `false()`, which disables automatic phrasing. Specifying `true()` enables automatic phrasing, which instructs the MDEX Engine to compute a phrasing alternative for a query and then rewrite the query using that phrase. For details on automatic phrasing (including adding phrases to the project with Developer Studio), see the topic "Using automatic phrasing."

Record search query examples

This topic contains examples of record search queries.

The first query example searches for the name *jane* against the `editor_name` property of any record. Because they are not specified, these defaults are used for the other arguments: MatchAll mode, language ID is the MDEX Engine default, and automatic phrasing is disabled.

```
collection()/record
[
 endeca:matches(.,"editor_name","jane")
]
```

The next query example is identical to the first one, except that the wildcard term *ja** is used for the search term. If the `editor_name` property is wildcard-enabled, this search returns records in which the value of the property has a value that begins with *ja* (such as "Jane" or "James").

```
collection()/record
[
 endeca:matches(.,"editor_name","ja*")
]
```

The next query example searches for four individual terms against the "description" property of any records. The `partialmax` argument specifies that the MatchPartialMax match mode be used for the search. The language ID is English (as specified by the "en" argument) and automatic phrasing is disabled (because the default setting is used). Because the MatchPartialMax match mode is specified, MatchAll results are returned if they exist. If no such results exist, then results matching all but one terms are returned; otherwise, results matching all but two terms are returned; and so forth.

```
collection()/record
[
 endeca:matches(.,"description",
  "sailor seafaring ship ocean","partialmax","en")
]
```

The next query example illustrates a phrase search. Any phrase term must be within a pair of double quotes, as in the example ""Tiny Tim"". This is because a pair of double quotes is the XPath escape sequence for a single double-quote character within a string literal. Thus, if the entire search term is a single phrase, there are three sets of quotes, as in the example. For more information on how phrase searches are handled by the MDEX Engine, see the *Basic Development Guide*.

```
collection()/record
[
 endeca:matches(.,"description","""Tiny Tim""")
]
```

In the final query example, the use of the `true()` function enables the automatic phrasing option. This example assumes that phrases have been added to the project with Developer Studio or Endeca Workbench. The example also illustrates the use of the MatchAll match mode.

```
collection()/record
[
 endeca:matches(.,"description","story of","all","en",true())
]
```

Range filter queries

The EQL range filter functionality allows a user, at request time, to specify either a literal value or a geocode value to limit the records returned for the query.

The remaining refinement dimension values for the records in the result set are also returned. The literal value expressions are called basic range queries and the geocode value expressions are geospatial range queries.

Note: Do not confuse EQL range filters with the range filters implemented by the `Nf` parameter. Although both types of range filters are similar in nature, EQL range filters are implemented differently, as described below.

Supported property types for range filters

EQL range filters can be applied only to Endeca properties of certain types.

The following types are supported:

- Integer (for basic range filters)
- Floating point (for basic range filters)
- DateTime (for basic range filters)
- Geocode (for geospatial range filters)

No special configuration is required for these properties. However, the property name must follow the NCName format. No Dgidx flags are necessary to enable range filters, as the range filter computational work is done at request-time. Likewise, no Dgraph flags are needed to enable EQL range filters.

Basic range filter syntax

This topic describes the syntax for EQL range filters.

The syntax for a basic range filter query is:

```
collection()/record[propName rangeOp numLiteral]
```

where:

- `propName` is the name (in an NCName format) of an Endeca property of type Integer or Floating point.
- `rangeOp` is a range (relational) operator from the table below.
- `numLiteral` is a numerical literal value used for the comparison by the range operator.

The property value of `propName` must be numeric in order that a successful comparison be made against the `numLiteral` argument. The supported range operators are the following:

Operator	Meaning
<	Less than. The value of the property is less than the numeric literal.
<=	Less than or equal to. The value of the property is less than the numeric literal or equal to the numerical literal.

Operator	Meaning
>	Greater than or equal to. The value of the property is greater than the numeric literal or equal to the numerical literal.
>=	Greater than. The value of the property is greater than the numeric literal.

Range filter query examples

This topic contains examples of basic range filter queries.

The first query example uses the > operator to return any record that has an `author_id` property whose value is greater than 100.

```
collection()/record
[
 author_id > 100
]
```

The next query example uses the >= operator to return Book records whose `book_id` property value is less than or equal to 99. The example also shows the use of the `and` operator.

```
collection()/record
[
 Recordtype = "book"
 and
 book_id <= 99
]
```

The last query example shows an RRN query that uses a range filter expression in its predicate. Based on a relationship filter applied to the Book and Author records, the query returns Book records (which have the `book_authorref` property) of authors whose books have been edited by an editor whose ID is less than or equal to 12.

```
collection()/record
[
 book_authorref = collection()/record
 [
  author_editorref <= 12
 ]
/author_id
]
```

Geospatial range filter syntax

Geospatial range filter queries will filter records based on the distance of a geocode property from a given reference point.

The reference point is a latitude/longitude pair of floating-point values that are arguments to the `endeca:geocode()` function. These queries are triggered by the `endeca:distance()` function, which in turn uses the `endeca:geocode()` function as one of two arguments in its predicate. The syntax for a geospatial range query is:

```
collection()/record[endeca:distance(geoPropName,
endeca:geocode(latValue,lonValue)) rangeOp distLimit]
```

where:

- `geoPropName` is the name (in NCName format) of an Endeca geocode property.
- `latValue` is the latitude of the location in either an integer or a floating point value. Positive values indicate north latitude and negative values indicate south latitude.
- `lonValue` is the longitude of the location either an integer or a floating point value. Positive values indicate east longitude and negative values indicate west longitude.
- `rangeOp` is either the < (less than) or > (greater than) operator. These range operators specify that the distance from the geocode property to the reference point is either less (<) or greater (>) than the given distance limit (the `distLimit` argument).
- `distLimit` is a numerical literal value used for the comparison by the range operator. Distance limits are always expressed in kilometers.

When the geospatial filter query is made, the records are filtered by the distance from the geocode property to the geocode reference point (the latitude/longitude pair of values).

For example, Endeca's main office is located at 42.365615 north latitude, 71.075647 west longitude. Assuming a geocode property named Location, a geospatial filter query would look like this:

```
collection()/record
[
 endeca:distance(Location,
  endeca:geocode(42.365615,-71.075647)) < 10
]
```

The query returns only those records whose location (as specified in the Location property) is less than 10 kilometers from Endeca's main office.

Dimension search queries

The `Drs` URL query parameter sets an EQL filter for a dimension search.

This filter restricts the scope of the records that will be considered for a dimension search. Only dimension values represented on at least one record satisfying the filter are returned as search results. For details on how dimension search works, see the *Basic Development Guide*.

Note the following about the `Drs` parameter:

- The syntax of `Drs` is identical to that of the `Nrs` parameter.
- `Drs` is dependent on the `D` parameter.

Because the `Drs` syntax is identical to that of `Nrs`, you can use the various EQL requests that are documented earlier in this section.

The following example illustrates a dimension search query using an EQL filter:

```
N=0&D=novel&Drs=collection()/record[author_deceased = "false"]
```

The query uses the `D` parameter to specify *novel* as the search term, while the `Drs` parameter sets a filter for records in which the `author_deceased` property is set to false (that is, records of deceased authors).

Endeca Query Language interaction with other features

Because EQL is a filtering language, it does not contain functionality to perform actions such as triggering Content Spotlighting, sorting, or relevance ranking.

However, EQL is compatible with other query parameters to provide these features for queries. A brief summary of these interactions is:

- `Nrs` is freely composable with the `N`, `Ntt`, `Nr`, and `Nf` filtering parameters. EQL filtering can be conceptualized as occurring after record filtering in terms of side-effects such as spelling auto-correction. This means that a record search within EQL, using the `endeca:matches()` function, cannot auto-correct to a spelling suggestion outside of the record filter.
- Ordering and relevance ranking parameters (`Ns`, `Nrk`, `Nrt`, `Nrr`, `Nrm`) are composable with EQL filters or other types of filters. The `Nrk`, `Nrt`, `Nrr`, and `Nrm` relevance ranking parameters take precedence over a relevance ranking declaration with the `Ntt` and `Ntx` parameters.

The following table provides an overview of these interactions. The sections after the table provide more information.

Parameter	Similar function in EQL?	Why use this parameter rather than Nrs?	Parameter interaction
N	Yes. Dimension filtering can be done in EQL.	Use `N` to trigger Content Spotlighting and refinement generation.	The results of `Nrs` are intersected with the results of `N`.
Nr	Yes. EQL can filter on properties or dimensions.	Use `Nr` for security reasons or to explicitly exclude certain records from being considered in the rest of the query (e.g., for spelling suggestions).	`Nr` is a pre-filter. Only the records that pass this filter are even considered in `Nrs`.
Ntt, Ntk	Yes. EQL provides the ability to do record search.	Use `Ntt`/`Ntk` to trigger Content Spotlighting, as record search within `Nrs` does not trigger it. Use `Ntt`/`Ntk` with `Nty` for DYM spelling suggestions. (`Nrs` record search does support autocorrection, but not DYM.)	Similar to `N` , the results of `Nrs` are intersected with the results of `Ntt`/`Ntk`.
Nf	Yes. EQL provides the ability to do range filtering.	No reason to do so. EQL actually provides greater flexibility because range filters within `Nrs` can be OR'ed, whereas `Nf`	Similar to `N`, the results of `Nrs` are intersected with the results of `Nf`.

Parameter	Similar function in EQL?	Why use this parameter rather than Nrs?	Parameter interaction
		range filters cannot. Similar to `N`, the results of `Nrs` are intersected with the results of `Nf`.	
`Ns`	No. EQL does not have the ability to sort results.	N/A	As long as the property specified in `Ns` exists on the records being returned, the `Ns` parameter will sort the results.
`Ne`	No. EQL does not have the ability to expose dimensions.	N/A	As long as the dimensions specified in `Ne` exist on the records being returned, the `Ne` parameter will expose those dimensions.
`Nrk, Nrt, Nrr, Nrm`	No. EQL does not provide the ability to relevance rank the results.	N/A	This set of parameters allow the ability to apply relevance ranking to results even if record search does not exist.
`Nrc`	No. EQL does not provide the ability to modify refinement configuration.	N/A	The `Nrc` parameter lets you modify refinement configuration at query time (for dynamic ranking, statistics, and so on).

N parameter interaction

The `Nrs` parameter has a dependency on the `N` parameter.

This means that you must use `N=0` if no navigation filter is intended. Note, however, that the presence of the `N` parameter does not affect `Nrs` (for example, for such actions as spelling correction and automatic phrasing).

If the `N` parameter is used with one or more dimension value IDs, it can trigger Content Spotlighting, since dimension filtering within `Nrs` does not trigger it. The resulting record set will be an intersection of both filters. In this case, the dimension value IDs specified by the `N` parameter must belong to dimensions that exist for the records being returned by `Nrs`.

For example, if the `N` parameter is filtering on Author Location but the `Nrs` parameter is returning only Book records, then this intersection will result in zero records. In addition, during a query the `Nrs` parameter does not trigger refinement generation for multi-select and hierarchical dimensions, while `N` does. Therefore, because `Nrs` is ignored for purposes of refinement generation while `N` plays a key role, the `N` parameter should be used instead of `Nrs` for parts of the query.

Nr record filter interactions

The `Nr` parameter sets a record filter for a navigation query.

When used with an EQL request, the `Nr` parameter acts as a prefilter. That is, it restricts the set of records that are visible to the `Nrs` parameter. Because it is a prefilter, the `Nr` parameter is especially useful as a security filter to control the records that a user can see. It is also useful to explicitly exclude certain records from being considered in the rest of the query (for example, for spelling suggestions).

When using the `Nr` parameter, keep in mind that only the records that pass the `Nr` filter are even considered in `Nrs`. For example, if you have Book records and Author records, both of these record types would have to pass the `Nr` record filter logic in order for the `Nrs` parameter to determine relationships between Books and Authors.

Nf range filter interactions

The `Nf` parameter enables range filter functionality.

Unlike a record filter, the `Nf` parameter does not act as a prefilter. Instead, when used with the `Nrs` parameter, the resulting record set will be an intersection of the results of the `Nf` and `Nrs` parameters. That is, an `Nf` range filter and an EQL filter together form an AND Boolean request. For more information on range filters, see "Using Range Filters" in the *MDEX Engine Basic Development Guide*.

Ntk and Ntt record search interaction

`Ntk` and `Ntt` are a set of parameters used for record search which act as a filter (not a prefilter).

Therefore, when used with the `Nrs` parameter, the resulting record set will be an intersection of the results of the `Nrs` parameter and the `Ntk/Ntt` parameters. There are two main advantages of using these parameters with the `Nrs` parameter:

- The `Ntk/Ntt` parameters can trigger Content Spotlighting, whereas the `Nrs` parameter cannot.
- The `Ntk/Ntt` parameters can return auxiliary information (such as DYM spelling suggestions and supplemental objects), whereas the `Nrs` parameter cannot.

In addition, you can use other parameters that depend on `Ntk`, such as the `Ntx` parameter to specify a match mode or a relevance ranking strategy. For details on the `Ntk`, `Ntt`, and `Ntx` parameters, see the *Basic Development Guide*.

Ns sorting interaction

You can append the `Ns` parameter to an EQL request to sort the returned record set by a property of the records.

To do so, use the following syntax:

```
Ns=sort-key-name[|order]
```

The `Ns` parameter specifies the property or dimension by which to sort the records, and an optional list of directions in which to sort. For example, this query:

```
Nrs=collection()/records[book_authorref = collection()
 /records[author_nationality = "american"]
 /author_id]&Ns=book_year
```

returns all books written by American authors and sorts the records by the year in which the book was written (the `book_year` property). You can also add the optional order parameter to `Ns` to control the order in which the property is sorted (0 is for an ascending sort while 1 is descending). The default sort order for a property is ascending. For example, the above query returns the records in ascending order (from the earliest year to the latest), while the following `Ns` syntax uses a descending sort order:

```
Ns=book_year|1
```

For more details on how to specify a sort order, see "Sorting Endeca Records" in the *Basic Development Guide*.

Nrk relevance ranking interaction

The `Nrk`, `Nrt`, `Nrr`, and `Nrm` set of parameters can be used to order the records of an EQL request, via a specified relevance ranking strategy.

The following is an example of using these parameters:

```
Nrs=collection()/record[Recordtype = "book"]
 &Nrk=All&Nrt=novel&Nrr=maxfield&Nrm=matchall
```

The sample query returns all Book records (that is, all records that are tagged with the Recordtype property set to "book"). The record set is ordered with the Maxfield relevance ranking module (specified via `Nrr`), which uses the word *novel* (specified via `Nrt`). The search interface is specified via the `Nrk` parameter.

The `Nrk`, `Nrt`, `Nrr`, and `Nrm` parameters take precedence over the `Ntk`, `Ntt`, and `Ntx` parameters. That is, if both sets of parameters are used in a query, the relevance ranking strategy specified by the `Nrr` parameter will be used to order the records. For more information on these parameters, see the topic "Using the Nrk, Nrt, Nrr, and Nrm parameters."

Ne exposed refinements interaction

The `Ne` parameter specifies which dimension (out of all valid dimensions returned in an EQL request) should return actual refinement dimension values.

The behavior of the `Ne` parameter is the same for EQL request as for other types of navigation queries.

The following example shows the `Ne` parameter being specified with an EQL text search:

```
Nrs=collection()/record[endeca:matches(.,"description",
 "story","partialmax")]&Ne=6
```

In the query, 6 is the root dimension value ID for the Genre dimension. The query will return all dimensions for records in which the term story appears in the description property, as well as the refinement dimension values for the Genre dimension.

Spelling auto-correction and Did You Mean interaction

Spelling auto-correction for dimension search and record search automatically computes alternate spellings for user query terms that are misspelled.

The Did You Mean (DYM) feature provides the user with explicit alternative suggestions for a keyword search. Both features are fully explained in the "Implementing Spelling Correction and Did You Mean" section.

Both DYM and spelling auto-correction work normally when the `Ntt` parameter is used with `Nrs`. For example, in the following query:

```
Nrs=collection()/record[Recordtype = "book"]
 &Ntk=description&Ntt=storye&Ntx=mode+matchall
```

the misspelled term *storye* is auto-corrected to *story* (assuming that the MDEX Engine was started with the `--spl` flag).

If DYM is enabled instead of auto-correction (using the `--dym` flag), then the `Nty=1` parameter can be used in the query:

```
Nrs=collection()/record[Recordtype = "book"]
 &Ntk=description&Ntt=storye&Ntx=mode+matchall&Nty=1
```

In this case, no records are returned (assuming that the misspelled term *storye* is not in the data set), but the term *story* is returned as a DYM suggestion.

If both spelling auto-correction and DYM are enabled, the spelling auto-correction feature will take precedence. However, for a full text search with the `endeca:matches()` function, the spelling auto-correction feature will work, but the DYM feature is not supported. For example, in this query:

```
collection()/record
[
 endeca:matches(.,"description","storye")
]
```

the misspelled term *storye* is auto-corrected to *story* if auto-correction is enabled. If DYM is enabled but auto-correction is not, then no records are returned (again assuming that the misspelled term *storye* is not in the data set).

Endeca Analytics interaction

The Endeca Analytics API can be used to request analytics operations based on the results of a navigation query.

The analytics operations will work the same way as with navigation queries that do not use EQL.

Endeca Query Language per-query statistics log

The MDEX Engine can log information about the processing time of an EQL request.

The log entry is at the level of a time breakdown across the stages of query processing (including relationship filters). This information will help you to identify and tune time-consuming queries.

Note: Only EQL requests produce statistics for this log. Therefore, you should enable this log only if you are using EQL.

Implementing the per-query statistics log

The EQL per-query statistics log is turned off by default.

You can specify its creation by using the Dgraph `--log_stats` flag:

```
--log_stats path
```

The path argument sets the path and filename for the log.

This argument must be a filename, not a directory. If the file cannot be opened, no logging will be performed. The log file uses an XML format, as shown in the following example that shows a log entry for this simple query:

```
fn:collection()/record[author_nationality = "english"]
```

To read the file, you can open it with a text editor, such as TextPad.

```
<?xml version="1.0" encoding="UTF-8"?>
<Queries>
<Query xmlns="endeca:stats">

  <EndecaQueryLanguage>
    <Stats>
      <RecordPath query_string="fn:collection()/record[author_nationality
= &quot;english&quot;]">
        <StatInfo number_of_records="2">
          <TimeInfo>
            <Descendant unit="ms">0.47705078125</Descendant>
            <Self unit="ms">0.194580078125</Self>
            <Total unit="ms">0.671630859375</Total>
          </TimeInfo>
        </StatInfo>
       <Predicate query_string="[author_nationality = &quot;english&quot;]">

          <StatInfo number_of_records="2">
            <TimeInfo>
              <Descendant unit="ms">0.287841796875</Descendant>
              <Self unit="ms">0.189208984375</Self>
              <Total unit="ms">0.47705078125</Total>
            </TimeInfo>
          </StatInfo>
          <PropertyComparison query_string="author_nationality = &quot;en¬
glish&quot;">
            <StatInfo number_of_records="2">
              <TimeInfo>
                <Descendant unit="ms">0.001953125</Descendant>
                <Self unit="ms">0.285888671875</Self>
                <Total unit="ms">0.287841796875</Total>
              </TimeInfo>
            </StatInfo>
            <StringLiteral query_string="&quot;english&quot;">
              <StatInfo number_of_records="0">
                <TimeInfo>
                  <Descendant unit="ms">0</Descendant>
                  <Self unit="ms">0.001953125</Self>
                  <Total unit="ms">0.001953125</Total>
                </TimeInfo>
              </StatInfo>
            </StringLiteral>
          </PropertyComparison>
        </Predicate>
      </RecordPath>
```

```
      </Stats>
  </EndecaQueryLanguage>

</Query>
</Queries>
```

Parts of the log file

The following table describes the meanings of the elements and attributes.

Element/Attribute	Description
Query	Encapsulates the statistics for a given query (that is, each query will have its own Query node).
RecordPath	The record path of a collection() function.
Predicate	Lists the time spent processing the predicate part of a query.
otherNodes	Lists the time spent processing an expression part of an query, such as PropertyComparison (property or range filter query), StringLiteral (property value query), MatchesExpr (text search query), and DValComparison (dimension value query).
query_string	For RecordPath, this attribute lists the full query that was issued. For the other elements, it lists the part of the query for which statistics in that element are given.
number_of_records	Returns the number of records which satisfy the query_string in a given node.
StatInfo	Encapsulates the TimeInfo and CacheInfo information.
TimeInfo	Encapsulates time-related information about the node.
Descendant	The time, in milliseconds, spent in the descendants of a given node.
Self	The total amount of time, in milliseconds, spent in this node.
Total	The total amount of time, in milliseconds, spent in this node and its descendants.
CacheInfo	Encapsulates information about cache hits, misses, and insertions. Cache is checked only when a combined relationship filter and range comparison is made.

Setting the logging threshold for queries

You can set the threshold above which statistics information for a query will be logged.

You do this by using the Dgraph `--log-stats-thresh` flag. Note that this flag is dependent on the `--log_stats` flag.

The syntax of the threshold flag is:

```
--log_stats_thresh value
```

The value argument is in milliseconds (1000 milliseconds = 1 second). However, the value can be specified in seconds by adding a trailing s to the number.

For example, this:

```
--log_stats_thresh 1s
```

is the same as:

```
--log_stat_thresh 1000
```

If the total execution time for an Endeca Query Language request (not the expression execution time) is above this threshold, per-query performance information will be logged. The default for the threshold is 1 minute (60000 milliseconds). That is, if you use the `--log_stats` flag but not the `--log_stats_thresh` flag, a value of 1 minute will be used as the threshold for the queries.

Creating an Endeca Query Language pipeline

This section provides information on configuring the pipeline for an application that implements EQL.

Also included are requirements for the Endeca properties and dimensions.

Creating the dimensions and properties

Before an Endeca property can be used in EQL requests, the property must be configured appropriately.

The details are as follows:

- One or more of the following must be true of the property:
  - It is explicitly enabled for use with record filters.
  - It is specified as a rollup key.
  - It is specified as a record spec.
  - It has one of the following types: double, integer, geocode, datetime, duration, or time.
- The property name must be in the NCName format, as explained in the topic "NCName format for properties and dimensions."
- If you want to allow wildcard terms for record searches, the property must be enabled for wildcard search.

To enable a property for record filters, open the property in the Developer Studio Property editor and check the "Enable for record filters" attribute.

Use the Property editor's Search tab to configure the property for record search and wildcard search. To use dimensions in Endeca Query Language queries:

- All dimensions are automatically enabled for use in EQL record filter expressions, and therefore do not need to be enabled for record filters.
- Dimension names (and therefore the names of root dimension values) must be in the NCName format. Names of non-root dimension values, however, do not have to be in the NCName format.

If you are using a search interface with EQL requests, the name of the search interface does not have be an NCName.

Configuring the pipeline for Switch joins

With one exception, the pipeline used for an application that implements EQL does not have any special configuration requirements.

The exception is if you purchased the RRN module and will be using it to apply relationship filters at query time. In this case, you will probably be using a Switch join in the pipeline. Note that using a Switch join is not mandatory for RRN queries, but you will use one if you want to keep different record types uncombined.

For example, the pipeline used in the application that provides the sample queries (for other sections of this chapter) assumes that the data set has three types of records. The pipeline looks like this in Developer Studio's Pipeline Diagram:
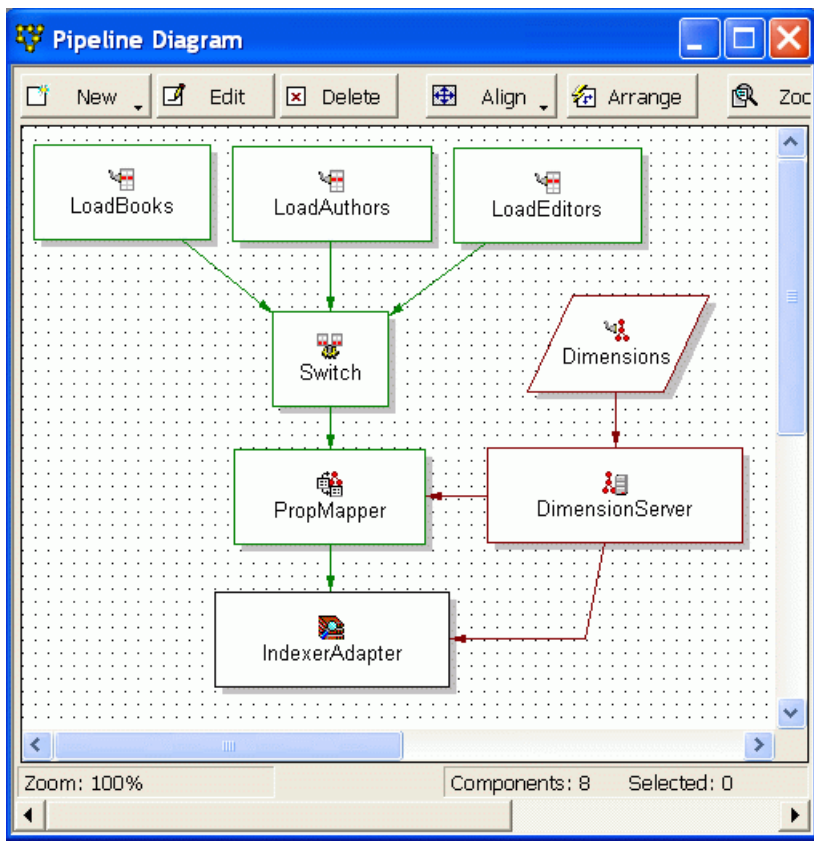
The pipeline has three record adapters to load the three record types (Book records, Author records, and Editor records). These are standard record adapters and do not require any special configuration.

The record assembler (named Switch) is used to implement a Switch join on the three sets of records. The Sources tab is where you add the record sources for the record assembler, which are the three record adapters:
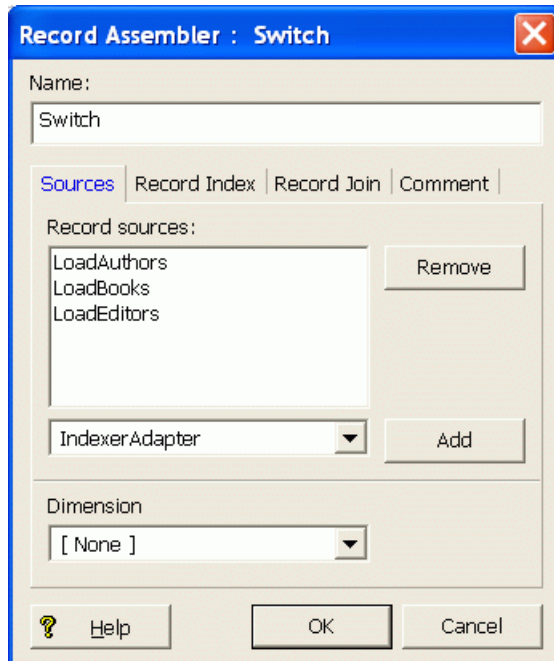
The record assembler will process all the records from the three record adapters. However, the records are never compared or combined. Because the three record types are not combined, you can use RRN queries to apply relationship filters. For more information on these types of queries, see the topic "Record Relationship Navigation queries."

Running the Endeca Query Language pipeline

No special configuration is needed for running an EQL pipeline.

You can run the pipeline with either the Endeca Application Controller (EAC) or control scripts. See the *Platform Services Application Controller Guide* for details on provisioning your application. For information on using control scripts, see the *Platform Services Control System Guide*.

Chapter 3

Record Filters

This section describes how to implement record filters in your Endeca application.

About record filters

Record filters allow an Endeca application to define arbitrary subsets of the total record set and dynamically restrict search and navigation results to these subsets.

For example, the catalog might be filtered to a subset of records appropriate to the specific end user or user role. The records might be restricted to contain only those visible to the current user based on security policies. Or, an application might allow end users to define their own custom record lists (that is, the set of parts related to a specific project) and then restrict search and navigation based on a selected list. Record filters enable these and many other application features that depend on applying Endeca search and navigation to dynamically defined and selected subsets of the data.

If you specify a record filter, whether for security, custom catalogs, or any other reason, it is applied before any search processing. The result is that the search query is performed as if the data set only contained records allowed by the record filter.

Record filters support Boolean syntax using property values and dimension values as base predicates and standard Boolean operators (AND, OR, and NOT) to compose complex expressions. For example, a filter can consist of a list of part number property values joined in a multi-way OR expression. Or, a filter might consist of a complex nested expression of ANDs, ORs, and NOTs on dimension IDs and property values.

Filter expressions can be saved and loaded from XML files, or passed directly as part of an MDEX Engine query. In either case, when a filter is selected, the set of visible records is restricted to those matching the filter expression. For example, record search queries will not return records outside the selected subset, and refinement dimension values are restricted to lead only to records contained within the subset.

Finally, it is important to keep in mind that record filters are case-sensitive.

Record filter syntax

Record filters are specified with query-based or file-based expressions.

Record filters can be specified directly within an MDEX Engine query. For example, the complete Boolean expression representing the desired record subset can be passed directly in an application URL.

In some cases, however, filter expressions require persistence (in the case where the application allows the end user to define and save custom part lists) or may grow too large to be passed conveniently as part of the query (in the case where a filter list containing thousands of part numbers). To handle cases such as these, the MDEX Engine also supports file-based filter expressions.

File-based filter expressions are simply files stored in a defined location containing XML representations of filter expressions. This section describes both the MDEX Engine query and XML syntaxes for filter expressions.

Query-level syntax

The query-level syntax supports prefix-oriented Boolean functions (AND, OR, and NOT), colon-separated paths for dimension values and property values, and numeric dimension value IDs.

The following BNF grammar describes the syntax for query-level filter expressions:

```
<filter>       ::= <and-expr>
                  | <or-expr>
                  | <not-expr>
                  | <filter-expr>
                  | <literal>
<and-expr>     ::= AND(<filter-list>)
<or-expr>      ::= OR(<filter-list>)
<not-expr>     ::= NOT(<filter>)
<filter-expr>  ::= FILTER(<string>)
<filter-list>  ::= <filter>
                  | <filter>,<filter-list>
<literal>      ::= <pval>
                  | <dval-id>
                  | <dval-path>
<pval>         ::= <prop-key>:<prop-value>
<prop-key>     ::= <string>
<prop-value>   ::= <string>
<dval-path>    ::= <string>
                  | <string>:<dval-path>
<dval-id>      ::= <unsigned-int>
<string>       ::= any character string
```

The following five special reserved characters must be prepended with an escape character (\) for inclusion in a string:

```
( ) , : \
```

Using the FILTER operator

Aside from nested Boolean operations, a key aspect of query filter expressions is the ability to refer to file-based filter expressions using the FILTER operator. For example, if a filter is stored in a file called MyFilter, that filter can be selected as follows:

```
FILTER(MyFilter)
```

FILTER operators can be combined with normal Boolean operators to compose filter operations, as in this example:

```
AND(FILTER(MyFilter),NOT(Manufacturer:Sony))
```

The expression selects records that are satisfied by the expression contained in the file MyFilter but that are not assigned the value Sony to the Manufacturer property.

Example of a query-level filter expression

The following example illustrates a basic filter expression that uses nested Boolean operations:

```
OR(AND(Manufacturer:Sony,1001),
    AND(Manufacturer:Aiwa,NOT(1002)), Manufacturer:Denon)
```

This expression will match the set of records satisfying any of the following statements:

- Value for the Manufacturer property is `Sony` and record assigned dimension value is `1001`.
- Value for Manufacturer is `Aiwa` and record is not assigned dimension value `1002`.
- Value for Manufacturer property is `Denon`.

XML syntax for file-based record filter expressions

The syntax for file-based record filter expressions closely mirrors the query level syntax, with some differences.

The file-based differences from the query-level syntax are:

- In place of the AND, OR, NOT, and FILTER operators, the FILTER_AND, FILTER_OR, FILTER_NOT, and FILTER_NAME XML elements are used, respectively.
- In place of the property and dimension value syntax used for query expressions, the PROP, DVAL_ID, and DVAL_PATH elements are used. Note that the DVAL_PATH element's PATH attribute requires that paths for dimension values and property values be separated by colons, not forward slashes.
- Instead of parentheses to enclose operand lists, normal XML element nesting (implicit in the locations of element start and end tags) is used.

The full DTD for XML file-based record filter expressions is provided in the `filter.dtd` file packaged with the Endeca software release.

Examples of file-based filter expressions

As an example, the following query-level expression:

```
OR(AND(Manufacturer:Sony,1001),
    AND(Manufacturer:Aiwa,NOT(1002)), Manufacturer:Denon)
```

is represented as a file-based expression using the following XML syntax:

```
<FILTER>
  <FILTER_OR>
    <FILTER_AND>
      <PROP NAME="Manufacturer"><PVAL>Sony</PVAL></PROP>
      <DVAL_ID ID="1001"/>
    </FILTER_AND>
    <FILTER_AND>
      <PROP NAME="Manufacturer"><PVAL>Aiwa</PVAL></PROP>
      <FILTER_NOT>
        <DVAL_ID ID="1002"/>
      </FILTER_NOT>
    </FILTER_AND>
    <PROP NAME="Manufacturer"><PVAL>Denon</PVAL></PROP>
  </FILTER_OR>
</FILTER>
```

Just as file-based expressions can be composed with query expressions, file expressions can also be composed within other file expressions. For example, the following query expression:

```
AND(FILTER(MyFilter),NOT(Manufacturer:Sony))
```

can be represented as a file-based expression using the following XML:

```
<FILTER>
   <FILTER_AND>
      <FILTER_NAME NAME="MyFilter"/>
      <FILTER_NOT>
          <PROP NAME="Manufacturer"><PVAL>Sony</PVAL></PROP>
      </FILTER_NOT>
   </FILTER_AND>
</FILTER>
```

Enabling properties for use in record filters

Endeca Properties must be explicitly enabled for use in record filters.

Note that all dimension values are automatically enabled for use in record filter expressions.

To enable a property for use with record filters:

1. In Developer Studio, open the Properties view.
2. Double-click on the Endeca property that you want to configure.
   The property is opened in the Property editor.
3. Check the **Enable for record filters** option, as in the following example.
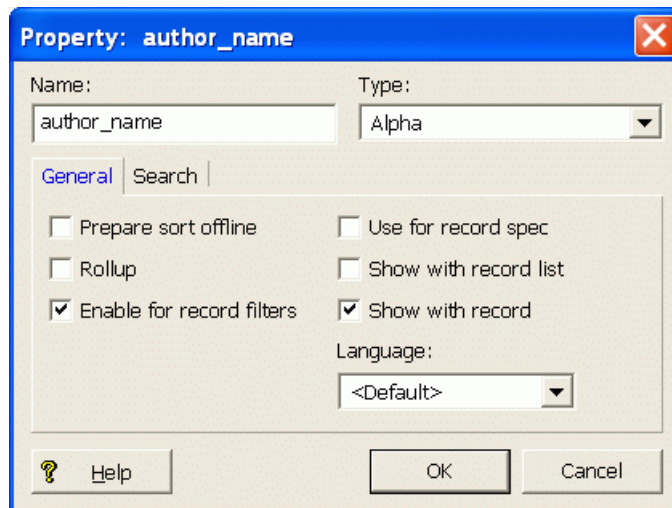
4. Click **OK** to save your changes.

Data configuration for file-based filters

To use file-based filter expressions in an application, you must create a directory to contain record filter files in the same location where the MDEX Engine index data will reside.

The name of this directory must be:

```
<index_prefix>.fcl
```

For example, if the MDEX Engine index data resides in the directory:

```
/usr/local/endeca/my_app/data/partition0/dgidx_output/
```

and the index data prefix is:

```
/usr/local/endeca/my_app/data/partition0/dgidx_output/index
```

then the directory created to contain record filter files must be:

```
/usr/local/endeca/my_app/data/partition0/dgidx_output/index.fcl
```

Record filters that are needed by the application should be stored in this directory, which is searched automatically when record filters are selected in an MDEX Engine query. For example, if in the above case you create a filter file with the path:

```
/usr/local/endeca/my_app/data/partition0/dgidx_output/index.fcl/MyFilter
```

then the filter expression stored in this file will be used when the query refers to the filter `MyFilter`.

For example, the URL query:

```
N=0&Nr=FILTER(MyFilter)
```

will use this file filter.

Record filter result caching

The MDEX Engine caches the results of file-based record filter evaluations for re-use.

The cached results are used on subsequent MDEX Engine queries as part of the global dynamic cache. The cache replacement policy is to discard least recently-used (LRU) entries.

Note: The MDEX Engine only caches the results of file-based record filters, because these are generally more costly to evaluate due to XML-parsing overhead.

URL query parameters for record filters

Three MDEX Engine URL query parameters are available to control the use of record filters.

The URL query parameters are as follows:

Parameter	Description
Nr	Links to the Java `ENEQuery.setNavRecordFilter()` method and the .NET `ENEQuery.NavRecordFilter` property. The `Nr` parameter can be used to specify a record filter expression that will restrict the results of a navigation query.
Ar	Links to the Java `ENEQuery.setAggrERecNavRecordFilter()` method and the .NET `ENEQuery.AggrERecNavRecordFilter` property. The `Ar` parameter can be used to specify a record filter expression that will restrict the records contained in an aggregated-record result returned by the MDEX Engine.

Parameter	Description
Dr	Links to the Java `ENEQuery.setDimSearchNavRecordFilter()` method and the .NET `ENEQuery.DimSearchNavRecordFilter` property. The `Dr` parameter can be used to specify a record filter expression that will restrict the universe of records considered for a dimension search. Only dimension values represented on at least one record satisfying the specified filter will be returned as search results.

Using the Nr query parameter

You can use the `Nr` parameter to perform a record query search so that only results tagged with a specified dimension value are returned. For example, say you have a dimension tree that looks like this, where Sku is the dimension root and 123, 456, and 789 are leaf dimension values:

```
Sku
  123
  456
  789
  ...
```

To perform a record query search so that results tagged with any of these dimension values is returned, use the following:

```
Nr=OR(sku:123,OR(sku:456),OR(sku:789))
```

To perform a record query search so that only results tagged with the dimension value 123 are returned, use the following:

```
Nr=sku:123
```

Examples of record filter query parameters

```
<application>?N=0&Nr=FILTER(MyFilter)

<application>?A=2496&An=0&Ar=OR(10001,20099)

<application>?D=Hawaii&Dn=0&Dr=NOT(Subject:Travel)
```

Record filter performance impact

Record filters can have an impact in some areas.

The evaluation of record filter expressions is based on the same indexing technology that supports navigation queries in the MDEX Engine. Because of this, there is no additional memory or indexing cost associated with using navigation dimension values in record filters.

Because expression evaluation is based on composition of indexed information, most expressions of moderate size (that is, tens of terms/operators) do not add significantly to request processing time.

Furthermore, because the MDEX Engine caches the results of file-based record filter operations on an LRU (least recently used) basis, the costs of expression evaluation are typically only incurred on the first use of a file-based filter during a navigation session. However, some expected uses of record filters have known performance bounds, which are described below.

Record filters can impact the following areas:

- Spelling auto-correction and spelling Did You Mean
- Memory cost

- Expression evaluation

Interaction with spelling auto-correction and spelling DYM

Record filters impose an extra cost on spelling auto-correction and spelling Did You Mean.

Memory cost

The use of properties in record filters incurs a memory cost.

The evaluation of record filter dimension value expressions is based on the same indexing technology that supports navigation queries in the Dgraph. Because of this, there is no additional memory or indexing cost associated with using navigation dimension values in record filters. When using property values in record filter expressions, additional memory and indexing cost is incurred because properties are not normally indexed for navigation.

This feature is controlled in Developer Studio by the **Enable for record filters** setting on the Property editor.

Expression evaluation

Expression evaluation of large OR filters and large scale negation can impose a performance impact on the system.

Because expression evaluation is based on composition of indexed information, most expressions of moderate size (that is, tens of terms and operators) do not add significantly to request processing time. Furthermore, because the Dgraph caches the results of record filter operations, the costs of expression evaluation are typically only incurred on the first use of a filter during a navigation session. However, some expected uses of record filters have known performance bounds, which are described in the following two sections.

Large OR filters

One common use of record filters is the specification of lists of individual records to identify data subsets (for example, custom part lists for individual customers, culled from a superset of parts for all customers).

The total cost of processing records can be broken down into two main parts: the parsing cost and the evaluation cost. For large expressions such as these, which will commonly be stored as file-based filters, XML parsing performance dominates total processing cost.

XML parsing cost is linear in the size of the filter expression, but incurs a much higher unit cost than actual expression evaluation. Though lightweight, expression evaluation exhibits non-linear slowdown as the size of the expression grows.

OR expressions with a small number of operands perform linearly in the number of results, even for large result sets. While the expression evaluation cost is reasonable into the low millions of records for large OR expressions, parsing costs relative to total query execution time can become too large, even for smaller numbers of records.

Part lists beyond approximately one hundred thousand records generally result in unacceptable performance (10 seconds or more load time, depending on hardware platform). Lists with over one million records can take a minute or more to load, depending on hardware. Because results are cached, load time is generally only an issue on the first use of a filter during a session. However, long load times can cause other Dgraph requests to be delayed and should generally be avoided.

Large-scale negation

In most common cases, where the NOT operator is used in conjunction with other positive expressions (that is, AND with a positive property value), the cost of negation does not add significantly to the cost of expression evaluation.

However, the costs associated with less typical, large-scale negation operations can be significant. For example, while still sub-second, top-level negation filtering (such as "NOT availability=FALSE") of a record set in the millions does not allow high throughput (generally less than 10 operations per second).

If possible, attempt to rephrase expressions to avoid the top-level use of NOT in Boolean expressions. For example, in the case where you want to list only available products, the expression "availability=TRUE" will yield better performance than "NOT availability=FALSE".

Chapter 4

Bulk Export of Records

This section describes the bulk export feature.

About the bulk export feature

The bulk export feature allows your application to perform a navigation query for a large number of records.

Each record in the resulting record set is returned from the MDEX Engine in a bulk-export-ready, gzipped format. The records can then be exported to external tools, such as a Microsoft Excel or a CSV (comma separated value) file.

Applications are typically limited in the number of records that can be requested by the memory requirements of the front-end application server. The bulk export feature adds a means of delaying parsing and `ERec` or `AggrERec` object instantiation, which allows front-end applications to handle requests for large numbers of records.

Configuring the bulk export feature

Endeca properties and dimensions must be configured for bulk export.

Endeca properties and/or dimensions that will be included in a result set for bulk exporting must be configured in Developer Studio with the **Show with Record List** checkbox enabled. When this checkbox is set, the property or dimension will appear in the record list display.

No Dgidx or Dgraph flags are necessary to enable the bulk exporting of records. Any property or dimension that has the **Show with Record List** attribute is available to be exported.

Using URL query parameters for bulk export

A query for bulk export records is the same as any valid navigation query.

Therefore, the Navigation parameter (`N`) is required for the request. No other URL query parameters are mandatory.

Setting the number of bulk records to return

By using members from the ENEQuery class, you can set the number of bulk-format records to be returned by the MDEX Engine.

When creating the navigation query, the application can specify the number of Endeca records or aggregated records that should be returned in a bulk format with these Java and .NET calls:

- The Java `ENEQuery.setNavNumBulkERecs()` method and the .NET `ENEQuery.NavNumBulkERecs` property set the maximum number of Endeca records (`ERec` objects) to be returned in a bulk format from a navigation query.
- The Java `ENEQuery.setNavNumBulkAggrERecs()` method and the .NET `ENEQuery.NavNumBulkAggrERecs` property set the maximum number of aggregated Endeca records (`AggrERec` objects) to be returned in bulk format from a navigation query.

The MAX_BULK_ERECS_AVAILABLE constant can be used with either call to specify that all of the records that match the query should be exported; for example:

```
// Java example:
usq.setNavNumBulkERecs(MAX_BULK_ERECS_AVAILABLE);

// .NET example:
usq.NavNumBulkERecs = MAX_BULK_ERECS_AVAILABLE;
```

To find out how many records will be returned for a bulk-record navigation query, use these calls:

- The Java `ENEQuery.getNavNumBulkERecs()` method and the .NET `ENEQuery.NavNumBulkERecs` property are for Endeca records.
- The Java `ENEQuery.getNavNumBulkAggrERecs()` method and the .NET `ENEQuery.NavNumBulkAggrERecs` property are for aggregated Endeca records.

Note that all of the above calls are also available in the `UrlENEQuery` class.

The following examples set the maximum number of bulk-format records to 5,000 for a navigation query.

Java example

```
// Set MDEX Engine connection
ENEConnection nec = new HttpENEConnection(eneHost,enePort);
// Create a query
ENEQuery usq = new UrlENEQuery(request.getQueryString(), "UTF-8");
// Specify the maximum number of records to be returned
usq.setNavNumBulkERecs(5000);
// Make the query to the MDEX Engine
ENEQueryResults qr = nec.query(usq);
```

.NET example

```
// Set Navigation Engine connection
HttpENEConnection nec = new HttpENEConnection(ENEHost, ENEPort);
// Create a query
String queryString = Request.Url.Query.Substring(1);
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");
// Specify the maximum number of records to be returned
usq.NavNumBulkERecs = 5000;
// Make the request to the Navigation Engine
ENEQueryResults qr = nec.Query(usq);
```

Retrieving the bulk-format records

By using members from the Navigation class, you can retrieve the returned set of bulk-format records from the Navigation query object.

The list of Endeca records is returned from the MDEX Engine inside the standard `Navigation` object. The records are returned compressed in a gzipped format. The format is not directly exposed to the application developer; the developer only has access to the bulk data through the methods from the language being used. Note that the retrieval method depends on whether you have a Java or .NET implementation.

It is up to the front-end application developer to determine what to do with the retrieved records. For example, you can display each record's property and/or dimension values, as described in this guide. You can also write code to properly format the property and dimension values for export to an external file, such as a Microsoft Excel file or a CSV file.

Using Java Bulk Export methods

In a Java-based implementation, the list of Endeca records is returned as a standard Java Iterator object.

To access the bulk-record `Iterator` object, use one of these methods:

- `Navigation.getBulkERecIter()` returns an Iterator object containing the list of Endeca bulk-format records (`ERec` objects).
- `Navigation.getBulkAggrERecIter()` returns an Iterator object containing the list of aggregated Endeca bulk-format records (`AggrERec` objects).

The `Iterator` class provides access to the bulk-exported records. The `Iterator.next()` method will gunzip the next result record and materialize the per-record object. The methods in the Iterator class that allow access to the exported records are the following:

- `Iterator.hasNext()` returns true if the iterator has more records.
- `Iterator.next()` returns the next record in the iteration. The record is returned as either an `ERec` or `AggrERec` object, depending on which `Navigation` method was used to retrieve the iterator.

The following Java code fragment shows how to set the maximum number of bulk-format records to 5,000 and then obtain a record list and iterate through the list.

```
// Create a query
ENEQuery usq = new UrlENEQuery(request.getQueryString(), "UTF-8");
// Specify the maximum number of bulk export records
// to be returned
usq.setNavNumBulkERecs(5000);
// Make the query to the MDEX Engine
ENEQueryResults qr = nec.query(usq);
// Verify we have a Navigation object before doing anything.
if (qr.containsNavigation()) {
  // Get the Navigation object
  Navigation nav = ENEQueryResults.getNavigation();
  // Get the Iterator object that has the ERecs
  Iterator bulkRecs = nav.getBulkERecIter();
  // Loop through the record list
  while (bulkRecs.hasNext()) {
    // Get a record, which will be gunzipped
    ERec record = (ERec)bulkRecs.next();
```

```
        // Display its properties or format the record for export
        ...
    }
}
```

Using .NET bulk export methods

In a .NET application, the list of Endeca records is returned as an Endeca ERecEnumerator object.

To retrieve the `ERecEnumerator` object, use the `Navigation.BulkAggrERecEnumerator` or `Navigation.BulkERecEnumerator` property.

The following .NET code sample shows how to set the maximum number of bulk-format records to 5000, obtain the record list, and iterate through the collection. After the `ERecEnumerator` object is created, an enumerator is positioned before the first element of the collection, and the first call to `MoveNext()` moves the enumerator over the first element of the collection. After the end of the collection is passed, subsequent calls to `MoveNext()` return false. The Current property will gunzip the current result record in the collection and materialize the per-record object.

```
// Create a query
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");
// Set max number of returned bulk-format records
usq.NavNumBulkERecs = 5000;
// Make the query to the Navigation Engine
ENEQueryResults qr = nec.Query(usq);
// First verify we have a Navigation object.
if (qr.ContainsNavigation()) {
  // Get the Navigation object
  Navigation nav = ENEQueryResults.Navigation;
  // Get the ERecEnumerator object that has the ERecs
  ERecEnumerator bulkRecs = nav.BulkERecEnumerator;
  // Loop through the record list
  while (bulkRecs.MoveNext()) {
    // Get a record, which will be gunzipped
    ERec record = (ERec)bulkRecs.Current;
    // Display its properties or format for export
    ...
  }
}
```

Performance impact for bulk export records

The bulk export feature can reduce memory usage in your application.

Unneeded overhead is typically experienced when exporting records from an MDEX Engine without the Bulk Export feature. Currently, the front-end converts the on-wire representation of all the records into objects in the API language, which is not appropriate for bulk export given the memory footprint that results from multiplying a large number of records by the relatively high overhead of the Endeca record object format. For export, converting all of the result records to API language objects at once requires an unacceptable amount of application server memory.

Reducing the per-record memory overhead allows you to output a large number of records from existing applications. Without this feature, applications that want to export large amounts of data are required

to split up the task and deal with a few records at a time to avoid running out of memory in the application server's threads. This division of exports adds query processing overhead to the MDEX Engine which reduces system throughput and slows down the export process.

In addition, the compressed format of bulk-export records further reduces the application's memory usage.

Part 2

Advanced Search Features

- *Spelling Correction and Did You Mean*
- *Stemming and Thesaurus*
- *Automatic Phrasing*
- *Stop Words*
- *Relevance Ranking*

Chapter 5

Spelling Correction and Did You Mean

This section describes how to implement the Spelling Correction and Did You Mean features of the Endeca MDEX Engine.

About Spelling Correction and Did You Mean

The Spelling Correction and Did You Mean features of the Endeca MDEX Engine enable search queries to return expected results when the spelling used in query terms does not match the spelling used in the result text (that is, when the user misspells search terms).

Spelling Correction operates by computing alternate spellings for user query terms, evaluating the likelihood that these alternate spellings are the best interpretation, and then using the best alternate spell-corrected query forms to return extra search results. For example, a user might search for records containing the text *Abrham Lincoln*. With spelling correction enabled, the Endeca MDEX Engine will return the expected results: those containing the text *Abraham Lincoln*.

Did You Mean (DYM) functionality allows an application to provide the user with explicit alternative suggestions for a keyword search. For example, if a user searches for *valle* in the sample wine data, he or she will get six results. The terms *valley* and *vale*, however, are much more prevalent (2,414 results and 20 results respectively.) When this feature is enabled, the MDEX Engine will respond with the six results for *valle*, but will also suggest that *valley* or *vale* may be what the end-user actually intended. If multiple suggestions are returned, they will be sorted and presented according to the closeness of the match.

The Endeca MDEX Engine supports two complementary forms of Spelling Correction:

- Auto-correction for record search and dimension search.
- Explicit spelling suggestions for record search (the "Did you mean?" dialog box).

Either or both features can be used in a single application, and all are supported by the same underlying spelling engine and Spelling Correction modules.

The behavior of Endeca spelling correction features is application-aware, because the spelling dictionary for a given data set is derived directly from the indexed source text, populated with the words found in all searchable dimension values and properties. For example, in a set of records containing computer equipment, a search for *graphi* might spell-correct to *graphics*. In a different data set for sporting equipment, the same search might spell-correct to *graphite*.

Endeca Spelling Correction features include a number of tuning parameters to control performance, behavior, and result presentation. This section describes the steps necessary to enable spelling correction for record and/or dimension search, and provides a reference to the tuning parameters

provided to allow applications to obtain various behavior and performance trade-offs from the spelling engine.

Spelling modes

Endeca spelling features compute contextual suggestions at the full query level.

That is, suggestions may include one or more corrected query terms, which can depend on context such as other words used in the query. To determine these full query suggestions, the MDEX Engine relies on low-level spelling modules to compute single-word suggestions, that is, words similar to a given user query term and contained within the application-specific dictionary.

Aspell and Espell spelling modules

The MDEX Engine supports two internal spelling modules, either or both of which can be used by an application:

- `Aspell` is the default module. It supports sound-alike corrections (using English phonetic rules). It does not support corrections to non-alphabetic/non-ASCII terms (such as *café*, *1234*, or *A&M*).
- `Espell` is a non-phonetic module. It supports non-phonetic (edit-distance-based) correction of any term.

Generally, applications that only need to correct normal English words can enable just the default Aspell module. Applications that need to correct international words, or other non-English/non-word terms (such as part numbers) should enable the Espell module.

In certain cases (such as an English-language application that also needs to correct part numbers), both Aspell and Espell can be enabled.

Supported spelling modes

Module selection is performed at index time through the selection of a spelling mode. The supported spelling modes are (the options below represent command line options you can specify to Dgidx):

- `aspell` – Use only the Aspell module. This is the default mode.
- `espell` – Use only the Espell module.
- `aspell_OR_espell` – Use both modules, segmenting the dictionary so that Aspell is loaded with all ASCII alphabetic words and Espell is loaded with other terms. Consult Aspell when attempting to correct ASCII alphabetic words; consult Espell to correct other words.
- `aspell_AND_espell` – Use both modules, each loaded with the full application dictionary. Consult both modules to correct any word, selecting the best suggestions from the union of the results.
- `disable` – Disable the Spelling Correction feature.

Disabling spelling correction on individual queries

This topic describes how to disable spelling correction and DYM suggestions on individual queries.

You may discover that it is desirable to disable spelling correction in order to reduce the cost of running some queries in performance-sensitive applications. For example:

- Queries where the MDEX Engine needs to perform matching on a very large number of terms all of which need to be ranked for spelling correction suggestions.

• Queries using terms derived directly from the raw data. For example, if your end users are searching for terms that are unique to their field, it may be desirable to disable spelling correction suggestions for those terms.

To disable spelling correction for a particular query:

Use a query configuration option, `spell`, with a parameter `nospell`.

This option has the following characteristics:

• Works for both record and dimension search.
• Disables both Aspell and Espell spelling correction modes.
• Disables spelling correction and DYM suggestions.
• Requires spelling to be enabled in Dgidx or in the Dgraph. Otherwise, this option has no effect.
• Requires that you provide a `nospell` parameter to it. Providing a parameter other than `nospell` results in a warning in the error log, and spelling correction proceeds as if the option were not provided to the MDEX Engine.
• Reduces the performance cost of a particular query. You can include this option in your front-end application for particular queries if you observe that disabling spelling correction is beneficial for increasing performance of your application overall. However, there is no need to modify your existing application if you don't observe a performance penalty from using spelling correction.

Examples

In the presentation API, use the `spell+nospell` option with `Ntx` and `Dx` parameters.

For example, to disable spelling correction for a dimension search query for `"blue suede shoes"`, change the query from this syntax:

```
D=blue+suede+shoes&Dx=mode+matchallpartial
```

To the following syntax:

```
D=blue+suede+shoes&Dx=mode+matchallpartial+spell+nospell
```

In the Dgraph URL, specify the `spell+nospell` value to the `opts` parameter. For example, change this type of query from this syntax:

```
/search?terms=blue+suede+shoes&opts=mode+matchallpartial
```

To the following syntax:

```
/search?terms=blue+suede+shoes&opts=mode+matchallpartial+spell+nospell
```

In the Java Presentation API, you can disable spelling for a specific query as shown in this example:

```
ENEQuery nequery = new ENEQuery();
nequery.setDimSearchTerms("blue suede shoes");
nequery.setDimSearchOpts("spell nospell");
```

In the .NET API, you can disable spelling for a specific query as shown in this example:

```
ENEQuery nequery = new ENEQuery();
nequery.DimSearchTerms = "blue suede shoes";
nequery.DimSearchOpts = "spell nospell";
```

Spelling dictionaries created by Dgidx

No index configuration setup is strictly necessary to enable spelling correction.

By default, all words contained in searchable dimensions and properties will be considered as possible spell correction recommendations. But in practice, to achieve the best possible spelling correction behavior and performance, it is typically necessary to configure bounds on the list of words available for spelling correction, commonly known as the dictionary.

The application-specific spelling dictionary is created by Dgidx. As Dgidx creates search indexes of property and dimension value text, it accumulates lists of words available for spelling correction into the following files:

- `<db_prefix>.worddat` (for the Aspell module)

   > 📝 **Note:** The `<db_prefix>.worddat` file for the Aspell module is also reloaded into the MDEX Engine each time you run the `admin?op=updateaspell` administrative command. This command lets you make updates to the Aspell spelling dictionary without stopping and restarting the Dgraph.

- `<db_prefix>.worddatn_default` (for the Espell module)

where *<db_prefix>* is the output index prefix.

These files contain application-specific dictionary words separated by new-line characters. Duplicate words listed in these files are ignored.

These files are automatically compiled by the Dgidx during the indexing operation.

Configuring spelling in Developer Studio

You can set constraints for the spelling dictionaries in Developer Studio.

By default, Dgidx examines dimensions and properties enabled for search and adds words that are larger than 3 characters and smaller than 16 characters to the dictionary. However, because

performance of spelling correction in the MDEX Engine depends heavily on the size of the dictionary, you can set constraints on the contents of the dictionary. For example, you might choose to either increase the default from a minimum of 3 characters or reduce the default from a maximum of 16 characters. These configuration settings are useful for improving the performance of spell-corrected search operations at runtime.

These configuration options can be used to tune and improve the types of spelling corrections produced by the MDEX Engine. For example, setting the minimum number of word occurrences can direct the attention of the spelling correction algorithm away from infrequent terms and towards more popular (frequently occurring) terms, which might be deemed more likely to correspond to intended user search terms.

To configure spelling dictionary entries:

1. In the Project Explorer, expand **Search Configuration**.
2. Double-click **Spelling** to display the Spelling editor.
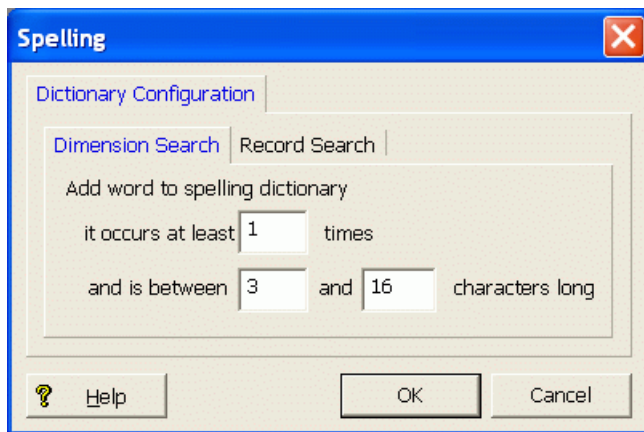
3. You can separately configure entries in the dictionary based for dimension search and record search. Therefore, select either the **Dimension Search** tab or the **Record Search** tab.

   In this example, the **Dimension Search** tab is selected.

4. Set the constraints for adding words to the spelling dictionary:

Field	Description
it occurs at least n times	Sets the minimum number of times the word must appear in your source data before the word should be included in the spelling dictionary.
and is between n1 and n2 characters long	Sets the minimum (n1) and maximum (n2) lengths of a word for inclusion in the dictionary.

5. If desired, select the other tab and set the constraints for that type of search.
6. Click **OK**.
7. Choose **Save** from the File menu to save the project changes.

Modifying the dictionary file

You can modify or replace the Aspell dictionary file. Use the `admin?op=updateaspell` operation for the Dgraph which causes updates to the Aspell dictionary file.

While the dictionary files automatically generated by Dgidx are generally adequate for most applications (especially when using a reasonable value for the minimum number of word occurrences), additional improvements in application-specific spelling behavior can be achieved through modification or replacement of the automatic dictionary file (Aspell module only).

For example, in applications with a specific set of technical terminology that requires focused spelling correction, you can replace the automatic dictionary with a manually-generated list of technical terms combined with a simple list of common words (such as `/usr/dict/words` on many UNIX systems).

About the admin?op=updateaspell operation

The `admin?op=updateaspell` administrative operation lets you rebuild the aspell dictionary for spelling correction from the data corpus without stopping and restarting the MDEX Engine.

The `admin?op=updateaspell` operation performs the following actions:

- Crawls the text search index for all terms
- Compiles a text version of the `aspell` word list
- Converts this word list to the binary format required by `aspell`
- Causes the Dgraph to finish processing all existing preceding queries and temporarily stop processing incoming queries
- Replaces the previous binary format word list with the updated binary format word list
- Reloads the `aspell` spelling dictionary
- Causes the Dgraph to resume processing queries waiting in the queue

The Dgraph applies the updated settings without needing to restart.

Only one `admin?op=updateaspell` operation can be processed at a time.

The `admin?op=updateaspell` operation returns output similar to the following in the Dgraph error log:

```
...
aspell update ran successfully.
...
```

🖉 **Note:** If you start the Dgraph with the `-v` flag, the output also contains a line similar to the following:

```
Time taken for updateaspell, including wait time on any
previous updateaspell, was 290.378174 ms.
```

Enabling language-specific spelling correction

If your application involves multiple languages, you may want to enable language-specific spelling correction.

For information on how to enable this feature, see the "Using Internationalized Data" section.

Related Links

Using internationalized data in your Endeca application on page 183
> The MDEX Engine supports the following 22 languages:

Dgidx flags for Spelling Correction

The spelling mode can be selected using the Dgidx `--spellmode` flag.

The default spelling mode is `aspell`, which enables only the Aspell module.

The full set of supported spelling modes is:

- `--spellmode aspell`
- `--spellmode espell`
- `--spellmode aspell_OR_espell`
- `--spellmode aspell_AND_espell`
- `--spellmode disable`

Behaviors for these modes are described in the "Spelling modes" topic. If a spelling mode that includes use of the Espell module is enabled, an additional Dgidx flag, `--spellnum`, can be used to control the contents of the Espell dictionary.

The default is to disable `--spellnum`. With this flag enabled, the Espell dictionary will be allowed to contain non-word terms. A word term is one that contains only ASCII alphabetic characters and ISO-Latin1 word characters listed in Appendix C of the *MDEX Engine Basic Development Guide*. In default mode, non-word terms are not allowed in the Espell dictionary.

> **Note:** Auto-correct should be relatively conservative. You only want the engine to complete the correction when there is a high degree of confidence. For more aggressive suggestions, it is best to use Did You Mean.

Related Links

Spelling modes on page 62
> Endeca spelling features compute contextual suggestions at the full query level.

Dgraph flags for enabling Spelling Correction and DYM

Four Dgraph flags enable the use of the Spelling Correction and DYM features. You can also use the `admin?op=updateaspell` operation on the Dgraph to update the Aspell spelling dictionary while running partial updates (without having to stop and restart the MDEX Engine).

Dgraph --spellpath flag

To enable use of spelling features in the MDEX Engine, you must first use the `--spellpath` flag to specify the path to the directory containing the spelling support files.

If you are using the Endeca Application Controller to provision and run the Dgraph, then this flag is set automatically. By default, the Dgraph component looks for the Aspell spelling support files in its input directory (that is, the Dgidx output directory). If you want to specify an alternative location, you can do so using the `spellPath` element in the WSDL, or by specifying arguments to the Dgraph in Endeca Workbench.

If you need to, you can specify the `--spellpath` parameter yourself. The value of the `--spellpath` parameter typically matches the value specified for `--out` on the `dgwordlist` program.

Note the following about the `--spellpath` flag:

- The directory passed to the `--spellpath` flag must be an absolute path. Paths relative to the current working directory are not allowed. This directory must have write permissions enabled for the user starting the MDEX Engine process.
- The `--spellpath` option on the MDEX Engine is required for spelling features to be enabled, but this flag does not activate any spelling features on its own. Additional flags are required to enable actual spelling correction in the MDEX Engine.

Additional Dgraph flags to enable spelling correction

The following MDEX Engine flags enable the supported spelling features. Any or all of these options can be specified in combination, because they control independent features.

Dgraph flag	Spelling feature
`--spl`	Enables automatic spelling correction (autosuggest) for record and dimension searches.
`--dym`	Enables explicit spelling suggestions (Did You Mean) for record search operations
`--spell_bdgt num`	Sets the maximum number (*num*) of variants to be considered when computing any spelling correction (autosuggest). The default value is 32.

If `--spl` and `--dym` are both specified, explicit spelling suggestions are guaranteed not to reuse suggestions already consumed by automatic spelling correction (autosuggest). For example, the MDEX Engine will not explicitly suggest "Did you mean 'Chardonnay'?" if it has already automatically included record search results matching *Chardonnay*.

Spelling corrections generated by the MDEX Engine are determined by considering alternate versions of the user query. The computation and scoring of alternate queries takes time and can decrease performance, especially in the case of search queries with many terms. To limit the amount of spelling work performed for any single search query, use the `--spell_bdgt` flag to place a maximum on the number of variants considered for all spelling and Did You Mean corrections.

For information on other spelling-related flags, see the *Dgraph Flags* topic in the *Oracle Endeca Commerce Administrator's Guide*.

URL query parameters for Spelling Correction and DYM

DYM suggestions are enabled by the `Nty` parameter.

No special URL query parameters are required for the dimension search and record search auto-correction features (`--spl` options). These features automatically engage when appropriate, given configuration settings and the user's query.

> 🖉 **Note:**
>
> To disable spelling correction on individual queries, you can use the `Ntx` and `Dx` parameters with the `spell+nospell` option specified.

Did You Mean suggestions for record search require the use of the `Nty=1` URL query parameter. For example:

```
<application>?N=0&Ntk=Description&Ntt=sony&Nty=1
```

Setting `Nty=0` (or omitting the `Nty` parameter) prevents Did You Mean suggestions from being returned. This allows an application to control the generation of suggestions after click-through from a previous suggestion.

Spelling Correction and DYM API methods

There are no modifications that are strictly necessary in the Presentation API code to support spelling correction. However, there are API calls that return information about automatic spelling correction and DYM objects.

Spelling corrected results for both dimension search and record search operations are returned as normal search results.

🖉 **Note:** You can disable spelling correction suggestions (autosuggest), auto-correct suggestions and DYM suggestions on individual queries using the `"spell nospell"` option in `nequery.set¬ DimSearchOpts` parameter of the ENEQuery method (Java), or in `nequery.DimSearchOpts` property (.NET). For more information, see the topic on disabling spelling correction on individual queries.

Optionally, applications can display information about automatic spelling corrections or Did You Mean suggestions for dimension or record search operations using the automatically-generated `ESearchReport` objects returned by the MDEX Engine.

For example, consider the following query, which performs two record search operations (a search for *cdd* in the AllText search interface and a search for *sny* in the Manufacturer search interface):

```
<application>?N=0&Ntk=AllText|Manufacturer&Ntt=cdd|sny&Nty=1
```

The Java `Navigation.getESearchReportsComplete()` method and the .NET `Navigation.ESearchReportsComplete` property return a map of search keys to a list of `ESearchReport` objects that provides access to the information listed in the following two tables.

ESearchReport Java method	Returned value
getKey()	AllText
getTerms()	Cdd
getSearchMode()	MatchAll
getMatchedMode()	MatchAll
getNumMatchingResults()	122
getAutoSuggestions().get(0).getTerms()	Cd
getDYMSuggestions().get(0).getTerms()	Ccd
getDYMSuggestions().get(0).getNumMatchingResults()	6
getDYMSuggestions().get(1).getTerms()	Cdp
getDYMSuggestions().get(1).getNumMatchingResults()	7
getKey()	Manufacturer
getTerms()	Sny
getSearchMode()	MatchAll

ESearchReport Java method	Returned value
getMatchedMode()	MatchAll
getNumMatchingResults()	121
getAutoSuggestions().get(0).getTerms()	Sony

ESearchReport .NET property	Returned value
Key	AllText
Terms	Cdd
SearchMode	MatchAll
MatchedMode	MatchAll
NumMatchingResults	122
AutoSuggestions[(0)].Terms	Cd
DYMSuggestions[(0)].Terms	Ccd
DYMSuggestions[(0)].NumMatchingResults	6
DYMSuggestions[(1)].Terms	Cdp
DYMSuggestions[(1)].NumMatchingResults	7
Key	Manufacturer
Terms	Sny
SearchMode	MatchAll
MatchedMode	MatchAll
NumMatchingResults	121
AutoSuggestions[(0)].Terms	Sony

Note that the auto-correct spelling corrections and the explicit Did You Mean suggestions are grouped with related record search operations. (In this case, *cd* is the spelling correction for *cdd* and *sony* is the spelling correction for *sny*.)

Java example of displaying autocorrect messages

```
// Get the Map of ESearchReport objects
Map recSrchRpts = nav.getESearchReports();
if (recSrchRpts.size() > 0) {
  // Get the user's search key
  String searchKey = request.getParameter("Ntk");
  if (searchKey != null) {
    if (recSrchRpts.containsKey(searchKey)) {
      // Get the ERecSearchReport for the search key
      ESearchReport srchRpt = (ESearchReport)recSrchRpts.get(searchKey);
      // Get the List of auto-correct values
      List autoCorrectList = srchRpt.getAutoSuggestions();
      // If the list contains Auto Suggestion objects,
      // print the value of the first corrected term
      if (autoCorrectList.size() > 0) {
        // Get the Auto Suggestion object
        ESearchAutoSuggestion autoSug = (ESearchAutoSuggestion)autoCor¬
```

```
rectList.get(0);
       // Display autocorrect message
       %>Corrected to <%= autoSug.getTerms() %>
     }
   }
 }
}
```

.NET example of displaying autocorrect messages

```
// Get the Dictionary of ESearchReport objects
IDictionary recSrchRpts = nav.ESearchReports;
// Get the user's search key
String searchKey = Request.QueryString["Ntk"];
if (searchKey != null) {
  if (recSrchRpts.Contains(searchKey)) {
    // Get the first Search Report object
    IDictionaryEnumerator ide = recSrchRpts.GetEnumerator();
    ide.MoveNext();
    ESearchReport searchReport = (ESearchReport)ide.Value;
    // Get the List of auto-correct objects
    IList autoCorrectList = searchReport.AutoSuggestions;
    // If the list contains Auto Suggestion objects,
    // print the value of the first corrected term
    if (autoCorrectList.Count > 0) {
      // Get the Auto Suggestion object
      ESearchAutoSuggestion autoSug = (ESearchAutoSuggestion)autoCor¬
rectList[0];
      // Display autocorrect message
      %>Corrected to <%= autoSug.Terms %>
    }
  }
}
```

Java example of creating links for Did You Mean suggestions

```
// Get the Map of ESearchReport objects
Map dymRecSrchRpts = nav.getESearchReports();
if (dymRecSrchRpts.size() > 0) {
  // Get the user's search key
  String searchKey = request.getParameter("Ntk");
  if (searchKey != null) {
    if (dymRecSrchRpts.containsKey(searchKey)) {
      // Get the ERecSearchReport for the user's search key
      ESearchReport searchReport = (ESearchReport) dymRec¬
SrchRpts.get(searchKey);
      // Get the List of Did You Mean objects
      List dymList = searchReport.getDYMSuggestions();
      // If the list contains Did You Mean objects, provide a
      // link to search on the first suggested term
      if (dymList.size() > 0) {
        // Get the Did You Mean object
       ESearchDYMSuggestion dymSug = (ESearchDYMSuggestion)dymList.get(0);

        String sug_val = dymSug.getTerms();
        if (sug_val != null){
          // Display didyoumean link
          %>Did You Mean: <%= sug_val %>
        }
      }
```

```
      }
    }
}
```

.NET example of creating links for Did You Mean suggestions

```
dd
// Get the Dictionary of ESearchReport objects
IDictionary dymRecSrchRpts = nav.ESearchReports;
// Get the user's search key
String dymSearchKey = Request.QueryString["Ntk"];
if (dymSearchKey != null) {
  if (dymRecSrchRpts.Contains(dymSearchKey)) {
    // Get the first Search Report object
    IDictionaryEnumerator ide = dymRecSrchRpts.GetEnumerator();
    ide.MoveNext();
    ESearchReport searchReport = (ESearchReport)ide.Value;
    // Get the List of DYM objects
    IList dymList = searchReport.DYMSuggestions;
    // If the list contains DYM objects, print the value
    // of the first suggested term
    if (dymList.Count > 0) {
      // Get the DYM object
      ESearchDYMSuggestion dymSug = (ESearchDYMSuggestion)dymList[0];
      String sug_val = dymSug.Terms;
      String sug_num = dymSug.NumMatchingResults.ToString();
      // Display DYM message
      if (sug_val != null){
        %>Did You Mean: <%= sug_val %>
      }
    }
  }
}
```

Dgraph tuning flags for Spelling Correction and Did You Mean

The MDEX Engine provides a number of advanced tuning options that allow you to achieve various performance and behavioral effects in the Spelling Correction feature.

An explanation of these tuning parameters relies on an understanding of the internal process used by the MDEX Engine to generate spelling suggestions.

At a high level, the spelling engine performs the following steps to generate alternate spelling suggestions for a given query:

1. If the user query generates more than a certain number of hits, then do not generate suggestions. This threshold number of hits is the hthresh parameter.
2. For each word in the user's search query, compute the *N* most similar words in the data set from a spelling similarity perspective (N words are computed for each user query term). This number is set internally and is not user-configurable.
3. For each word in the user's search query, from the set of *N* most similar spelling words determined in step 2, pick the *M* most likely replacement words (where M<=N), based on a scoring process that combines factors such as spelling similarity and word frequency (number of hits). This narrows the

set of possible spelling replacements for each user query word to *M*. This number is set internally and is not user-configurable.

4. Consider combinations of these replacements for the user query words, limiting consideration to only combinations that gain more than a threshold percentage number of hits relative to the user's original query, without reducing the number of query terms matched. This gain threshold percent is set internally and is not user-configurable.

5. Scoring each such alternate query using a combination of factors such as spelling similarity of words used and the number of hits generated by the query, select the *K* best queries and use them as suggestions. *K* (the maximum number of replacement queries to generate) is called the `nsug` parameter.

6. Finally, consider alternate queries computed by changing the word divisions in the user's query, with the word-break analysis feature. Using the same scoring technique and limits on suggested queries described in steps 4 and 5, include alternate word-break queries in the final suggestion set.

User-configurable parameters

The following table summarizes the user-configurable parameters described in the above process:

Parameter	Description
hthresh	Specifies the threshold number of hits at or above which spelling suggestions will not be generated. That is, above this threshold, the spelling feature is disabled, allowing correctly spelled queries to return only actual (non-spell-corrected) results. Results that don't match all query terms don't count toward the hthresh threshold. For example, if you have a 1000 results which are all partial matches (they match only a proper subset of the query terms) and `hthresh` is set to 1, then spelling correction will still engage because you have 0 full matches. Note that the case where results only match a proper subset of the query terms can only occur when the match mode is set appropriately to allow such partial matches (`matchany`, `matchpartial`, `matchpartialmax`, and so on).
nsug	Specifies the maximum number of alternate spelling queries to generate for a single user search query.
sthresh	Specifies the threshold spelling similarity score for words considered by the spelling correction engine. Scores are based on a scale where 100 points corresponds approximately to an edit distance of 1. The cost associated with correcting a query term is higher if the term corresponds to an actual word in the data. That is, correcting *modem* to *model* is considered a more significant edit than correcting *modek* to *model*, if *modem* occurs as a word in the data but *modek* does not. The threshold applies to the entire query; for multi-word queries, the edit scores associated with correcting multiple words are added together, and the sum cannot exceed the threshold. For details on the interaction of the `--spl_sthresh` and `--dym_sthresh` settings, see the section below.
glom	Specifies that cross-property matches are considered valid when scoring replacement queries. By default, hits that result from applying some queries terms to one text field on a record and other terms to a different text field are not counted. In some cases, these results are desirable and should be considered when computing spelling suggestions.
nobrk	Specifies that the word-break analysis portion of the spelling correction process described above is disabled.

Each of these parameters can be specified independently for each of the spelling correction features:

- For record and dimension search auto-correct, the `--spl_` prefix is used (for example, `--spl_nsug`). The flag --spl by itself enables auto-suggest spelling corrections for record search and dimension search.
- For explicit suggestions, the `--dym_` prefix is used (for example, `--dym_nsug`). The flag `--dym` by itself enables Did You Mean explicit query spelling suggestions for record search queries.
- For parameters that apply to all of the above, the `--spell_` prefix is used.

For additional configuration of the word-break analysis feature (beyond disabling it with `--spell_nobrk`), use the following `--wb_` flags:

- `--wb_noibrk` disables the insertion of breaks in word-break analysis.
- `--wb_norbrk` disables the removal of breaks in word-break analysis.
- `--wb_maxbrks` specifies the maximum number of word breaks to be added to or removed from a query. The default is one.
- `--wb_minbrklen` specifies the minimum length of a new term created by word-break analysis. The default is two.

Summary of the Spelling Correction and Did You Mean options

The following table summarizes the complete set of options:

Feature	Available Dgraph flags
Record Search and Dimension Search	--spl, --spl_hthresh, --spl_nsug, --spl_sthresh
Did You Mean	--dym, --dym_hthresh, --dym_nsug, --dym_sthresh
Record Search and Did You Mean	--spell_glom Note that the --spell_glom option does not apply to dimension search, because cross-property matching is inherently incompatible with the dimension search feature. Dimension search matches always represent a single dimension value.
Record Search, Dimension Search, and Did You Mean	--spell_nobrk, --wb_noibrk, --wb_norbrk, --wb_maxbrks, --wb_minbrklen

Note: Terms that appear in the corpus more than *2×max(spl_hthresh, dym_hthresh)* are never corrected, because such terms are unlikely to be misspelled.

Interaction of --spl_sthresh and --dym_sthresh

The `--spl_sthresh` and `--dym_sthresh` flags are used to set the threshold spelling correction score for words used by the auto-correct or DYM engines, respectively. This is the threshold at which the engine will consider the suggestion. Words that qualify have a score below a given threshold. The higher the edit distance for a term, the higher the score. The default for `--spl_sthresh` is 125, and the default for `--dym_sthresh` is 175.

Based on these default values, if a particular suggestion has a score of 100, it can be used for either DYM or auto-correct, and if it has a score of 200, it is not used by either. If the suggested word has a score better (that is, lower) than the default DYM threshold of 175, but not good enough (that is, higher) than the default auto-correct threshold of 125, it qualifies only for DYM.

A higher value for either of these settings generally results in more suggestions being generated for a misspelled word. In an example query against the sample wine data, changing the `--dym_sthresh` value from 175 to 225 increased the number of terms considered for DYM from one to ten. However, raising scores too high could result in a lot of noise. That is to say, it is generally a good thing if nonsense strings used as search terms receive neither auto-correct nor DYM suggestions.

Related Links

About word-break analysis on page 79
> Word-break analysis allows the Spelling Correction feature to consider alternate queries computed by changing the word divisions in the user's query.

How dimension search treats number of results

Dimension search results may vary if spelling correction is performed.

An important note applies to the options and behavior associated with dimension search spelling correction: in situations where the number of results is evaluated by an option or in the scoring of words or queries performed by the spelling engine, dimension search uses an alternate definition of number of results. Instead of using the simple number of hits returned to the user as this value (which is perfectly reasonable in the case of record search), dimension search instead uses the number of records associated with the set of dimension value search results computed for a given query.

In other words, dimension search follows an additional level of indirection to weight the dimension value results computed by spelling suggestion queries according to the number of records that these dimension values would lead to if selected in a navigation query. This alternate definition of number or results allows consistent behavior between spelling corrections computed for dimension and record search operations when given the same query terms.

Troubleshooting Spelling Correction and Did You Mean

This topic provides some corrective solutions for spelling correction problems.

If spell-corrected results are not returned for words with expected spell-corrected options in the data, check the potential problems described in this topic.

When debugging spelling behavior, pay close attention to the errors of the Dgraph on startup, at which point problems in spelling configuration are typically reported.

Did You Mean and stop words interaction

Did You Mean can in some cases correct a word to one on the stop words list.

Did You Mean and query configuration

If a record search query produces Did You Mean options, each DYM query has the same configuration as the initial record search query. For example, if the record search query had **Allow cross field matches** set to **On Failure**, then the DYM query also runs with cross field matching set to **On Failure**.

Interaction of Aspell, Espell and DYM

This section is relevant to you if you are using `aspell_AND_espell` mode with DYM enabled. It describes the interaction of both spelling modes with DYM and explains why in some instances,

suggestions that should have been found by Aspell or Espell are not considered by DYM. In other words, you may observe that in some instances user-entered words with misspellings in them do not return DYM suggestions, if the `aspell_AND_espell` mode is used.

The following statements describe the reasons behind this behavior in more detail:

- Both spelling modes, Aspell or Espell, work by generating a list of suggestion results. These suggestions are weighted based on the lowest score, according to a scoring algorithm.
- Aspell and Espell generate scores based on different scoring algorithms (described below in this section).
- When both modes are used, as is the case with `aspell_AND_espell`, DYM uses the union of the scored suggestions provided by each spelling mode, and keeps the top 10 terms from the combined list, based on the lowest scores.
- As a result, some suggestions found by Espell (that could have been relevant) do not pass the scoring criteria in the combined list, and are thus not considered by DYM.
- The following statements discuss how scores are calculated for each of the spelling engines (Aspell and Espell):
    - For information on the GNU Aspell scoring algorithm, see the documentation for this open source product.
    - The Espell scoring algorithm uses the following formula:

      ```
      (85 - num_matching_characters_in_prefix)* edit_distance
      ```

      The parameter `edit_distance` specifies a regular Levenshtein distance (see the Internet for more information). In `edit_distance`, character swaps, insertions and deletions count as an edit distance of 1.

      The `num_matching_characters_in_prefix` is a number of all matching characters before a mismatch occurs. For example, for the term "java", this number is 2 (matching "j" and "a"); for the term "jsva", this number is 1 (matching only "j").

The directory specified for the --spellpath flag

- The directory specified in the `--spellpath` flag to the MDEX Engine must be an absolute path. If a relative path is used, an error message is sent to the standard error output in the format:

  ```
  [Warning] OptiSpell couldn't open pwli file
  "<--spell param>/<db_prefix>-aspell.pwli"
  'Permission denied'
  ```

- The directory specified for the `--spellpath` flag must either be writable or already contain a valid `.pwli` file that contains an absolute path to the `spelldat` binary dictionary file. Check the permissions on this directory. If the directory is not writable or does not contain a valid `.pwli` file, an error is issued as in the previous example.

Performance impact for Spelling Correction and Did You Mean

Spelling correction performance is impacted by the size of the dictionary in use.

Spell-corrected keyword searches with many words, in systems with very large dictionaries, can take a disproportionately long time to process relative to other MDEX Engine requests. Those searches

can cause requests that immediately follow such a search to wait while the spelling recommendations are being sought and considered.

Because of this, it is important to carefully analyze the performance of the system together with application requirements prior to production application deployment.

Consider also whether performance could be improved if you disable spelling correction on individual queries. For information on disabling spelling correction on individual queries, see the topic in this guide.

Related Links

Disabling spelling correction on individual queries on page 62
> This topic describes how to disable spelling correction and DYM suggestions on individual queries.

About compiling the Aspell dictionary

The Aspell dictionary must be compiled before it can used by the MDEX Engine.

The Espell dictionary is automatically compiled at index time, and requires no further processing. But if the selected spelling mode includes use of the Aspell module, the Aspell dictionary must be compiled. If you are manually compiling this file, perform this step after indexing but before starting the MDEX Engine.

Compilation transforms the text-based dictionary into a binary dictionary file suitable for use by Aspell module in the MDEX Engine. This indexed form of the dictionary is contained in a file with a name of the form `<dbPath>-aspell.spelldat`.

Use one of the following ways to compile the dictionary file:

- Automatically, by running the `admin?op=updateaspell` administrative operation. For information about this operation, see the topic in this section.
- Manually, by running the `dgwordlist` utility script.
- Automatically, by letting the Endeca Application Controller create them implicitly in the Dgidx component.

Related Links

About the admin?op=updateaspell operation on page 66
> The `admin?op=updateaspell` administrative operation lets you rebuild the aspell dictionary for spelling correction from the data corpus without stopping and restarting the MDEX Engine.

Compiling the dictionary manually on page 77
> The `dgwordlist` utility script is provided to compile the Aspell dictionary.

Compiling the dictionary with EAC on page 78
> The Dgidx component contains a `run-aspell` setting that specifies Aspell as the spelling correction mode for the implementation.

Compiling the dictionary manually

The `dgwordlist` utility script is provided to compile the Aspell dictionary.

To manually compile the text-based `worddat` dictionary into the binary `spelldat` dictionary, you must use the utility script `dgwordlist` (on UNIX; on Windows, it is `dgwordlist.exe`).

The usage for `dgwordlist` is:

```
dgwordlist [--out <output_dir>] [--aspell <aspell_location>]
           [--datfiles <aspell_dat_files_location>] [--help]
           [--version] <dbPath>
```

Argument for dgwordlist	Description
`--out`	Specifies the directory where the resulting binary `spelldat` dictionary file is placed. If not specified, this defaults to the same directory where the input index files reside (`<dbPath>`).
`--aspell`	Deprecated. If you specify this flag, it is ignored. The `dgwordlist` utility no longer needs to know the location of the `Aspell` dictionary indexing program. In previous releases, this flag specified the location of Aspell. This parameter could also be omitted if `aspell` (or `aspell.exe` on Windows) was in the current path.
`--datfiles`	Specifies the input directory location containing the spelling support files. These support files contain information such as language and character set configuration (these files end with `.map` or `.dat` extensions). If not specified, this defaults to the same directory where the input index files reside (`<dbPath>`).
`<dbPath>`	Specifies a prefix path to the input index data, including the text-based `worddat` dictionary file. This should match the index prefix given to Dgidx.
`--version`	Prints the version information and exits.
`--help`	Prints the command usage and exits.

In typical operational configurations, the binary `spelldat` dictionary file created by `dgwordlist` and the `.map` and/or `.dat` files located in the `--datfiles` directory are placed in the same directory as the indexed data prior to starting the MDEX Engine.

Example of running dgwordlist

```
$ cp /usr/local/endeca/6.1.3/lib/aspell/* ./final_output
$ /usr/local/endeca/6.1.3/bin/dgwordlist
/usr/local/endeca/6.1.3/bin/aspell ./final_output/wine
Creating "./final_output/wine-aspell.spelldat"
```

Related Links

The `admin?op=updateaspell` administrative operation lets you rebuild the aspell dictionary for spelling correction from the data corpus without stopping and restarting the MDEX Engine.

Compiling the dictionary with EAC

The Dgidx component contains a `run-aspell` setting that specifies Aspell as the spelling correction mode for the implementation.

The default value of `run-aspell` is true; that is, it compiles the dictionary file for you by default and copies the Aspell files into its output directory, where the Dgraph can access them.

If you do not want the spelling dictionary to be created, you must set `run-aspell` to false in the Dgidx component. You can change this setting either by directly editing your Endeca Application Controller provisioning file, or by editing the arguments for the Dgidx component located in Endeca Workbench on the EAC Administration Console page.

Related Links

> The `admin?op=updateaspell` administrative operation lets you rebuild the aspell dictionary for spelling correction from the data corpus without stopping and restarting the MDEX Engine.

About word-break analysis

Word-break analysis allows the Spelling Correction feature to consider alternate queries computed by changing the word divisions in the user's query.

For example, if the query is *Back Street Boys*, word-break analysis could instruct the MDEX Engine to consider the alternate *Backstreet Boys*.

When word-break analysis is applied to a query, it requires that the substrings that the term is broken up into appear in the data in succession.

For example, starting with the query *box17*, word-break analysis would find *box 17*, as well as *box-17*, assuming that the hyphen (-) has not been specified as a search character. However, it would not find *17 old boxes*, because the target terms do not appear in order.

Disabling word-break analysis

You can disable the word-break analysis feature with a Dgraph flag.

Word-break analysis is enabled by default, as are its associated parameters. You can disable word-break analysis by starting the MDEX Engine with the `--spell_nobrk` flag.

Word-break analysis configuration parameters

You configure the details of word-break analysis with four Dgraph flags.

Keep in mind that word-break analysis must be enabled in order for these flags to have any effect.

The four Dgraph flags are as follows:

- To control the maximum number of word breaks to be added to or removed from a query, use the `--wb_maxbrks` flag. The default is one.
- To specify the minimum length for a new term created by word-break analysis, use the `--wb_minbrklen` flag. The default is two.
- To disable the ability of word-break analysis to remove breaks from the original term, use the `--wb_norbrk` flag.
- To disable the ability of word-break analysis to add breaks to the original term, use the `--wb_noibrk` flag.

Performance impact of word-break analysis

The performance impact of word-break analysis can be considerable, depending on your data.

Seemingly small deviations from default values (such as increasing the value of `--wb_maxbrks` from one to two) can have a significant impact, because they greatly increase the workload on the MDEX Engine. Endeca suggests that you tune this feature carefully and test its impact thoroughly before exposing it in a production environment.

Chapter 6
Stemming and Thesaurus

This section describes how to implement the Stemming and Thesaurus features of the Endeca MDEX Engine.

Overview of Stemming and Thesaurus

The Endeca MDEX Engine supports Stemming and Thesaurus features that allow keyword search queries to match text containing alternate forms of the query terms or phrases.

The definitions of these features are as follows:

- The Stemming feature allows the system to consider alternate forms of individual words as equivalent for the purpose of search query matching. For example, it is often desirable for singular nouns to match their plural equivalents in the searchable text, and vice versa.
- The Thesaurus feature allows the system to return matches for related concepts to words or phrases contained in user queries. For example, a thesaurus entry may allow searches for *Mark Twain* to match text containing the phrase *Samuel Clemens*.

Both the Thesaurus and Stemming features rely on defining equivalent textual forms that are used to match user queries to searchable text data. Because these features are based on similar concepts, and because they are typically configured to operate in conjunction to achieve desired query matching effects, both features and their interactions are discussed in one section.

About the Stemming feature

The Stemming feature broadens search results to include root words and variants of root words.

Stemming is intended to allow words with a common root form (such as the singular and plural forms of nouns) to be considered interchangeable in search operations. For example, search results for the word *shirt* will include the derivation *shirts*, while a search for *shirts* will also include its word root *shirt*.

Stemming equivalences are defined among single words. For example, stemming is used to produce an equivalence between the words *automobile* and *automobiles* (because the first word is the stem form of the second), but not to define an equivalence between the words *vehicle* and *automobile* (this type of concept-level mapping is done via the Thesaurus feature).

Stemming equivalences are strictly two-way (that is, all-to-all). For example, if there is a stemming entry for the word *truck*, then searches for *truck* will always return matches for both the singular form

(*truck*) and its plural form (*trucks*), and searches for *trucks* will also return matches for *truck*. In contrast, the Thesaurus feature supports one-way mappings in addition to two-way mappings.

Language support for stemming

The MDEX Engine supports stemming in 22 languages. For details about stemming and non-English data, see *Using Internationalized Data* on page 183.

Types of stemming matches and sort order

Stemming can produce one of three match types.

If stemming is enabled, a search on a given term (*T*) will produce one or more of these results:

- Literal matches: Any occurrence of *T* always produce a match.
- Stem form matches: Matches occur on the stem form of *T* (assuming that *T* is not a stem form). For example, if *T* is *children*, then *child* (the stem form) also matches.
- Inflected form matches: Matches occur on all inflected forms of the stem form of *T*. For example, if *T* is the verb *ran* (as in *Jane ran in the Boston Marathon*), then matches include the stem form (*run*) and inflected forms (such as *runs* and *running*). (Note that although this example is in English, stemming for inflected verb forms is not supported for English; see below for support details).

The order of the returned results depends on the sorting configuration:

- If relevance ranking is enabled and the Interpreted (interp) module is used, literal matches will always have higher priority than stem form and inflected form matches.
- If relevance ranking is not enabled but you have set a record sort order, the results will come back in that sort order.
- If relevance ranking is not enabled and there is no record sort order, the order of the results is completely arbitrary.

> 🖉 **Note:** The type of stemming described in this chapter is tokenized using a Latin-1 analyzer and is referred to as static stemming. If you are using non-English stemming, the stemming implementation may use either the Latin-1 analyzer or the Oracle Language Technology analyzer to tokenize source data. This choice results in slightly different stemming behavior. For details, see *Configuring language-specific analysis* on page 185.

Enabling stemming

Stemming is enabled in Developer Studio for a subset of supported languages listed in the **Stemming** editor.

Additional ISO-639 languages are supported but not listed in the **Stemming** editor. For details about enabling stemming with other international languages, see *Configuring language-specific analysis* on page 185.

To enable stemming:

1. Open the project in Developer Studio.
2. In the Project Explorer, expand **Search Configuration**.
3. Double-click **Stemming** to display the **Stemming** editor.

4. Check one or more of the language check boxes on the list.
5. Click **OK**.

To disable stemming, use the above procedure, but uncheck the languages for which you do not want stemming.

Supplementing the default static stemming dictionaries

You can supplement the default stemming dictionaries by specifying the `--stemming-updates` flag to Dgidx and providing an XML file of custom stemming changes. The stemming update file may include additions and deletions. Dgidx processes the file by adding and deleting entries in the stemming dictionary file.

The default stemming dictionary files are stored in `Endeca\MDEX\`*`version`*`\conf\stemming` (on Windows) and `usr/local/endeca/MDEX/version/conf/stemming` (on UNIX).

For most supported languages, the `stemming` directory contains two types of stemming dictionaries per language. One dictionary (*`<RFC 3066 Language Code>`*`_word_forms_collection.xml`) contains stemming entries that support accented characters for the particular *`<RFC 3066 Language Code>`* .

The other dictionary (*`<RFC 3066 Language Code>`*`-x-folded_word_forms_collection.xml`) contains stemming entries where all accented characters have been folded down (removed) for the particular *`<language_code>`.* If present, this is the stemming dictionary that is used if you specify `--diacritic-folding`. For details about `--diacritic-folding`, see *Mapping accented characters to unaccented characters* on page 191.

Adding entries to a stemming dictionary

To illustrate the XML you add the stemming update file, it is helpful to treat each operation (adding and deleting) as a separate use-case and show the required XML for each operation.

You specify stemming entries to add within a `<ADD_WORD_FORMS>` element and its sub-element `<WORD_FORMS_COLLECTION>`. For example, the following XML adds `apple` and its stemmed variant `apples` to the stemming dictionary:

```
<!DOCTYPE WORD_FORMS_COLLECTION_UPDATES SYSTEM "word_forms_collection_up¬
dates.dtd">
<WORD_FORMS_COLLECTION_UPDATES>
    <ADD_WORD_FORMS>
```

```
        <WORD_FORMS_COLLECTION>
            <WORD_FORMS>
                <WORD_FORM>apple</WORD_FORM>
                <WORD_FORM>apples</WORD_FORM>
            </WORD_FORMS>
        </WORD_FORMS_COLLECTION>
    </ADD_WORD_FORMS>
</WORD_FORMS_COLLECTION_UPDATES>
```

Deleting entries from a stemming dictionary

You specify stemming entries to delete in a `<REMOVE_WORD_FORMS_KEYS>` element. All word forms
that correspond to that key are deleted. For example, the following XML deletes `aalborg` and all of
its stemmed variants from the stemming dictionary:

```
<!DOCTYPE WORD_FORMS_COLLECTION_UPDATES SYSTEM "word_forms_collection_up¬
dates.dtd">
<WORD_FORMS_COLLECTION_UPDATES>
    <REMOVE_WORD_FORMS_KEYS>
        <WORD_FORM>aalborg</WORD_FORM>
    </REMOVE_WORD_FORMS_KEYS>
</WORD_FORMS_COLLECTION_UPDATES>
```

Combining deletes and adds

You can also specify a combination of deletes and then adds. Deletes are processed first and then
adds are processed. For example, the following XML removes `aachen` and then adds it and several
stemmed variants of it.

```
<!DOCTYPE WORD_FORMS_COLLECTION_UPDATES SYSTEM "word_forms_collection_up¬
dates.dtd">
<WORD_FORMS_COLLECTION_UPDATES>
    <REMOVE_WORD_FORMS_KEYS>
        <WORD_FORM>aachen</WORD_FORM>
    </REMOVE_WORD_FORMS_KEYS>
    <ADD_WORD_FORMS>
        <WORD_FORMS_COLLECTION>
            <WORD_FORMS>
                <WORD_FORM>aachen</WORD_FORM>
                <WORD_FORM>aachens</WORD_FORM>
                <WORD_FORM>aachenes</WORD_FORM>
            </WORD_FORMS>
        </WORD_FORMS_COLLECTION>
    </ADD_WORD_FORMS>
</WORD_FORMS_COLLECTION_UPDATES>
```

Syntax of the stemming update file name

The syntax of the stemming update file name must be as follows:

```
user_specified.<RFC 3066 Language Code>.xml
```

where

- *user_specified* is any string that is relevant to your application or stemming dictionary, for
  example `myAppStemmingChanges`.
- *RFC 3066 Language Code* is a two-character language code, of the stemming dictionary you
  want to update, for example,`en` or `en-us`. See ISO 639-1 for the full list of two-character codes
  and RFC 3066 for the two-character sub tag for region.

Processing the update file

To process the stemming update file, you specify the `--stemming-updates` flag to Dgidx and specify the XML file of stemming updates.

For example:

```
dgidx --stemming-updates myAppStemmingChanges.en.xml
```

Conflicts during updates

When Dgidx merges the changes in an update file into the stemming dictionary, there may be conflicts in cases where the variant for one root in the stemming dictionary is the same as a variant for another root in the update file. Any duplicate variants of different root words constitute a conflict.

In this case, Dgidx throws a warning about conflicting variants and rejects the variant that was specified in the update file.

Adding a custom static stemming dictionary

If your application requires a stemming language that is not available in the Stemming editor of Developer Studio, you can create and add a custom stemming dictionary. A custom stemming dictionary is available in addition to any stemming selections you may have enabled in Developer Studio. For example, you can enable English and Dutch, and then add an additional custom stemming dictionary for Swahili.

Although you can create any number of custom stemming dictionaries, only one custom stemming dictionary can be loaded into the MDEX Engine. You indicate which custom stemming dictionary to load with the `--lang` flag to Dgidx.

To add a custom stemming dictionary:

1. Create a custom dictionary file with stemming entries. For sample XML, see the XML schema of any default stemming dictionary stored in `<install path>\MDEX\<version>\conf\stemming`. For example, this simplified file contains one term and one stemmed variant:

```
<?xml version="1.0"?>

<!DOCTYPE WORD_FORMS_COLLECTION SYSTEM "word_forms_collection.dtd">

<WORD_FORMS_COLLECTION>

<WORD_FORMS>

<WORD_FORM>swahiliterm</WORD_FORM>

<WORD_FORM>swahiliterms</WORD_FORM>

</WORD_FORMS>

</WORD_FORMS_COLLECTION>
```

2. Once you have created the custom stemming dictionary, save the XML file with one of the following name formats:

   - If the dictionary contains *unaccented* characters and you use the Dgidx flag `--diacritic-folding`, save the file as `<RFC 3066 Language Code>-x-folded_word_forms_collection.xml`.

- If the dictionary contains *accented* characters and you are *not* using the Dgidx flag `--diacrit¬ic-folding`, save the file as `<RFC 3066 Language Code>_word_forms_collection.xml`.

  For example, the XML above would be saved as `sw_word_forms_collection.xml` where `sw` is the ISO639-1 language code for Swahili.

3. Place the XML file in `<install path>\MDEX\<version>\conf\stemming\custom`.
4. Specify the `--lang` flag to Dgidx with a `<lang id>` argument that matches the language code of the custom stemming dictionary file.
   In the example above that uses a Swahili (`sw`) dictionary, you would specify:

```
dgidx --lang sw
```

Replacing a default static stemming dictionary with a custom stemming dictionary

Rather than supplement a default stemming dictionary, you may chose to entirely replace a default stemming dictionary with a custom a stemming dictionary.

To replace a default stemming dictionary with a custom stemming dictionary:

1. Create a custom dictionary file with stemming entries. For example XML, see the XML schema of any default stemming dictionary stored in `<install path>\MDEX\<version>\conf\stemming`. For example, this simplified English stemming dictionary contains one term and one stemmed variant:

```
<?xml version="1.0"?>

<!DOCTYPE WORD_FORMS_COLLECTION SYSTEM "word_forms_collection.dtd">

<WORD_FORMS_COLLECTION>

<WORD_FORMS>

<WORD_FORM>car</WORD_FORM>

<WORD_FORM>cars</WORD_FORM>

</WORD_FORMS>

</WORD_FORMS_COLLECTION>
```

2. Once you have created the custom stemming dictionary, save the XML file with one of the following name formats:

   - If the dictionary contains *unaccented* characters and you use the Dgidx flag `--diacritic-folding`, save the file as `<RFC 3066 Language Code>-x-folded_word_forms_collection.xml`.
   - If the dictionary contains *accented* characters and you are *not* using the Dgidx flag `--diacrit¬ic-folding`, save the file as `<RFC 3066 Language Code>_word_forms_collection.xml`.

   For example, the XML above would be saved as `en_word_forms_collection.xml` where `en` is the ISO639-1 code for English.

3. Place the XML file in `<install path>\MDEX\<version>\conf\stemming\custom`.

4. Open your project in Developer Studio.

5. In the Project Explorer, expand **Search Configuration**.

6. Double-click **Stemming** to display the Stemming editor.

7. Un-check the language you want to replace.

8. Click **OK**.

9. Specify the `--lang` flag to Dgidx with a `<lang id>` argument that matches the language code
    of the custom stemming dictionary file.
    In the example above that uses an English (`en`) dictionary, you would specify:

    ```
    dgidx --lang en
    ```

About the Thesaurus feature

The Thesaurus feature allows you to configure rules for matching queries to text containing equivalent words or concepts.

The thesaurus is intended for specifying concept-level mappings between words and phrases. Even a modest number of well-thought-out thesaurus entries can greatly improve your users' search experience.

The Thesaurus feature is a higher level than the Stemming feature, because thesaurus matching and query expansion respects stemming equivalences, whereas the stemming module is unaware of thesaurus equivalences.

For example, if you define a thesaurus entry mapping the words *automobile* and *car*, and there is a stemming equivalence between *car* and *cars*, then a search for *automobile* will return matches for *automobile*, *car*, and *cars*. The same results will also be returned for the queries *car* and *cars*.

The thesaurus supports specifying multi-word equivalences. For example, an equivalence might specify that the phrase *Mark Twain* is interchangeable with the phrase *Samuel Clemens*. It is also possible to mix the number of words in the phrase-forms for a single equivalence. For example, you can specify that *wine opener* is equivalent to *corkscrew*.

Multi-word equivalences are matched on a phrase basis. For example, if a thesaurus equivalence between *wine opener* and *corkscrew* is defined, then a search for *corkscrew* will match the text *stainless steel wine opener*, but will not match the text *an effective opener for wine casks*.

Thesaurus equivalences can be either one-way or two-way:

*   One-way mapping specifies only one direction of equivalence. That is, one "From" term is mapped to one or more "To" terms, but none of the "To" terms are mapped to the "From" term. Only one "From" term can be specified.

    For example, assume you define a one-way mapping from the phrase *red wine* to the phrases *merlot* and *cabernet sauvignon*. This one-way mapping ensures that a search for *red wine* also returns any matches containing the more specific terms *merlot* or *cabernet sauvignon*. But you avoid returning matches for the more general phrase *red wine* when the user specifically searches for either *merlot* or *cabernet sauvignon*.

*   Two-way (or all-to-all) mapping means that the direction of a word mapping is equivalent between the words. For example, a two-way mapping between *stove*, *range*, and *oven* means that a search for one of these words will return all results matching any of these words (that is, the mapping marks the forms as strictly interchangeable).

When you define a two-way mapping, you do not specify a "From" term. Instead, you specify two or more "To" terms.

Unlike the Stemming module, the Thesaurus feature lets you define multiple equivalences for a single word or phrase. These multiple equivalences are considered independent and non-transitive.

For example, we might define one equivalence between *football* and *NFL*, and another between *football* and *soccer*. With these two equivalences, a search for *NFL* will return hits for *NFL* and hits for *football*, a search for *soccer* will return hits for *soccer* and *football*, and a search for *football* will return all of the hits for *football*, *NFL*, and *soccer*. However, searches for *NFL* will not return hits for *soccer* (and vice versa).

This non-transitive nature of the thesaurus is useful for defining equivalences containing ambiguous terms such as *football*. The word *football* is sometimes used interchangeably with *soccer*, but in other cases *football* refers to American football, which is played professionally in the NFL. In other words, the term *football* is ambiguous.

When you define equivalences for ambiguous terms, you do not want their specific meanings to overlap into one another. People searching for *soccer* do not want hits for *NFL*, but they may want at least some of the hits associated with the more general term *football*.

Thesaurus entries are essentially used to produce alternate forms of the user query, which in turn are used to produce additional query results. As a rule, the MDEX Engine will expand the user query into the maximum possible set of alternate queries based on the available thesaurus entries.

This behavior is particularly important in the presence of overlapping thesaurus forms. For example, suppose that you define an equivalence between *red wine* and *vino rosso*, and a second equivalence between *wine opener* and *corkscrew*. The query *red wine opener* might match the thesaurus entries in two different ways: *red wine* could be mapped to *vino rosso* based on the first entry; or *wine opener* could be mapped to *corkscrew* based on the second entry.

Using the maximal-expansion rule, this issue is resolved by expanding to all possible queries. In other words, the MDEX Engine returns hits for all of the queries: *red wine opener*, *vino rosso opener*, and *red corkscrew*.

Adding thesaurus entries

Thesaurus entries are added in Developer Studio.

To add a one-way or two-way thesaurus entry:

1. Open the project in Developer Studio.
2. In the Project Explorer, expand **Search Configuration**.
3. Double-click **Thesaurus** to display the Thesaurus view.
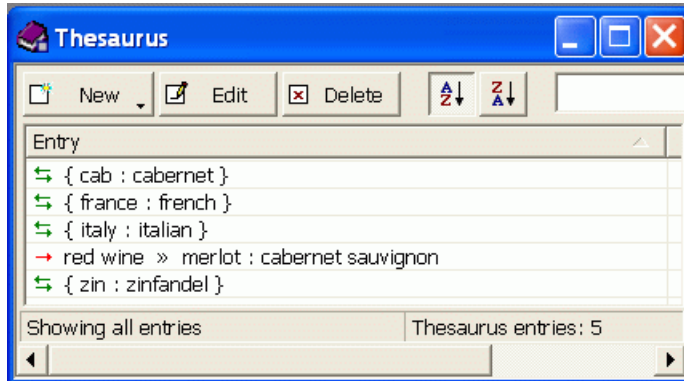
4. Click **New** and select either **One Way** or **Two Way**.

5. Configure the entry in the Thesaurus Entry dialog:

   • For a one-way entry: type in one term in the "From" field, add one or more "To" terms, and click **OK**.

   • For a two-way entry: add two or more "To" terms and click **OK**.

6. Save the project.

The Thesaurus view also allows you to modify and delete existing thesaurus entries.

Troubleshooting the thesaurus

The following thesaurus clean-up rules should be observed to avoid performance problems related to expensive and non-useful thesaurus search query expansions.

   • Do not create a two-way thesaurus entry for a word with multiple meanings. For example, *khaki* can refer to a color as well as to a style of pants. If you create a two-way thesaurus entry for `khaki = pants`, then a user's search for *khaki towels* could return irrelevant results for *pants*.

   • Do not create a two-way thesaurus entry between a general and several more-specific terms, such as:

```
top = shirt = sweater = vest
```

   This increases the number of results the user has to go through while reducing the overall accuracy of the items returned. In this instance, better results are attained by creating individual one-way thesaurus entries between the general term top and each of the more-specific terms.

   • A thesaurus entry should never include a term that is a substring of another term in the entry.

   For example, consider the two-way equivalency:

```
Adam and Eve = Eve
```

   If users type *Eve*, they get results for *Eve or (Adam and Eve)* (that is, the same results they would have gotten for *Eve* without the thesaurus). If users type *Adam and Eve*, they get results for *(Adam and Eve) or Eve*, causing the *Adam and* part of the query to be ignored.

   • Stop words such as *and* or *the* should not be used in single-word thesaurus forms. For example, if *the* has been configured as a stop word, an equivalency between *thee* and *the* is not useful.

   You can use stop words in multi-word thesaurus forms, because multi-word thesaurus forms are handled as phrases. In phrases, a stop word is treated as a literal word and not a stop word.

- Avoid multi-word thesaurus forms where single-word forms are appropriate. In particular, avoid multi-word forms that are not phrases that users are likely to type, or to which phrase expansion is likely to provide relevant additional results.

  For example, the two-way thesaurus entry:

  ```
  Aethelstan, King Of England (D. 939) = Athelstan, King Of England (D. 939)
  ```

  should be replaced with the single-word form:

  ```
  Aethelstan = Athelstan
  ```

- Thesaurus forms should not use non-searchable characters. For example, the one-way thesaurus entry:

  ```
  Pikes Peak -> Pike's Peak
  ```

  should be used only if the apostrophe (') is enabled as a search character.

Dgidx and Dgraph flags for the Thesaurus

No Dgidx flags are needed to configure the Thesaurus features.

Thesaurus entries are automatically enabled for use during text indexing and during MDEX Engine search query processing. In addition, there is no MDEX Engine configuration necessary to configure thesaurus information.

The Dgraph `--thesaurus_cutoff` flag can be used to tune performance associated with thesaurus expansion. By default, this flag is set to 3, meaning that if a search query contains more than 3 words that appear in "From" entries, none of the query terms are expanded.

No Presentation API development is necessary to use the Thesaurus feature.

Interactions with other search features

As core features of the MDEX Engine search subsystem, Stemming and the Thesaurus have interactions with other search features.

The following sections describe the types of interactions between the various search features.

Search characters

The search character set configured for the application dictates the set of available characters for stemming and thesaurus entries. By default, only alphanumeric ASCII characters may be used in stemming and thesaurus entries. Additional punctuation and other special characters may be enabled for use in stemming and thesaurus entries by adding these characters to the search character set.

The MDEX Engine matches user query terms to thesaurus forms using the following rule: all alphanumeric and search characters must match against the stemming and thesaurus forms exactly; other characters in the user search query are treated as word delimiters. For details on search characters, see the *MDEX Engine Basic Development Guide*.

Spelling

Spelling correction is a closely-related feature to stemming and thesaurus functionality, because spelling auto-correction essentially provides an additional mechanism for computing alternate versions of the user query. In the MDEX Engine, spelling is handled as a higher-level feature than stemming and thesaurus. That is, spelling correction considers only the raw form of the user query when producing alternate query forms.

Alternate spell-corrected queries are then subject to all of the normal stemming and thesaurus processing. For example, if the user enters the query *telvision* and this query is spell-corrected to *television*, the results will also include results for the alternate forms *televisions*, *tv*, and *tvs*.

Note that in some cases, the Thesaurus feature is used as a replacement or in addition to the system's standard spelling correction features. In general, this technique is discouraged. The vast majority of actual misspelled user queries can be handled correctly by the Spelling Correction subsystem. But in some rare cases, the Spelling Correction feature cannot correct a particular misspelled query of interest; in these cases it is common to add a thesaurus entry to handle the correction. If at all possible, such entries should be avoided as they can lead to undesirable feature interactions.

Stop words

Stop words are words configured to be ignored by the MDEX Engine search query engine. A stop word list typically includes words that occur too frequently in the data to be useful (for example, the word *bottle* in a wine data set), as well as words that are too general (such as *clothing* in an apparel-only data set).

If *the* is marked as a stopword, then a query for *the computer* will match to text containing the word *computer*, but possibly missing the word *the*.

Stop words are not currently expanded by the stemming and thesaurus equivalence set. For example, suppose you mark *item* as a stopword and also include a thesaurus equivalence between the words *item* and *items*. This will not automatically mark the word *items* as a stopword; such expansions must be applied manually.

Stop words are respected when matching thesaurus entries to user queries. For example, suppose you define an equivalence between *Muhammad Ali* and *Cassius Clay* and also mark *M* as a stopword (it is not uncommon to mark all or most single letter words as stopwords). In this case, a query for *Cassius M. Clay* would match the thesaurus entry and return results for *Muhammad Ali* as expected.

Phrase search

A phrase search is a search query that contains one or more multi-word phrases enclosed in quotation marks. The words inside phrase-query terms are interpreted strictly literally and are not subject to stemming or thesaurus processing. For example, if you define a thesaurus equivalence between *Jennifer Lopez* and *JLo*, normal (unquoted) searches for *Jennifer Lopez* will also return results for *JLo*, but a quoted phrase search for *"Jennifer Lopez"* will not return the additional *JLo* results.

Relevance Ranking

It is typically desirable to return results for the actual user query ahead of results for stemming and/or thesaurus transformed versions of the query. This type of result ordering is supported by the Relevance Ranking modules. The module that is affected by thesaurus expansion and stemming is **Interp**. The module that is not affected by thesaurus and stemming is **Freq**.

Performance impact of Stemming and Thesaurus

Stemming and thesaurus equivalences generally add little or no time to data processing and indexing, and introduce little space overhead (beyond the space required to store the raw string forms of the equivalences).

In terms of online processing, both features will expand the set of results for typical user queries. While this generally slows search performance (search operations require an amount of time that grows linearly with the number of results), typically these additional results are a required part of the application behavior and cannot be avoided.

The overhead involved in matching the user query to thesaurus and stemming forms is generally low, but could slow performance in cases where a large thesaurus (tens of thousands of entries) is asked to process long search queries (dozens of terms). Typical applications exhibit neither extremely large thesauri nor very long user search queries.

Because matching for stemming entries is performed on a single-word basis, the cost for stemming-oriented query expansion does not grow with the size of the stemming database or with the length of the query. However, the stemming performance of a specific language is affected by the degree to which the language is inflected. For example, German words are much more inflected than English ones, and a query term can expand into a much larger set of compound words of which its stem is a component.

Chapter 7

Automatic Phrasing

This section describes how to implement the Automatic Phrasing feature of the Endeca MDEX Engine.

About Automatic Phrasing

When an application user provides individual search terms in a query, the Automatic Phrasing feature groups those individual terms into a search phrase and returns query results for the phrase.

Automatic Phrasing is similar to placing quotation marks around search terms before submitting them in a query. For example *"my search terms"* is the phrased version of the query *my search terms*. However, Automatic Phrasing removes the need for application users to place quotation marks around search phrases to get phrased results.

The result of Automatic Phrasing is that a Web application can process a more restricted query and therefore return fewer and more focused search results. This feature is available only for record search.

The Automatic Phrasing feature works by:

1. Comparing individual search terms in a query to a list of application-specific search phrases. The list of search phrases are stored in a project's phrase dictionary.
2. Grouping the search terms into search phrases.
3. Returning query results that are either based on the automatically-phrased query, or returning results based on the original unphrased query along with automatically-phrased Did You Mean (DYM) alternatives.

Implementation scenarios

Step 3 above suggests the two typical implementation scenarios to choose from when using Automatic Phrasing:

- Process an automatically-phrased form of the query and suggest the original unphrased query as a DYM alternative.

  In this scenario, the Automatic Phrasing feature rewrites the original query's search terms into a phrased query before processing it. If you are also using DYM, you can display the unphrased alternative so the user can opt-out of Automatic Phrasing and select their original query, if desired.

  For example, an application user searches a wine catalog for the unquoted terms *low tannin*. The MDEX Engine compares the search terms against the phrase dictionary, finds a phrase entry for "low tannin", and processes the phrased query as *"low tannin"*. The MDEX Engine returns 3 records for the phrased query *"low tannin"* rather than 16 records for the user's original unphrased query

low tannin. However, the Web application also presents a "Did you mean low tannin?" option, so the user may opt-out of Automatic Phrasing, if desired.

- Process the original query and suggest an automatically-phrased form of the query as a DYM alternative.

  In this scenario, the Automatic Phrasing feature processes the unphrased query as entered and determines if a phrased form of the query exists. If a phrased form is available, the Web application displays an automatically-phrased alternative as a Did You Mean option. The user can opt-in to Automatic Phrasing, if desired.

  For example, an application user searches a wine catalog for the unquoted terms *low tannin.* The MDEX Engine returns 16 records for the user's unphrased query *low tannin.* The Web application also presents a *Did you mean "low tannin"?* option so the user may opt-in to Automatic Phrasing, if desired.

Tasks for implementation

There are two tasks to implement Automatic Phrasing:

- Add phrases to your project using Developer Studio.
- Add Presentation API code to your Web application to support either of the two implementation scenarios described above.

Using Automatic Phrasing with Spelling Correction and DYM

You should enable the MDEX Engine for both Spelling Correction and Did You Mean.

If you want spelling corrected automatic phrases, the Spelling Correction feature ensures search terms are corrected *before* the terms are automatically phrased. The DYM feature provides users the choice to opt-in or opt-out of Automatic Phrasing.

The Endeca MDEX Engine applies spelling correction to a query before automatically phrasing the terms. This processing order means, for example, if a user misspells the query as *Napa Valle*, the MDEX Engine first spell corrects it to *Napa Valley* and then automatically phrases to *"Napa Valley"*. Without Spelling Correction enabled, Automatic Phrasing would typically not find a matching phrase in the phrase dictionary.

If you implement Automatic Phrasing to rewrite the query using an automatic phrase, then enabling DYM allows users a way to opt-out of Automatic Phrasing if they want to. On the other hand, if you implement Automatic Phrasing to process the original query and suggest automatically-phrased alternatives, then enabling DYM allows users to take advantage of automatically-phrased alternatives as follow-up queries.

Automatic Phrasing and query expansion

Once individual search terms in a query are grouped as a phrase, the phrase is not subject to thesaurus expansion or stemming by the MDEX Engine.

Adding phrases to a project

This section describes the two methods of adding phrases to your project.

There are two ways to include phrases in your Developer Studio project:

- Import phrases from an XML file.
- Choose dimension names and extract phrases from the dimension values.

After you add phrases and update your instance configuration, the MDEX Engine builds the phrase dictionary. You cannot view the phrases in Developer Studio. However, after adding phrases and saving your project, you can examine the phrases contained in a project's phrase dictionary by using a text editor to open the `phrases.xml` project file. Directly modifying `phrases.xml` is not supported.

Importing phrases from an XML file

You import an XML file of phrases using the Import Phrases dialog box in Developer Studio.

The import phrases XML file must conform to `phrase_import.dtd`, found in the Endeca MDEX Engine `conf/dtd` directory. Here is a simple example of a phrase file that conforms to `phrase_import.dtd`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE PHRASE_IMPORT SYSTEM "phrase_import.dtd">
<PHRASE_IMPORT>
    <PHRASE>Napa Valley</PHRASE>
    <PHRASE>low tannin</PHRASE>
</PHRASE_IMPORT>
```

To import phrases from an XML file:

1. Create the phrases XML file, using the format in the example above. You can create the file in any way you like. For example, you can type phrases into the file using an XML editor, or you can perform an XSLT transform on a phrase file in another format, and so on.

   To maintain naming consistency with other Endeca project files and their corresponding DTD files, you may choose to name your file `phrase_import.xml`.

2. Open your project in Developer Studio.
3. In the Project Explorer, expand **Search Configuration**.
4. Double-click **Automatic Phrasing** to display the Automatic Phrasing editor.
5. Click the **Import Phrases...** button.
6. In the Import Phrases dialog box, either type the path to your phrases file or click the **Browse** button to locate the file.
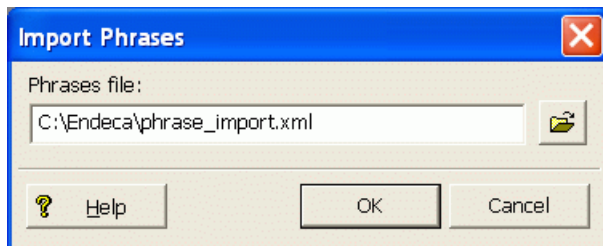
7. Click **OK** on the Import Phrases dialog box.
   The Messages pane displays the number of phrases read in from the XML file.
8. Click **OK** on the Automatic Phrasing dialog box.

9. Select **Save** from the File menu.

The project's `phrases.xml` configuration file is updated with the new phrases.

Keep in mind that if you import a newer version of an `import_phrases.xml` file, the most recent import overwrites phrases from any previous import. All phrases you want to import should be contained in a single XML file.

Extracting phrases from dimension names

Using Developer Studio, you can add phrases to your project based on the dimension values of any dimension you choose.

The MDEX Engine adds each multi-term dimension value in a selected dimension to the phrase dictionary. Single-term dimension values are not included. For example, if you import a WineType dimension from a wine catalog, the MDEX Engine creates a phrase entry for multi-term names such as "Pinot Noir" but not for single-term names such as "Merlot".

To extract phrases from dimension names:

1. Open your project in Developer Studio.
2. In the Project Explorer, expand **Search Configuration**.
3. Double-click **Automatic Phrasing** to display the Automatic Phrasing editor.
4. Select a dimension from the **All dimensions** panel and add it to the Selected dimensions panel by clicking **Add**. The editor should look like this example:
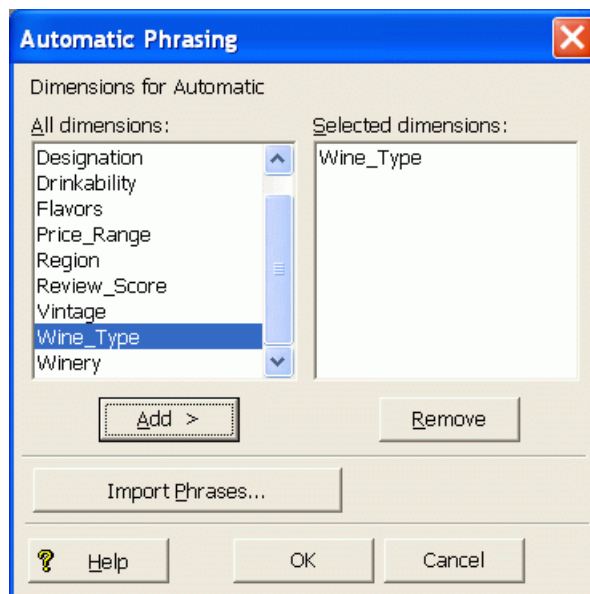
5. If desired, repeat step 4 to add more dimensions.
6. Click **OK** on the Automatic Phrasing dialog box.
7. Select **Save** from the File menu.

The project's `phrases.xml` configuration file is updated with the dimension names. Note that imported phrases are not overwritten by this procedure.

Adding search characters

If you have phrases that include punctuation, add those punctuation marks as search characters.

Adding the punctuation marks ensures that the MDEX Engine includes the punctuation when tokenizing the query, and therefore the MDEX Engine can match search terms with punctuation to phrases with punctuation. For details on using search characters, see the *MDEX Engine Basic Development Guide*.

For example, suppose you add phrases based on a Winery dimension, and consequently the Winery name "Anderson & Brothers" exists in your phrase dictionary. You should create a search character for the ampersand (&).

Presentation API development for Automatic Phrasing

The ENEQuery class has calls that handle Automatic Phrasing.

The Automatic Phrasing feature requires that the MDEX Engine compute whether an automatic phrase is available for a particular query's search terms.

The MDEX Engine computes the available phrases when setting the Java `setNavERecSearchComputeAlternativePhrasings()` method and the .NET `NavERecSearchComputeAlternativePhrasings` property to `true` in the `ENEQuery` object.

You can then optionally submit the phrased query to the MDEX Engine, instead of the user's original query, by calling the Java `setNavERecSearchRewriteQueryToAnAlternativePhrasing()` method or the .NET `NavERecSearchRewriteQueryToAnAlternativePhrasing` property with a value of `true`.

You can also call these methods by sending the necessary URL query parameters to the MDEX Engine via the `URLENEQuery` class, as shown in the next section.

When the MDEX Engine returns query results, your Web application displays whether the results were spell corrected, automatically phrased, or have DYM alternatives. Each of these Web application tasks are described in the sections below.

URL query parameters for Automatic Phrasing

Automatic Phrasing has two associated URL query parameters: `Ntpc` and `Ntpr`.

Both `Ntpc` and `Ntpr` are Boolean parameters that are enabled by setting to 1 and disabled by setting to 0.

The Ntpc parameter

Adding the `Ntpc=1` parameter instructs the MDEX Engine to compute phrasing alternatives for a query. Using this parameter alone, the MDEX Engine processes the original query and not any of the automatic phrasings computed by the MDEX Engine.

Here is an example URL that processes a user's query *napa valley* without phrasing and provides an alternative automatic phrasing, *Did you mean "napa valley"?*:

```
<application>?N=0&Ntk=All&Ntt=napa%20valley&Nty=1&Ntpc=1
```

If you omit `Ntpc=1` or set `Ntpc=0`, then automatic phrasing is disabled.

The Ntpr parameter

The `Ntpr` parameter instructs the MDEX Engine to rewrite the query using the available automatic phrase computed by `Ntpc`. The `Ntpr` parameter depends on the presence of `Ntpc=1`.

Here is an example URL that automatically phrases the user's query *napa valley* to *"napa valley"* and processes the phrased query. The Web application may also provide an unphrased alternative, so users can submit their original unphrased query (for example, *"Did you mean napa valley?"*):

```
<application>?N=0&Ntk=All&Ntt=napa%20valley&Nty=1&Ntpc=1&Ntpr=1
```

If you omit `Ntpr=1` or set `Ntpr=0`, then the query is not re-written using an automatic phrasing alternative. You can omit `Ntpr=1` and still use the `Ntpc=1` parameter to compute an available alternative for display as a DYM option.

Displaying spell-corrected and auto-phrased messages

To display messages for spell-corrected and automatically-phrased queries, your Web application code should be similar to these examples.

Java example

```
// Get the Map of lists of ESearchReport objects
Map recSrchRpts = nav.getESearchReportsComplete();
if (recSrchRpts.size() > 0) {
   // Get the user's search key
   String searchKey = request.getParameter("Ntk");
   if (searchKey != null) {
     if (recSrchRpts.containsKey(searchKey)) {
        // Get the ERecSearchReports for the search key
        List srchRptList = (List)recSrchRpts.get(searchKey);

        // for each report, display appropriate info
        for (Iterator i = srchRptList.Iterator(); i.hasNext();) {
           ESearchReport srchRpt = (ESearchReport)i.next();

           // Get the List of auto-correct values
             List autoCorrectList = searchReport.getAutoSuggestions();

           // If the list contains Auto Suggestion objects,
           // print the value of the first corrected term
           if (autoCorrectList.size() > 0) {
              // Get the Auto Suggestion object
              ESearchAutoSuggestion autoSug =
                   (ESearchAutoSuggestion)autoCorrectList.get(0);

              // Display appropriate autocorrect message
              if (autoSug.didSuggestionIncludeSpellingCorrection() &&
                  !autoSug.didSuggestionIncludeAutomaticPhrasing()) {
                     %>Spelling corrected to <%= autoSug.getTerms() %> <%
              }
              else if(autoSug.didSuggestionIncludeSpellingCorrection() &&
                  autoSug.didSuggestionIncludeAutomaticPhrasing()) {
                     %>Spelling corrected and then phrased
                     to <%= autoSug.getTerms() %> <%
              }
              else if(!autoSug.didSuggestionIncludeSpellingCorrection()&&
                   autoSug.didSuggestionIncludeAutomaticPhrasing()) {
                     %>Phrased to <%= autoSug.getTerms() %> <%
              }
```

```
            }
        }
    }
}
```

.NET example

```
// Get the Dictionary of lists of ESearchReport objects
IDictionary recSrchRpts = nav.ESearchReportsComplete;

// Get the user's search key
String searchKey = Request.QueryString["Ntk"];

if (searchKey != null) {
    if (recSrchRpts.Contains(searchKey)) {
        // Get the list of Search Report objects
        IList srchReportList = (IList)recSrchRpts[searchKey];

        // for each report, display appropriate info
        foreach (object ob in srchReportList) {
            ESearchReport searchReport = (ESearchReport)ob;

            // Get the List of auto correct objects
            IList autoCorrectList = searchReport.AutoSuggestions;

            // If the list contains auto correct objects,
            // print the value of the first corrected term
            if (autoCorrectList.Count > 0) {
                // Get the Auto Suggestion object
                ESearchAutoSuggestion autoSug =
                    (ESearchAutoSuggestion)autoCorrectList[0];

                // Display appropriate autocorrect message
                if (autoSug.GetDidSuggestionIncludeSpellingCorrection() &&
                    !autoSug.GetDidSuggestionIncludeAutomaticPhrasing()) {
                        %>Spelling corrected to <%= autoSug %> <%
                else if (autoSug.GetDidSuggestionIncludeSpellingCorrection()
 &&
                    autoSug.GetDidSuggestionIncludeAutomaticPhrasing()) {
                        %>Spelling corrected and phrased to
                        <%= autoSug.getTerms() %> <%
                }
              else if (!autoSug.GetDidSuggestionIncludeSpellingCorrection()&&

                    autoSug.GetDidSuggestionIncludeAutomaticPhrasing()) {
                        %>Phrased to <%= autoSug.getTerms() %> <%
                }
            }
        }
    }
}
```

Displaying DYM alternatives

To create a link for each Did You Mean alternative, your Web application code should look similar to these examples.

Note that it is important to display all the DYM alternatives (rather than just the first DYM alternative) because the user's desired query may not be the first alternative in the list of returned DYM options.

Java example

```java
// Get the Map of ESearchReport objects
Map dymRecSrchRpts = nav.getESearchReports();
  if (dymRecSrchRpts.size() > 0) {
  // Get the user's search key
  String searchKey = request.getParameter("Ntk");
  if (searchKey != null) {
    if (dymRecSrchRpts.containsKey(searchKey)) {
        // Get the List of ERecSearchReports for the user's search key
        List searchReportList = (List)dymRecSrchRpts.get(searchKey);

        // for each report, get the list of Did You Mean objects
        for (Iterator i = searchReportList.Iterator(); i.hasNext();) {
           ESearchReport searchReport = (ESearchReport)i.next();

           // Get the List of Did You Mean objects
           List dymList = searchReport.getDYMSuggestions();
           // Get all Did You Mean objects to display each available
           // DYM alternative.
           for (Iterator j = dymList.Iterator(); j.hasNext();) {
              ESearchDYMSuggestion dymSug =
                 (ESearchDYMSuggestion)j.next();
              String sug_val = dymSug.getTerms();
              String sug_num =
                String.valueOf(dymSug.getNumMatchingResults());
              String sug_sid = (String)request.getAttribute("sid");
              if (sug_val != null) {
                 ...
                 // Adjust URL parameters to create new search query
                 UrlGen urlg =
                    new UrlGen(request.getQueryString(), "UTF-8");
                 urlg.removeParam("Ntt");
                 urlg.addParam("Ntt", sug_val);
                 urlg.removeParam("Ntpc");
                 urlg.addParam("Ntpc", "1");
                 urlg.removeParam("Ntpr");
                 urlg.addParam("Ntpr", "0");
                 String url = CONTROLLER+"?"+urlg;
                 // Display Did You Mean link for each DYM alternative
                 %>Did You Mean <a href="<%=url%>">
                    <%= sug_val %></a><%
              }
           }
        }
     }
   }
 }
```

.NET example

```
// Get the Dictionary of ESearchReport objects
IDictionary dymRecSrchRpts = nav.ESearchReports;

// Get the user's search key
String dymSearchKey = Request.QueryString["Ntk"];
if (dymSearchKey != null) {
  if (dymRecSrchRpts.Contains(dymSearchKey)) {
```

```
      // Get the list of Search Report objects
      IList srchReportList = (IList)recSrchRpts[searchKey];

      // for each report, display all its DYM suggestions
      foreach (object srObj in srchReportList) {
          // Get the List of Did You Mean objects
          IList dymList = ((ESearchReport)srObj).DYMSuggestions;
          foreach (object dymObj in dymList) {
            ESearchDYMSuggestion dymSug = (ESearchDYMSuggestion)dymObj;
            String sug_val = dymSug.Terms;
            String sug_num = dymSug.NumMatchingResults.ToString();
            // Adjust URL parameters to create new search query
            UrlGen urlg =
              new UrlGen(Request.Url.Query.Substring(1),"UTF-8");
            urlg.RemoveParam("Ntt");
            urlg.AddParam("Ntt", sug_val);
            urlg.RemoveParam("Ntpc");
            urlg.AddParam("Ntpc", "1");
            urlg.RemoveParam("Ntpr");
            urlg.AddParam("Ntpr", "0");
            urlg.AddParam("sid",Request.QueryString["sid"]);
            String url = Application["CONTROLLER"].ToString()+"?"+urlg;
            // Display Did You Mean message and link
            // for each DYM option
            %>Did You Mean <a href="<%= url %>">
            <%= sug_val %></a>?<%
        }
      }
  }
}
```

Tips and troubleshooting for Automatic Phrasing

The following sections provide tips and troubleshooting guidance about using the Automatic Phrasing feature.

Examining how a phrased query was processed

If automatically-phrased query results are not what you expected, you can run the Dgraph with the `--wordinterp` flag to show how the MDEX Engine processed the query.

Single-word phrases

You can include a single word in your `phrases_import.xml` file and treat the word as a phrase in your project. This may be useful if you do not want stemming or thesaurus expansion applied to single-word query terms. You cannot include single word phrases by extracting them from dimension values using the Phrases dialog box. They have to be imported from your `phrases_import.xml` file.

Extending user phrases

The MDEX Engine does not extend phrases a user provides to match a phrase in the phrase dictionary. For example, if a user provides the query *A "BC" D* and "BCD" is in the phrase dictionary, the MDEX Engine does not extend the user's original phrasing of *"BC"* to "BCD".

Term order is significant in phrases

Phrases are matched only if search terms are provided in the same exact order and with the same exact terms as the phrase in the phrase dictionary. For example, if "weekend bag" is in the phrase dictionary, the MDEX Engine does not automatically phrase the search terms *weekend getaway bag* or *bag, weekend* to match *weekend bag*.

Possible dead ends

If an application automatically phrases search terms, it is possible a query may not produce results when it seemingly should have. Specifically, one way in which a dead-end query can occur is when a search phrase is displayed as a DYM link with results and navigation state filtering excludes the results.

For example, suppose a car sales application is set up to process a user's original query and display any automatic phrase alternatives as DYM options. Further suppose a user navigates to **Cars > Less than $15,000** and then provides the search terms *luxury package*. The search terms match the phrase "luxury package" in the phrase dictionary.

The user receives query results for **Cars > Less than $15,000** and results that matched some occurrences of the terms *luxury* and *package*. However, if the user clicks the *Did you mean "luxury package"?* link, then no results are available because the navigation state **Cars > Less than $15,000** excludes them. For details about how processing order affects queries, see the *MDEX Engine Basic Development Guide*.

Chapter 8

Stop Words

This section describes how to implement the Stop Words feature of the Endeca MDEX Engine.

About stop words

Stop words are words that are ignored by the Endeca MDEX Engine when the words are part of a keyword search. Typically, common words (like "the", "a", "as", etc.) are included in a stop word list and also terms that are very common in your data set.

For example, if your data set consists of lists of books, you might want to add the word "book" itself to the stop word list, because a search on that word would return an impractically large set of records. In addition, a stop word list can include the extraneous words contained in a typical question, allowing the query to focus on what the application user is really searching for.

Specifying stop words

There are two ways to specify stop words. You specify stop words manually in Developer Studio, and you can include one of the sample lists of stop words that are installed with the MDEX Engine.

Sample stop word lists

There is one sample list per language that the MDEX Engine supports. The sample lists are installed into `MDEX\<version>\olt\lang\stopword_samples` directory.

The MDEX Engine provides stop word files for the following languages: Catalan, Chinese (Simplified), Chinese (Traditional), Czech, Dutch, English, French, German, Greek, Hebrew, Hungarian, Italian, Japanese, Korean, Polish, Portuguese, Romanian, Russian, Spanish, Swedish, Thai, and Turkish. The language is identified in the file name by the `<language code>` value in `stop_words.<language code>.xml`.

Notes:

- Stop words are counted in any search mode that calculates results based on number of matching terms. However, the Endeca MDEX Engine reduces the minimum term match and maximum word omit requirement by the number of stop words contained in the query.
- Did You Mean can in some cases correct a word to one on the stop words list.
- The `--diacritic-folding` flag removes accent characters from stop words and prevents accented stop words from being returned in query results. For example, if `für` is a stop word, and

you specify the `--diacritic-folding` flag, then that flag treats the stop word as `fur`. Any queries that search for `fur` will not return results.

Adding a sample list of stop words to an application

The MDEX Engine installation includes sample lists of stop words for each language that the MDEX Engine supports. If desired, you can incorporate one sample list of stop words into your application. The sample list can provide either the full set of stop words for your application, or it can provide a starting point that you add to using Developer Studio.

If you are using Developer Studio to create stop words, be sure to copy the sample list into your project before manually adding any stop words. The copy operation replaces the stop words created using Developer Studio.

To add a sample list of stop words to an application:

1. Locate the `olt\lang\stopword_samples` directory in the MDEX Engine installation directory. For example, in a default Windows installation, this is `C:\Endeca\MDEX\6.4.0\olt\lang\stopword_samples`.
2. In the `stopword_samples` directory, locate a sample file for the language of the records in your application.
3. Copy the sample file to the `<app dir>\config\pipeline` directory.
4. Rename the file from `stop_words.<language code>.xml` to `<app prefix>.stop_words.xml`

   This step replaces the old stop words file.

5. Optionally, you can manually add stop words to the sample list using Developer Studio. To do so:
   a) Start Developer Studio.
   b) Open the Endeca project in the `<app dir>\config\pipeline` directory.
   c) Double click the **Stop Words** editor.
      (Developer Studio loads the sample list of stop words and displays them in the **Stop Words** editor.)
   d) Create additional stop words as necessary.
6. When you are adding the sample list of stop words, save and closet the project, then run a baseline update to process them using Dgidx.

Chapter 9

Relevance Ranking

This section describes the tasks involved in implementing the Relevance Ranking feature of the Endeca MDEX Engine.

About the Relevance Ranking feature

Relevance Ranking controls the order in which search results are displayed to the end user of an Endeca application.

You configure the Relevance Ranking feature to display the most important search results earliest to the user, because application users are often unwilling to page through large result sets.

Relevance ranking can be used to independently control the result ordering for both record search and dimension search queries. However, while relevance ranking for record search can be configured with Developer Studio, relevance ranking for dimension search cannot. (You assign relevance ranking for dimension search via the `RELRANK_STRATEGY` attribute of `dimsearch_config.xml`, or at query time by specifying the `Dx` and `Dk` parameters of the `UrlENEQuery`.)

The importance of a search result is generally an application-specific concept. The Relevance Ranking feature provides a flexible, configurable set of result ranking modules. These modules can be used in combinations (called ranking strategies) to produce a wide range of relevance ranking effects. Because Relevance Ranking is a complex and powerful feature, Endeca provides recommended strategies that you can use as a point of departure for further development. For details, see the "Recommended strategies" topic.

Related Links

   This section provides some recommended strategies that depend on the implementation type.

Relevance Ranking modules

Relevance Ranking modules are the building blocks from which you build the relevance ranking strategies that you actually apply to your search interfaces.

This section describes the available set of Relevance Ranking modules and their scoring behaviors.

Note: Some modules are listed in the Developer Studio interface by their abbreviated spellings, such as "Interp" for Interpreted.

Exact

The Exact module provides a finer grained (but more computationally expensive) alternative to the Phrase module.

The Exact module groups results into three strata based on how well they match the query string:

- The highest stratum contains results whose complete text matches the user's query exactly.
- The middle stratum contains results that contain the user's query as a subphrase.
- The lowest stratum contains other hits (such as normal conjunctive matches). Any match that would not be a match without query expansion lands in the lowest stratum. Also in this stratum are records that do not contain relevance ranking terms (such as those specified in the `Nrr` query parameter).

Note: The Exact module is computationally expensive, especially on large text fields. It is intended for use only on small text fields (such as dimension values or small property values like part IDs). This module should not be used with large or offline documents (such as FILE or ENCODED_FILE properties). Use of this module in these cases will result in very poor performance and/or application failures due to request timeouts. The Phrase module, with and without approximation turned on, does similar but less sophisticated ranking that can be used as a higher performance substitute.

Field

The Field module ranks documents based on the search interface field with the highest priority in which it matched.

Only the best field in which a match occurs is considered. The Field module is often used in relevance ranking strategies for catalog applications, because the category or product name is typically a good match. Field assigns a score to each result based on the static rank of the dimension or property member or members of the search interface that caused the document to match the query. In Developer Studio, static field ranks are assigned based on the order in which members of a search interface are listed in the Search Interfaces view. The first (left-most) member has the highest rank.

By default, matches caused by cross-field matching are assigned a score of zero. The score for cross-field matches can be set explicitly in Developer Studio by moving the `<<CROSS_FIELD>>` indicator up or down in the Selected Members list of the Search Interface editor. The `<<CROSS_FIELD>>` indicator is available only for search interfaces that have the Field module and are configured to support cross-field matches. All non-zero ranks must be non-equal and only their order matters.

For example, a search interface might contain both Title and DocumentContent properties, where hits on Title are considered more important than hits on DocumentContent (which in turn are considered more important than `<<CROSS_FIELD>>` matches). Such a ranking is implemented by assigning the highest rank to Title, the next highest rank to DocumentContent, and setting the `<<CROSS_FIELD>>` indicator at the bottom of the Selected Members list in the Search Interface editor.

Note: The Field module is only valid for record search operations. This module assigns a score of zero to all results for other types of search requests. In addition, Field treats all matches the same, whether or not they are due to query expansion.

First

Designed primarily for use with unstructured data, the First module ranks documents by how close the query terms are to the beginning of the document.

The First module groups its results into variably-sized strata. The strata are not the same size, because while the first word is probably more relevant than the tenth word, the 301st is probably not so much more relevant than the 310th word. This module takes advantage of the fact that the closer something is to the beginning of a document, the more likely it is to be relevant.

The First module works as follows:

- When the query has a single term, First's behavior is straight-forward: it retrieves the first absolute position of the word in the document, then calculates which stratum contains that position. The score for this document is based upon that stratum; earlier strata are better than later strata.
- When the query has multiple terms, First behaves as follows: The first absolute position for each of the query terms is determined, and then the median position of these positions is calculated. This median is treated as the position of this query in the document and can be used with stratification as described in the single word case.
- With query expansion (using stemming, spelling correction, or the thesaurus), the First module treats expanded terms as if they occurred in the source query. For example, the phrase *glucose intolerence* would be corrected to *glucose intolerance* (with *intolerence* spell-corrected to *intolerance*). First then continues as it does in the non-expansion case. The first position of each term is computed and the median of these is taken.
- In a partially matched query, where only some of the query terms cause a document to match, First behaves as if the intersection of terms that occur in the document and terms that occur in the original query were the entire query. For example, if the query *cat bird dog* is partially matched to a document on the terms *cat* and *bird*, then the document is scored as if the query were *cat bird*. If no terms match, then the document is scored in the lowest strata.
- The First relevance ranking module is supported for wildcard queries.

Note: The First module does not work with Boolean searches and cross-field matching. It assigns all such matches a score of zero.

Frequency

The Frequency (Freq) module provides result scoring based on the frequency (number of occurrences) of the user's query terms in the result text.

Results with more occurrences of the user search terms are considered more relevant.

The score produced by the Freq module for a result record is the sum of the frequencies of all user search terms in all fields (properties or dimensions in the search interface in question) that match a sufficient number of terms. The number of terms depends on the match mode, such as all terms in a MatchAll query, a sufficient number of terms in a MatchPartial query, and so on. Cross-field match records are assigned a score of zero. Total scores are capped at 1024; in other words, if the sum of frequencies of the user search terms in all matching fields is greater than or equal to 1024, the record gets a score of 1024 from the Freq module.

For example, suppose we have the following record:

```
{Title="test record", Abstract="this is a test", Text="one test this is"}
```

A MatchAll search for *test this* would cause Freq to assign a score of 4, since *this* and *test* occur a total of 4 times in the fields that match all search terms (Abstract and Text, in this case). The number of phrase occurrences (just one in the Text field) doesn't matter, only the sum of the individual word occurrences. Also note that the occurrence of *test* in the Title field does not contribute to the score, since that field did not match all of the terms.

A MatchAll search for *one record* would hit this record, assuming that cross field matching was enabled. But the record would get a score of zero from Freq, because no single field matches all of the terms. Freq ignores matches due to query expansion (that is, such matches are given a rank of 0).

Glom

The Glom module ranks single-field matches ahead of cross-field matches and also ahead of non-matches (records that do not contain the search term).

This module serves as a useful tie-breaker function in combination with the Maximum Field module. It is only useful in conjunction with record search operations. If you want a strategy that ranks single-field matches first, cross-field matches second, and no matches third, then use the Glom module followed by the Nterms (Number of Terms) module.

Note: Glom treats all matches the same, whether or not they are due to query expansion.

Glom interaction with search modes

The Glom module considers a single-field match to be one in which a single field has enough terms to satisfy the conditions of the match mode. For this reason, in MatchAny search mode, cross-field matches are impossible, because a single term is sufficient to create a match. Every match is considered to be a single-field match, even if there were several search terms.

For MatchPartial search mode, if the required number of matches is two, the Glom module considers a record to be a single-field match if it has at least one field that contains two or more or the search terms. You cannot rank results based on how many terms match within a single field.

For more information about search modes, see the *MDEX Engine Basic Development Guide*.

Interpreted

Interpreted (Interp) is a general-purpose module that assigns a score to each result record based on the query processing techniques used to obtain the match.

Matching techniques considered include partial matching, cross-attribute matching, spelling correction, thesaurus, and stemming matching.

Specifically, the Interpreted module ranks results as follows:

1. All non-partial matches are ranked ahead of all partial matches. For more information, see "Using Search Modes" in the *MDEX Engine Basic Development Guide*.
2. Within the above strata, all single-field matches are ranked ahead of all cross-field matches. For more information, see "About Search Interfaces" in the *MDEX Engine Basic Development Guide*.
3. Within the above strata, all non-spelling-corrected matches are ranked above all spelling-corrected matches. See the topic "Using Spelling Correction and Did You Mean" for more information.

4. Within the above strata, all thesaurus matches are ranked below all non-thesaurus matches. See the topic "Using Stemming and Thesaurus" for more information.
5. Within the above strata, all stemming matches are ranked below all non-stemming matches. See "Using Stemming and Thesaurus" for more information.

> **Note:** Because the Interpreted module comprises the matching techniques of the Spell, Glom, Stem, and Thesaurus modules, there is no need to add them to your strategy individually as well if you are using Interpreted.

Related Links

About Spelling Correction and Did You Mean on page 61
> The Spelling Correction and Did You Mean features of the Endeca MDEX Engine enable search queries to return expected results when the spelling used in query terms does not match the spelling used in the result text (that is, when the user misspells search terms).

Overview of Stemming and Thesaurus on page 81
> The Endeca MDEX Engine supports Stemming and Thesaurus features that allow keyword search queries to match text containing alternate forms of the query terms or phrases.

Maximum Field

The Maximum Field (Maxfield) module behaves identically to the Field module, except in how it scores cross-field matches.

Unlike Field, which assigns a static score to cross-field matches, Maximum Field selects the score of the highest-ranked field that contributed to the match.

Note the following:

- Because Maximum Field defines the score for cross-field matches dynamically, it does not make use of the `<<CROSS_FIELD>>` indicator set in the Search Interface editor.
- Maximum Field is only valid for record search operations. This module assigns a score of zero to all results for other types of search requests.
- Maximum Field treats all matches the same, whether or not they are due to query expansion.

Number of Fields

The Number of Fields (Numfields) module ranks results based on the number of fields in the associated search interface in which a match occurs.

Note that we are counting whole-field rather than cross-field matches. Therefore, a result that matches two fields matches each field completely, while a cross-field match typically does not match any field completely.

> **Note:** Numfields treats all matches the same, whether or not they are due to query expansion. The Numfields module is only useful in conjunction with record search operations.

Number of Terms

The Number of Terms (or Nterms) module ranks matches according to how many query terms they match.

For example, in a three-word query, results that match all three words will be ranked above results that match only two, which will be ranked above results that match only one, which will be ranked above results that had no matches.

Note the following:

- The Nterms module is only applicable to search modes where results can vary in how many query terms they match. These include MatchAny, MatchPartial, MatchAllAny, and MatchAllPartial. For details on these search modes, see the *MDEX Engine Basic Development Guide* .
- Nterms treats all matches the same, whether or not they are due to query expansion.

Phrase

The Phrase module states that results containing the user's query as an exact phrase, or a subset of the exact phrase, should be considered more relevant than matches simply containing the user's search terms scattered throughout the text.

Records that have the phrase are ranked higher than records which do not contain the phrase.
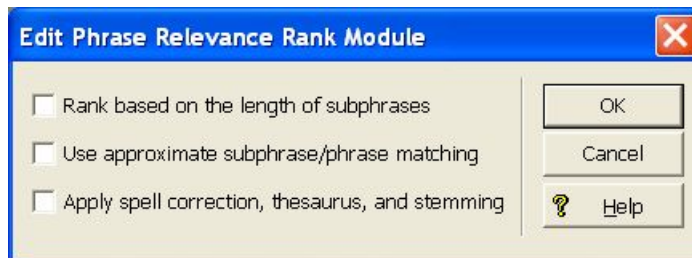
Configuring the Phrase module

The Phrase module has a variety of options that you use to customize its behavior.

The Phrase options are:

- Rank based on length of subphrases
- Use approximate subphrase/phrase matching
- Apply spell correction, thesaurus, and stemming

When you add the Phrase module in the Relevance Ranking Modules editor, you are presented with the following editor that allows you to set these options.

Ranking based on length of subphrases
When you configure the Phrase module, you have the option of enabling subphrasing.

Subphrasing ranks results based on the length of their subphrase matches. In other words, results that match three terms are considered more relevant than results that match two terms, and so on.

A subphrase is defined as a contiguous subset of the query terms the user entered, in the order that he or she entered them. For example, the query "fax cover sheets" contains the subphrases "fax", "cover", "sheets", "fax cover", "cover sheets", and "fax cover sheets", but not "fax sheets".

Content contained inside nested quotes in a phrase is treated as one term. For example, consider the following phrase:

```
the question is "to be or not to be"
```

The quoted text ("to be or not to be") is treated as one query term, so this example consists of four query terms even though it has a total of nine words.

When subphrasing is not enabled, results are ranked into two strata: those that matched the entire phrase and those that did not.

Using approximate matching
Approximate matching provides higher-performance matching, as compared to the standard Phrase module, with somewhat less exact results.

With approximate matching enabled, the Phrase module looks at a limited number of positions in each result that a phrase match could possibly exist, rather than all the positions. Only this limited number of possible occurrences is considered, regardless of whether there are later occurrences that are better, more relevant matches.

The approximate setting is appropriate in cases where the runtime performance of the standard Phrase module is inadequate because of large result contents and/or high site load.

Applying spelling correction, thesaurus, and stemming
Applying spelling correction, thesaurus, and stemming adjustments to the original phrase is generically known as query expansion.

With query expansion enabled, the Phrase module ranks results that match a phrase's expanded forms in the same stratum as results that match the original phrase.

Consider the following example:

- A thesaurus entry exists that expands "US" to "United States".
- The user queries for "US government".

The query "US government" is expanded to "United States government" for matching purposes, but the Phrase module gives a score of two to any results matching "United States government" because the original, unexpanded version of the query, "US government", only had two terms.

Summary of Phrase option interactions
The three configuration settings for the Phrase module can be used in a variety of combinations for different effects.

The following matrix describes the behavior of each combination.

Subphrase	Approximate	Expansion	Description
Off	Off	Off	Default. Ranks results into two strata: those that match the user's query as a whole phrase, and those that do not.
Off	Off	On	Ranks results into two strata: those that match the original, or an extended version, of the query as a whole phrase, and those that do not.
Off	On	Off	Ranks results into two strata: those that match the original query as a whole phrase, and those that do not. Look only at the first possible phrase match within each record.
Off	On	On	Ranks results into two strata: those that match the original, or an extended version, of the query as a whole phrase, and those that do not. Look only at the first possible phrase match within each record.
On	Off	Off	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase.
On	Off	On	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched

Subphrase	Approximate	Expansion	Description
			subphrase. Extend subphrases to facilitate matching but rank based on the length of the original subphrase (before extension).
			Note: This combination can have a negative performance impact on query throughput.
On	On	Off	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase. Look only at the first possible phrase match within each record.
On	On	On	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase. Expand the query to facilitate matching but rank based on the length of the original subphrase (before extension). Look only at the first possible phrase match within each record.

Note: You should only use one Phrase module in any given search interface and set all of your options in it.

Effect of search modes on Phrase behavior

Endeca provides a variety of search modes to facilitate matching during search (MatchAny, MatchAll, MatchPartial, and so on).

These modes only determine which results match a user's query, they have no effect on how the results are ranked after the matches have been found. Therefore, the Phrase module works as described in this section, regardless of search mode. The one exception to this rule is MatchBoolean. Phrase, like the other relevance ranking modules, is never applied to the results of MatchBoolean queries.

Results with multiple matches

If a single result has multiple subphrase matches, either within the same field or in several different fields, the result is slotted into a stratum based on the length of the longest subphrase match.

Stop words and Phrase behavior

When using the Phrase module, stop words are always treated like non-stop word terms and stratified accordingly.

For example, the query "raining cats and dogs" will result in a rank of two for a result containing "fat cats and hungry dogs" and a rank of three for a result containing "fat cats and dogs" (this example assumes subphrase is enabled).

Cross-field matches and Phrase behavior

An entire phrase, or subphrase, must appear in a single field in order for it to be considered a match.

(In other words, matches created by concatenating fields are not considered by the Phrase module.)

Treatment of wildcards with the Phrase module

The Phrase module translates each wildcard in a query into a generic placeholder for a single term.

For example, the query "sparkling w* wine" becomes "sparkling * wine" during phrase relevance ranking, where "*" indicates a single term. This generic wildcard replacement causes slightly different behavior depending on whether subphrasing is enabled.

When subphrasing is not enabled, all results that match the generic version of the wildcard phrase exactly are still placed into the first stratum. It is important, however, to understand what constitutes a matching result from the Phrase module's point of view.

Consider the search query "sparkling w* wine" with the MatchAny mode enabled. In MatchAny mode, search results only need to contain one of the requested terms to be valid, so a list of search results for this query could contain phrases that look like this:

```
sparkling white wine
sparkling refreshing wine
sparkling wet wine
sparkling soda
wine cooler
```

When phrase relevance ranking is applied to these search results, the Phrase module looks for matches to "sparkling * wine" not "sparkling w* wine." Therefore, there are three results—"sparkling white wine," "sparkling refreshing wine," and "sparkling wet wine"—that are considered phrase matches for the purposes of ranking. These results are placed in the first stratum. The other two results are placed in the second stratum.

When subphrasing is enabled, the behavior becomes a bit more complex. Again, we have to remember that wildcards become generic placeholders and match any single term in a result. This means that any subphrase that is adjacent to a wildcard will, by definition, match at least one additional term (the wildcard). Because of this behavior, subphrases break down differently. The subphrases for "cold sparkling w* wine" break down into the following (note that w* changes to *):

```
cold
sparkling *
* wine
cold sparkling *
sparkling * wine
cold sparkling * wine
```

Notice that the subphrases "sparkling," "wine," and "cold sparkling" are not included in this list. Because these subphrases are adjacent to the wildcard, we know that the subphrases will match at least one additional term. Therefore, these subphrases are subsumed by the "sparkling *", "* wine", and "cold sparkling *" subphrases.

Like regular subphrase, stratification is based on the number of terms in the subphrase, and the wildcard placeholders are counted toward the length of the subphrase. To continue the example above, results that contain "cold" get a score of one, results that contain "sparkling *" get a score of two, and so on. Again, this is the case even if the matching result phrases are different, for example, "sparkling white" and "sparkling soda."

Finally, it is important to note that, while the wildcard can be replaced by any term, a term must still exist. In other words, search results that contain the phrase "sparkling wine" are not acceptable matches for the phrase "sparkling * wine" because there is no term to substitute for the wildcard. Conversely, the phrase "sparkling cold white wine" is also not a match because each wildcard can be replaced by one, and only one, term. Even when wildcards are present, results must contain the correct number of terms, in the correct order, for them to be considered phrase matches by the Phrase module.

Notes about the Phrase module

Keep the following points in mind when using the Phrase module.

* If a query contains only one word, then that word constitutes the entire phrase and all of the matching results will be put into one stratum (score = 1). However, the module can rank the results into two strata: one for records that contain the phrase and a lower-ranking stratum for records that do not contain the phrase.
* Because of the way hyphenated words are positionally indexed, Oracle recommends that you enable subphrase if your results contain hyphenated words.

Proximity

Designed primarily for use with unstructured data, the Proximity module ranks how close the query terms are to each other in a document by counting the number of intervening words.

Like the First module, this module groups its results into variable sized strata, because the difference in significance of an interval of one word and one of two words is usually greater than the difference in significance of an interval of 21 words and 22. If no terms match, the document is placed in the lowest stratum.

Single words and phrases get assigned to the best stratum because there are no intervening words. When the query has multiple terms, Proximity behaves as follows:

1. All of the absolute positions for each of the query terms are computed.
2. The smallest range that includes at least one instance of each of the query terms is calculated. This range's length is given in number of words. The score for each document is the strata that contains the difference of the range's length and the number of terms in the query; smaller differences are better than larger differences.

Under query expansion (that is, stemming, spelling correction, and the thesaurus), the expanded terms are treated as if they were in the query, so the proximity metric is computed using the locations of the expanded terms in the matching document.

For example, if a user searches for *big cats* and a document contains the sentence, "Big Bird likes his cat" (stemming takes *cats* to *cat*), then the proximity metric is computed just as if the sentence were, "Big Bird likes his cats."

Proximity scores partially matched queries as if the query only contained the matching terms. For example, if a user searches for *cat dog fish* and a document is partially matched that contains only *cat* and *fish*, then the document is scored as if the query *cat fish* had been entered.

> **Note:** Proximity does not work with Boolean searches, cross-field matching, or wildcard search. It assigns all such matches a score of zero.

Spell

The Spell module ranks spelling-corrected matches below other kinds of matches.

Spell assigns a rank of 0 to matches from spelling correction, and a rank of 1 from all other sources. That is, it ignores all other sorts of query expansion.

Static

The Static module assigns a static or constant data-specific value to each search result, depending on the type of search operation performed and depending on optional parameters that can be passed to the module.

For record search operations, the first parameter to the module specifies a property, which will define the sort order assigned by the module. The second parameter can be specified as ascending or descending to indicate the sort order to use for the specified property.

For example, using the module `Static(Availability,descending)` would sort result records in descending order with respect to their assignments from the Availability property. Using the module `Static(Title,ascending)` would sort result records in ascending order by their Title property assignments.

In a catalog application, setting the static module by Price, descending leads to more expensive products being displayed first.

For dimension search, the first parameter can be specified as `nbins`, `depth`, or `rank`:

- Specifying `nbins` causes the static module to sort result dimension values by the number of associated records in the full data set.
- Specifying `depth` causes the static module to sort result dimension values by their depth in the dimension hierarchy.
- Specifying `rank` causes dimension values to be sorted by the ranks assigned to them for the application.

Stratify

The Stratify module is used to boost or bury records in the result set.

The Stratify module takes one or more EQL (Endeca Query Language) expressions and groups results into strata, based on how well they match the record search (with the `Ntx` parameter). Records are placed in the stratum associated with the first EQL expression they match. The first stratum is the highest ranked, the next stratum is next-highest ranked, and so forth. If an asterisk is specified instead of an EQL expression, unmatched records are placed in the corresponding stratum.

The Stratify module can also be used for record boost and bury sort operations. In this usage, you must specify `Endeca.stratify` as the name for the `Ns` parameter.

The Stratify module is the basic component of the record boost and bury feature, which is described in the *MDEX Engine Basic Development Guide*.

Stem

The Stem module ranks matches due to stemming below other kinds of matches.

Stem assigns a rank of 0 to matches from stemming, and a rank of 1 from all other sources. That is, it ignores all other sorts of query expansion.

Thesaurus

The Thesaurus module ranks matches due to thesaurus entries below other sorts of matches.

Thesaurus assigns a rank of 0 to matches from the thesaurus, and a rank of 1 from all other sources. That is, it ignores all other sorts of query expansion.

Weighted Frequency

Like the Frequency module, the Weighted Frequency (Wfreq) module scores results based on the frequency of user query terms in the result.

Additionally, the Weighted Frequency module weights the individual query term frequencies for each result by the information content (overall frequency in the complete data set) of each query term. Less frequent query terms (that is, terms that would result in fewer search results) are weighted more heavily than more frequently occurring terms.

Note: The Weighted Frequency module ignores matches due to query expansion (that is, such matches are given a rank of 0).

Relevance Ranking strategies

Relevance Ranking modules define the primitive search result ordering functions provided by the MDEX Engine. These primitive modules can be combined to compose more complex ordering behaviors called Relevance Ranking strategies.

You may also define and apply a strategy that consists of a single module, rather than a group of modules.

A Relevance Ranking strategy is essentially an ordered list of relevance ranking modules and (in a URL relevance ranking string) references to other relevance ranking strategies. The scores assigned by a strategy are composed from the scores assigned by its constituent modules. This composite score is constructed so that records are first ordered by the first module. After that, ties are broken by the subsequent modules in order. If any ties remain after all modules have run, the ties are resolved by the default sort. If after that any ties still remain, the order of records is determined by the system.

Relevance Ranking strategies are used in two main contexts in the MDEX Engine:

- In Developer Studio, you apply Relevance Ranking to a search interface via the Search Interface editor and the Relevance Ranking Modules editor, both of which are documented in Developer Studio online help.
- At the MDEX Engine query level, Relevance Ranking strategies can be selected to override the default specified for the selected search interface. This allows Relevance Ranking behavior to be fully customized on a per-query basis. For details, see the "URL query parameters for relevance ranking" topic.

Implementing relevance ranking

Developer Studio allows you to create and control relevance ranking for record search.

You can apply record search relevance ranking as you are creating a search interface, or afterwards. A search interface is a named group of at least one dimension and/or property. You create search interfaces so you can apply behavior like relevance ranking across a group. For more information about search interfaces, see "About Search Interfaces" in the *MDEX Engine Basic Development Guide*.

Adding a Static module

Keep the following in mind when you add a Static module to the ranking strategy.

The Static module is the only one that you can add multiple times. The interface prevents the addition of multiple instances of the other modules. In addition, adding a Static module launches the Edit Static Relevance Rank Module editor. Use this editor to add the required parameters (dimension or property name and sort order).

Ranking order for Field and Maximum Field modules

The Field and Maximum Field modules ranks results based on which member property or dimension of the selected search interface caused the match.

Higher relevance-ranked values correspond to greater importance. This behavior means that the Field and Maximum Field modules will score results caused by higher-ranked properties and dimensions ahead of those caused by lower-ranked properties and dimensions.

To change the relevance ranking behavior for these modules, you would move the search interface members to the appropriate position in the Search Interface editor's Selected Members list, using the Up and Down arrows.

Cross-field matching for the Field module

For search interfaces that allow cross-field matches and have a Field module, you can configure the static score assigned to cross-field matches by the Field module on an individual search interface.

You might do this if you considered cross-field matches better than description-only matches.

Such a search interface would appear similar to this example in the Search Interface editor:
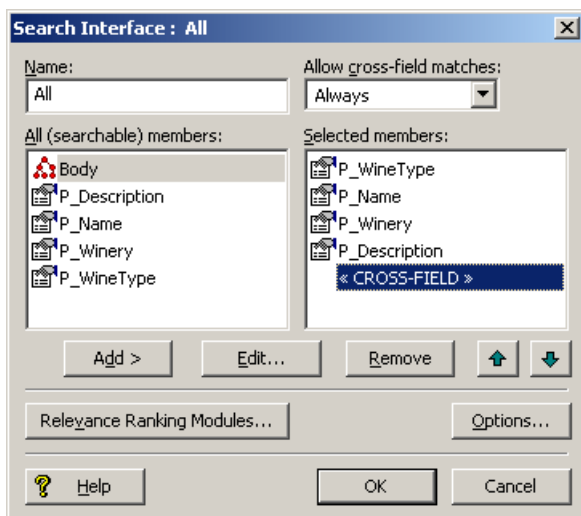
In the example, note the presence of the `<<CROSS-FIELD>>` indicator in the Selected Members list. This indicator is present only in search interfaces with Always or On Failure cross-field matches and a ranking strategy that includes a Field module.

How relevance ranking score ties between search interfaces are resolved

In the case of multiple search interfaces and relevance ranking score ties, ties are broken based on the relevance ranking sort strategy of the search interface with the highest relevance ranking score for a given record.

If two different records belong to different search interfaces, the record from the search interface specified earlier in the query comes first.

Implementing relevance ranking strategies for dimension search

There is no MDEX Engine configuration necessary to configure a relevance ranking strategy for record search.

To define the relevance ranking strategy for dimension search operations, modify the RELRANK_STRATEGY attribute of `dimsearch_config.xml`. This attribute specifies the name of a relevance ranking strategy for dimension search. The content of this attribute should be a relevance ranking string, as in the following examples:

```
exact,static(rank,descending)
interp,exact
```

For details on the format of the relevance ranking string, see the "URL query parameters for relevance ranking" topic.

The default ranking strategy for dimension search operations, which is applied if you do not make any changes to it, is:

```
interp,exact,static
```

The default ranking strategy for record search operations is no strategy. That is, unless you explicitly establish a relevance ranking strategy, none is used.

Related Links

URL query parameters for relevance ranking on page 121
>   URL query parameters allow you to communicate with the Presentation API from your client browser.

Retrieving the relevance ranking for records

The Dgraph `--stat-brel` flag creates a dynamic property on each record named `DGraph.BinRel¬evanceRank`. The value of this property reflects the relevance rank assigned to a record in full text search.

The Java `ERec.getProperties()` method and the .NET `ERec.Properties` property return a list of properties (`PropertyMap` object) associated with the record. At this point, calling the Java `PropertyMap.get()` method or the .NET `PropertyMap` object with the `DGraph.BinRelevanceR¬ank` argument returns the value of the property.

The following code samples show how to retrieve the `DGraph.BinRelevanceRank` for a given record.

Java example

```java
// get the record list from the navigation object
ERecList recs = nav.getERecs();
// loop over record list
for (int i=0; i<recs.size(); i++) {
```

```
 // get individual record
 ERec rec = (ERec)recs.get(i);
 // get property map for record
 PropertyMap propsMap = rec.getProperties();
 // Check for a non-null relevance rank property
 if (propsMap.get("DGraph.BinRelevanceRank") != null) {
  String rankNum =
     (String)propsMap.get("DGraph.BinRelevanceRank");
  %>Relevance ranking for this record:
  <%= rankNum %>
  <%
 } // end of if
} // end of for loop iteration
```

.NET example

```
// get the record list from the navigation object
ERecList recs = nav.ERecs;
// loop over record list
for (int i=0; i<recs.Count; i++) {
 // get individual record
 ERec rec = (ERec)recs[i];
 // get property map for record
 PropertyMap propsMap = rec.Properties;
 // Check for a non-null relevance rank property
 String rankNum = "";
 if (propsMap["DGraph.BinRelevanceRank"] != null) {
  rankNum = (String)propsMap["DGraph.BinRelevanceRank"];
  %>Relevance ranking for this record:
  <%= rankNum %>
  <%
 } // end of if
} // end of for loop iteration
```

Interpreting the values of `DGraph.BinRelevanceRank`

The MDEX Engine sorts records for relevance ranking using a more granular algorithm than the number you retrieve with `DGraph.BinRelevanceRank`.

If, for example, you need to interpret the values of the `DGraph.BinRelevanceRank` property for two different records, it is helpful to know that while these values roughly represent the sorting used for relevance-ranked records, they are not as precise as the internal sorting numbers the MDEX Engine actually uses to sort the records.

For example, you may see the same `DGraph.BinRelevanceRank` value for two records that are sorted slightly differently. When interpreting the results of `DGraph.BinRelevanceRank` for two different records, consider these values as providing rough guidance only on whether one record has a significantly higher relevance rank than the other. However, if the value of `DGraph.BinRelevanceR¬ank` is the same, this does not mean that the records are sorted the same, since the underlying sorting mechanism in the MDEX Engine is more precise. It is important to note that the MDEX Engine always returns consistent results and consistently interprets tie breaks in sorting, if they occur.

Controlling relevance ranking at the query level

At the MDEX Engine query level, relevance ranking strategies can be selected to override the default specified for the selected search interface.

This allows relevance ranking behavior to be fully customized on a per-query basis. MDEX Engine URL relevance ranking strategy strings must contain one or more relevance ranking module names. Module names can be any of these pre-defined modules:

- exact
- field (useful for record search only)
- first
- freq
- glom (useful for record search only)
- interp
- maxfield (useful for record search only)
- nterms
- numfields (useful for record search only)
- phrase (for details on using phrase, see the section below)
- proximity
- spell
- stem
- thesaurus
- static (for details on using static, see the section below)
- wfreq

Module names are delimited by comma (,) characters. No other stray characters (such as spaces) are allowed. Module names are listed in descending order of priority.

Exact module, First module, Nterms module, and Proximity module details

The Exact, First, Nterms, and Proximity modules can take one parameter named `considerField¬Ranks`. If specified, the `considerFieldRanks` parameter indicates that the module should further sort records according to field ranking scores, after the records have been sorted according to the standard behavior of the module.

For example, if you specify exact without the parameter in a query, records that are an exact match are sorted into a strata that is higher than non-exact matches. Within each strata, the records are only sorted according to the default sort order or a specified sort key.

If you add the `considerFieldRanks` parameter to URL query syntax and specify `exact(consid¬erFieldRanks)`, the records within each strata are sorted so that those with higher field ranking scores are more relevant than those with lower field ranking scores within the same strata.

Freq module and Numfields module details

The Freq module and also the Numfields module can take one parameter named `considerField¬Ranks`. If specified, the `considerFieldRanks` parameter indicates that the module should further sort records according to ranking scores that are calculated across multiple fields, after the records have been sorted according to the standard behavior of the module. For these modules, cross-field matches are weighted such that matches in higher ranked fields contribute more than matches in lower ranked fields.

Phrase module details

The Phrase module can take up to four parameters:

- `approximate` - enables approximate matching.
- `considerFieldRanks` - enables further sorting according to the field rank score of the match. If specified, the `considerFieldRanks` parameter indicates that the module should further sort records according to field ranking scores, after the records have been sorted according to the standard behavior of the module.
- `query_expansion` - enables query expansion.
- `subphrase` - enables subphrase matching

The presence of a parameter indicates that the feature should be enabled, and the parameters can be in any order. For example: `phrase(subphrase,approximate,query_expansion)`

Static module details

The Static module takes two parameters. For record search, the first parameter is a property or dimension to use for assigning static scores (based on sort order) and the second is the sort order: ascending (ascend is an accepted abbreviation) or descending (or descend). The default is ascending. The parameters must be a comma-separated list enclosed in parentheses. For example: `stat¬ ic(Price,ascending)`

For dimension search, the first parameter can be specified as nbins, depth, or rank:

- Specifying nbins causes the static module to sort result dimension values by the number of associated records in the full data set.
- Depth causes the static module to sort result dimension values by their depth in the dimension hierarchy.
- Rank causes dimension values to be sorted by the ranks assigned to them for the application. In cases when there are ties, (for example, if you specify nbins and the number of associated records is the same), the system ranks dimension search results based on the dimension value IDs.

Valid relevance ranking strings

The following are examples of valid relevance ranking strategy strings:

- `exact`
- `exact(considerFieldRanks)`
- `field,phrase,interp`
- `static(Price,ascending)`
- `static(Availability,descending),exact,static(Price,ascending)`
- `field,MyStrategy,exact` (assuming that `MyStrategy` is the name of a valid search interface with a relevance ranking strategy)
- `phrase(approximate,subphrase)`

URL query parameters for relevance ranking

URL query parameters allow you to communicate with the Presentation API from your client browser.

There are two sets of URL query parameters that allow you to specify relevance ranking modules that will order the returned record set:

- `Dk`, `Dx`, and `Ntx` parameters.
- `Nrk`, `Nrt`, and `Nrr` parameters.

Note that all of these parameters must be specified together. These sets of URL parameters are described in the following two sections.

Using the Dk, Dx, and Ntx parameters

This topic describes the use of query parameters with relevance ranking.

The following query parameters affect relevance ranking:

```
Dk=<0|1>
Dx=rel+strategy
Ntx=rel+strategy
```

For the `Dx` and `Ntx` parameters, the `rel` option sets the relevance ranking strategy. For a list of valid module names to use in the strategy, see the "Controlling relevance ranking at the query level" topic.

Relevance ranking for record search operations is automatic. Results are returned in descending order of relevance as long as a relevance ranking strategy is enabled (either in the URL or as the default for the selected search interface) and if the user has not selected an explicit record sort operation in the record search request. If the user has requested an explicit sort ordering, relevance rank ordering for results does not apply.

For dimension search operations, relevance ranking is enabled by the `Dk` parameter. The value of this (optional) parameter can be set to zero or one:

- If the value is set to one, the dimension search results will be returned in relevance-ranked order
- If the value is set to zero, the results will be returned in their default order

The default value if the parameter is omitted is zero (that is, relevance ranking is not enabled).

For both dimension search and record search operations, the relevance ranking strategy used for the current request can be selected using the search option URL parameters (`Dx` and `Ntx`) as in the following examples:

```
<application>?D=mark+twain&Dk=1
&Dx=rel+exact,static(rank,descending)

<application>?N=0&Ntk=All&Ntt=polo+shirt
&Ntx=mode+matchany+rel+MyStrategy
```

The second example assumes that MyStrategy was defined in Developer Studio, and is specified via the rel option (which sets the relevance ranking option. The example also uses the mode option (which requests "match any word" query matching).

Using URL-defined strategies (as in the first example) can be especially useful during development, when you want to compare the results of multiple strategies quickly. Once you have determined what strategy works best, you can define the strategy in a search interface in Developer Studio.

Related Links

Controlling relevance ranking at the query level on page 120
> At the MDEX Engine query level, relevance ranking strategies can be selected to override the default specified for the selected search interface.

Using the Nrk, Nrt, Nrr, and Nrm parameters

You can use the following set of parameters to order the records of a record search via a specified relevance ranking strategy.

The parameters are:

```
Nrk=search-interface
Nrt=relrank-terms
Nrr=relrank-strategy
Nrm=relrank-matchmode
```

All of these parameters must be specified together. None of the parameters allow the use of a pipe character (|) to specify multiple sets of arguments.

The definition of the parameters is as follows:

- `Nrk` sets the search interface to use in the navigation query for a record search. Only search interfaces can be specified; Endeca properties and dimensions cannot be used. Note that the search interface does not need to have a relevance ranking strategy defined in it.
- `Nrt` sets one or more terms that will be used by the relevance ranking module to order the records. For multiple terms, each term is delimited by a plus (+) sign. Note that these relevance ranking terms can be different from the search terms (as set by the `Ntt` parameter, for example).
- `Nrr` sets the relevance ranking strategy to be used to rank the results of the record search. For a list of valid module names to use in the *relrank-strategy* argument, see the "Controlling relevance ranking at the query level" topic.
- `Nrm` sets the relevance ranking match mode to be used to rank the results of the record search. With the exception of MatchBoolean, all of the search mode values listed in "Using Search Modes" (in the *MDEX Engine Basic Development Guide*) are valid for use with the `Nrm` parameter. Attempting to use MatchBoolean with the `Nrm` parameter will cause the record search results to be returned without relevance ranking and a warning to be issued to the Dgraph log.

All four parameters link to the Java `ENEQuery.setNavRelRankERecRank()` method and the .NET `ENEQuery.NavRelRankERecRank` property. Note that these parameters have a dependency on the `N` parameter, because a navigation query is being performed.

Because the `Nrt` parameter lets you specify relevance ranking terms (and not search terms), you have the freedom to perform a record search based on one set of terms (for example, *merlot* and *2003*) and then have the record set ordered by another set of terms (for example, *pear*). This behavior is different from that of the Ntx parameter, which uses the terms of the `Ntt` parameter to order the record set (in other words, the same set of search terms are also used to perform relevance ranking).

The following is an example of using these parameters:

```
<application>?N=0&Ntk=P_Description&Ntt=sonoma
&Nrk=All&Nrt=citrus&Nrr=maxfield&Nrm=matchall
```

In the example, a record search is first performed for the word sonoma against the P_Description property. Then `Nrk` specifies that the search interface named All be used. `Nrr` specifies that the Maxfield relevance ranking module use the word *citrus* (specified via Nrt) as the term by which the records are ordered, using the match mode specified by `Nrm`.

> **Note:** The `Nrk`, `Nrt`, `Nrr`, and `Nrm` parameters take precedence over the `Ntk`, `Ntt`, and `Ntx` parameters. That is, if both sets of parameters are used in a query, the relevance ranking strategy specified by the `Nrr` parameter will be used to order the records.

Related Links

Controlling relevance ranking at the query level on page 120
> At the MDEX Engine query level, relevance ranking strategies can be selected to override the default specified for the selected search interface.

Using relevance ranking methods

Because relevance ranking only affects the order of results (and not the content of results), there are no special objects or rendering techniques associated with relevance ranking.

Remember, though, that this ordering can have significant impact on how quickly results are rendered.

Relevance Ranking sample scenarios

This section contains two examples of relevance ranking behavior to further illustrate the capabilities of this feature.

In the first example, we first look at the effects of various relevance ranking strategies on a small sample data set that supports record search, examining the range of possible result orderings possible using only a limited set of ranking modules.

In the second example, we look at how adding a simple relevance ranking strategy can affect user results in the reference implementation.

> **Note:** These extremely simple scenarios are provided for illustrative purposes only. For more realistic examples, see the "Recommended strategies" topic. Also note that in many relevance ranking scenarios you can set `considerFieldRanks` for tie breaking. This setting is not useful for Dimension search because all searchable dimension value synonyms are in the same field.

Related Links

Recommended strategies on page 127
> This section provides some recommended strategies that depend on the implementation type.

Example 1: Using a small data set

This scenario shows the effects of various relevance ranking strategies on a small data set.

This example illustrates the richness of relevance ranking tuning possible with the modular Endeca relevance ranking system: using two modules on a data set of three records, we found that all four possible combinations of the modules into strategies resulted in different orderings, all of which were different from the default ordering.

The example uses the following example record set:

Record	Title property	Author property
1	Great Short Stories	Mark Twain and other authors
2	Mark Twain	William Lyon Phelps
3	Tom Sawyer	Mark Twain

Creating the search interface in Developer Studio

In Endeca Developer Studio, we have defined a search interface named Books that contains both Title and Author properties. The relevance rank is determined by the order in which the dimensions or properties appear in the Selected Members list.

Assume that we have not defined an explicit default sort order for the records, in which case their default order is determined by the system.

Without relevance ranking

Suppose that the user enters a record search query against the Books search interface for *Mark Twain*. All three of the records are matches, because each record has at least one searchable property value containing at least one occurrence of both the words Mark and Twain. But in what order should the results be presented to the application user? Without relevance ranking enabled, the results are returned in their default order: 1, 2, 3.

If relevance ranking were enabled, the order depends on the relevance ranking strategy selected.

With an Exact ranking strategy

Suppose we have selected the Exact relevance ranking strategy, either by assigning this as the default strategy for the Books search interface in Developer Studio or by using URL-level search options.

In this case, the order of results would be based only on whether results were Exact, Phrase, or other matches. Because records 2 and 3 have properties whose complete values exactly match the user query *Mark Twain*, these results would be returned ahead of record 1, with the tie being broken by the default sort set by the system (remember that we have not defined a default sort).

With an Exact ranking strategy and the `considerFieldRanks` parameter

Suppose we have selected the Exact relevance ranking strategy and also specified the `consider¬ FieldRanks` parameter in the query URL. Also, suppose that the Title property has a higher field rank value than Author for any search matches.

In this case, the order of results would be based only on whether results were Exact, Phrase, or other matches. Because records 2 and 3 have properties whose complete values exactly match the user query *Mark Twain*, these results would be returned ahead of record 1. And further, because we specified `considerFieldRanks`, record 2 would be returned ahead of record 3.

With a Field ranking strategy

Now, assume that we have selected the Field relevance ranking strategy.

The order of results would be based only on which property caused the match, with Author matches being prioritized over Title matches. Because records 1 and 3 match on Author, these are returned ahead of record 2 (again, with ties broken by the default sort imposed by the system).

With a Field,Exact ranking strategy

Now, consider using a combination of these two strategies: Field,Exact.

In this case, the primary sort is determined by the first module, Field, which again dictates that records 1 and 3 should be returned ahead of record 2. But in this case, the Field tie between records 1 and 3 is resolved by the Exact module, which prioritizes record 3 ahead of record 1. Thus, the order of results returned is: 3, 1, 2.

With an Exact,Field ranking strategy

Finally, consider combining the same two modules but in a different priority order: Exact,Field.

In this case, the primary sort is determined by the Exact module, which again prioritizes records 2 and 3 ahead of record 1. In this case, the Exact tie between records 2 and 3 is resolved by the Field module,

which orders record 3 ahead of record 2 because record 3 is an Author match. Thus, the order of results returned is: 3, 2, 1.

Example 2: UI reference implementation

This scenario shows how adding a relevance ranking module can change the order of the returned records.

This example, which is somewhat more realistically scaled, uses the sample wine data in the UI reference implementation. It demonstrates how relevance ranking can affect the results displayed to your users.

In this scenario, we use the thesaurus and relevance ranking features to enable end users' access to Flavor results similar to the one they searched on, while still seeing exact matches first.

First, in Developer Studio, we establish the following two-way thesaurus entries:

```
{ cab : cabernet }
{ cinnamon : spice : nutmeg }
{ tangy : tart : sour : vinegary }
{ dusty : earthy }
```

Before applying these thesaurus equivalencies, if we search on the Dusty flavor, 83 records are returned, and if we search on the Earthy flavor, 3,814 records are returned.

After applying these thesaurus equivalencies, if we search on the Dusty property, results for both Dusty and Earthy are returned. (Because some records are flagged with both the Dusty and Earthy descriptors, the number of records is not an exact total of the two.)

Wine (by order returned)	Relevant property
A Tribute Sonoma Mountain	Earthy
Against the Wall California	Earthy
Aglianico Irpinia Rubrato	Dusty
Aglianico Sannio	Earthy

Because the application is sorting on Name in ascending order, the Dusty and Earthy results are intermingled. That is, the first two results are for Earthy and the third is for Dusty, even though we searched on Dusty, because the two Earthy records came before the Dusty one when the records were sorted in alphabetical order.

Now, suppose that while we want our users to see the synonymous entries, we want records that exactly match the search term Dusty to be returned first. We therefore would use the Interpreted ranking module to ensure that outcome.

Wine (by order returned)	Relevant property
Aglianico Irpinia Rubrato	Dusty
Bandol Cuvee Speciale La Miguoa	Dusty
Beaujolais-Villages Reserve du Chateau de Montmelas	Dusty
Beauzeaux Winemaker's Collection Napa Valley	Dusty

With the Interpreted ranking strategy, the results are different. When we search on Dusty, we see the records that matched for Dusty sorted in alphabetical order, followed by those that matched for Earthy. The wine Aglianico Irpinia Rubrato, which was returned third in the previous example, is now returned first.

Recommended strategies

This section provides some recommended strategies that depend on the implementation type.

Relevance ranking behavior is complex and powerful and requires careful, iterative development. Typically, selection of the ideal relevance ranking strategy for a given application depends on extensive experimentation during application development. The set of possible result ranking strategies is extremely rich, and because setting ranking strategies is highly dependent on the quantity and type of data you are working with, a strategy that works well in one situation could be unsatisfactory in another.

For this reason, Oracle provides recommended strategies for different types of implementations and suggests that you use them as a point of departure in creating your own strategies. The following sections describe recommended general strategies for each product in detail.

Note: These recommendations are not meant to overrule custom strategies developed for your application by Oracle Services.

Testing your strategies

When testing your own strategies, it is a good idea to try searching on diverse examples: single word terms, multi-word terms that you know are an exact match for records in your data, and multi-word terms that contain additional words as well as the ones in your data. In this way you will see the full range of relevance ranking effects.

Recommended strategy for retail catalog data

This topic describes a good starting strategy to try if you are a retailer working with a catalog data set.

The strategy assumes the following:

- The search mode is MatchAllPartial. By using this mode, you ensure that a user's search would return a two-words-out-of-five match as well as a four-words-out-of-five match, just at a lower priority.
- The strategy is based on a search interface with members such as Category, Name, and Description, in that order. The order is significant because a match on the first member ranks more highly than a cross-field match or match on the second or third member. (For details, see "About Search Interfaces" in the *MDEX Engine Basic Development Guide*.)

The strategy is as follows:

- NTerms
- MaxField
- Glom
- Exact
- Static

The modules in this strategy work like this:

1. NTerms, the first module, ensures that in a multi-word search, the more words that match the better.
2. Next, MaxField puts cross-field matches as high in priority as possible, to the point where they could tie with non-cross-field matches.
3. The next module, Glom, decomposes cross-field matches, effectively breaking any ties resulting from MaxField. Together, MaxField and Glom provide the proper ordering, depending upon what matched.
4. Applying the Exact module means that an exact match in a highly-ranked member of the search interface is placed higher than a partial or cross-field match.
5. Optionally, the Static module can be used to sort remaining ties by criteria such as Price or SalesRank.

Recommended strategy for document repositories

This topic describes a good starting strategy to try if you are working with a document repository.

The strategy assumes the following:

- The search mode is MatchAllPartial. By using this mode, you ensure that a user's search would return a two-words-out-of-five match as well as a four-words-out-of-five match, just at a lower priority.
- The strategy is based on a search interface with members such as Title, Summary, and DocumentText, in that order. The order is significant because a match on the first member ranks more highly than a cross-field match or match on the second or third member.

The strategy is as follows:

- NTerms
- MaxField
- Glom
- Phrase (with or without approximate matching enabled)
- Static

The modules in this strategy work like this:

1. NTerms, the first module, ensures that in a multi-word search, the more words that match the better.
2. Next, MaxField puts cross-field matches as high in priority as possible, to the point where they could tie with non-cross-field matches.
3. The next module, Glom, decomposes cross-field matches, effectively breaking any ties resulting from MaxField. Together, MaxField and Glom provide the proper ordering, depending upon what matched.
4. Applying the Phrase module ensures that results containing the user's query as an exact phrase are given a higher priority than matching containing the user's search terms sprinkled throughout the text.
5. Optionally, the Static module can be used to sort the remaining ties by criteria such as ReleaseDate or Popularity.

Performance impact of Relevance Ranking

Relevance ranking can impose a significant computational cost in the context of affected search operations (that is, operations where relevance ranking is actually enabled).

You can minimize the performance impact of relevance ranking in your implementation by making module substitutions when appropriate, and by ordering the modules you do select sensibly within your relevance ranking strategy.

Making module substitutions

Because of the linear cost of relevance ranking in the size of the result set, the actual cost of relevance ranking depends heavily on the set of ranking modules used.

In general, modules that do not perform text evaluation introduce significantly lower computational costs than text-matching-oriented modules.

Although the relative cost of the various ranking modules is dependent on the nature of your data and the number of records, the modules can be roughly grouped into four tiers:

- Exact is very computationally expensive.
- Proximity, Phrase with Subphrase or Query Expansion options specified, and First are all high-cost modules, presented in the order of decreasing cost.
- WFreq can also be costly in some situations.
- The remaining modules (Static, Phrase with no options specified, Freq, Spell, Glom, Nterms, Interp, Numfields, Maxfields and Field) are generally relatively cheap.

In order to maximize the performance of your relevance ranking strategy, consider a less expensive way to get similar results. For example, replacing Exact with Phrase may improve performance in some cases with relatively little impact on results.

Note: Choose the set of modules used for relevance ranking most carefully when the data set is large or contains large/offline file content that is used for search operations.

Ordering modules sensibly

Relevance ranking modules are only evaluated as needed.

When higher-priority ranking modules determine the order of records, lower-priority modules do not need to be calculated. This can have a dramatic impact on performance when higher-cost modules have a lower priority than a lower-cost module.

While you have the freedom to order modules as you like, for best performance, make sure that the cheaper modules are placed before the more expensive ones in your strategy.

Part 3

Understanding and Debugging Query Results

Chapter 10

Using Why Match

This section describes the tasks involved in implementing the Why Match feature of the Endeca MDEX Engine.

About the Why Match feature

The Why Match functionality allows an application developer to debug queries by examining which property value of a record matched a record search query and why it matched.

With Why Match enabled in an application, records returned as part of a record search query are augmented with extra dynamically generated properties that provide information about which record properties were involved in search matching.

Enabling Why Match

You enable Why Match on a per-query basis using the `Nx` (Navigation Search Options) query parameter. No Developer Studio configuration or Dgraph flags are required to enable this feature.

However, because Why Match applies only to record search navigation requests, dynamically-generated properties only appear in records that are the result of a record search navigation query. Records in non-search navigation results do not contain Why Match properties.

Why Match API

The MDEX Engine returns match information for each record as a `DGraph.WhyMatch` property in the search results.

The following code samples show how to extract and display the `DGraph.WhyMatch` property from a record.

Java example

```
// Retrieve properties from record
PropertyMap propsMap = rec.getProperties();
// Get the WhyMatch property value
String wm = (String) propsMap.get("DGraph.WhyMatch");
// Display the WM value if one exists
```

```
if (wm != null) {
    %>This record matched on <%= wm %>
    <%
}
```

.NET example

```
// Retrieve properties from record
PropertyMap propsMap = rec.Properties;
// Get the WhyMatch property value
String wm = propsMap["DGraph.WhyMatch"].ToString();
// Display the WM value if one exists
if (wm != null) {
    %>This record matched on <%= wm %>
    <%
}
```

Why Match property format

The `DGraph.WhyMatch` property value has a three-part format that is made up of a list of fields where the terms matched, a list of the terms that matched, and several possible query expansions that may have been applied to the during processing.

The `DGraph.WhyMatch` property is returned as a JSON object with the following format: :

```
[{fields: [<FieldName>, <FieldName>, ... ], terms:[
            {term:<TermName>, expansions:[{type:<TypeName>},
                {type:<TypeName>}, ... ]},
            {term:<TermName>, expansions:[{type:<TypeName>},
                {type:<TypeName>}, ... ]}]}
        ... ]
```

where the supported expansion types (i.e. the `<TypeName>` values) are as follows:

- `Stemming` – returned results based on the stemming dictionaries available in the MDEX Engine.
- `Thesaurus` – returned augmented results based on thesaurus entries added in Developer Studio or Endeca Workbench.
- `Spell-corrected` – returned spell-corrected results using application-specific dictionary words.

The availability of these values depends on which search features have been enabled in the MDEX Engine.

For example, suppose there is a matchpartial query for "nueve uno firefighter" that produces a single-field match in "Spanish", a cross-field match in Spanish and English (i.e. "one" appears in English via thesaurus from uno), and firefighter is not in any field. The following `DGraph.WhyMatch` property value is returned:

```
[{fields:[Spanish], terms:[{term:neuve,expansions:[]},
                            {term:uno,expansions:[]}]},
 {fields:[Spanish,English], terms:[{term:neuve,expansions:[]},
                                    {term:uno,  expansions:[{type:The¬
saurus}]}]}]
```

Why Match performance impact

The response times for MDEX Engine requests that include Why Match properties are more expensive than requests without this feature. The performance cost increases as the number of records returned with the `DGraph.WhyMatch` property increases.

This feature is intended for development environments to record matching. The feature is not intended for production environments and is not particularly optimized for performance.

Chapter 11

Using Word Interpretation

This section describes the tasks involved in implementing the Word Interpretation feature of the Endeca MDEX Engine.

About the Word Interpretation feature

The Word Interpretation feature reports word or phrase substitutions made during text search processing.

The Word Interpretation feature is particularly useful for highlighting variants of search keywords that appear in displayed search results. These variants may result from stemming, thesaurus expansion, or spelling correction.

Implementing Word Interpretation

The `--wordinterp` flag to the Dgraph command must be set to enable the Word Interpretation feature.

The Word Interpretation feature does not require any work in Developer Studio. There are no Dgidx flags necessary to enable this feature, nor are there any MDEX Engine URL query parameters.

Word Interpretation API methods

The MDEX Engine returns word interpretation match information in `ESearchReport` objects.

This word interpretation information is useful for highlighting or informing users about query expansion.

The Java `ESearchReport.getWordInterps()` method and the .NET `ESearchReport.WordInterps` property return the set of word interpretations used in the current text search. Each word interpretation is a key/value pair corresponding to the original search term and its interpretation by the MDEX Engine.

In this thesaurus example, assume that you have added the following one-way thesaurus entry:

```
cab > cabernet
```

If a search for the term *cab* finds a match for *cabernet*, a single word interpretation will be returned with this key/value pair:

```
Key="cab" Value="cabernet"
```

When there are multiple substitutions for a given word or phrase, they are comma-separated; for example:

```
Key="cell phone" Value="mobile phone, wireless phone"
```

In this Automatic Phrasing example, a search for the terms *Napa Valley* are automatically phrased to *"Napa Valley"*. A key/value word interpretation is returned with the original search terms as the key and the phrased terms in double quotes as the value.

```
Key=Napa Valley Value="Napa Valley"
```

The following code snippets show how to retrieve word interpretation match information.

Java example

```
// Get the Map of ESearchReport objects
Map recSrchRpts = nav.getESearchReports();
if (recSrchRpts.size() > 0) {
  // Get the user's search key
  String searchKey = request.getParameter("Ntk");
  if (searchKey != null) {
    if (recSrchRpts.containsKey(searchKey)) {
      // Get the ERecSearchReport for the search key
     ESearchReport searchReport = (ESearchReport)recSrchRpts.get(searchKey);

      // Get the PropertyMap of word interpretations
      PropertyMap wordMap = searchReport.getWordInterps();
      // Get property iterator
      Iterator props = wordMap.entrySet().iterator();
      // Loop over properties
      while (props.hasNext()) {
        // Get individual property
        Property prop = (Property)props.next();
        String propKey = (String)prop.getKey();
        String propVal = (String)prop.getValue();
        // Display word interpretation information
        %><tr>
        <td>Original term: <%= propKey %></td>
        <td>Interpreted as: <%= propVal %></td>
        </tr><%
      }
    }
  }
}
```

.NET example

```
// Get the Dictionary of ESearchReport objects
IDictionary recSrchRpts = nav.ESearchReports;
// Get the user's search key
String searchKey = Request.QueryString["Ntk"];
if (searchKey != null) {
  if (recSrchRpts.Contains(searchKey)) {
    // Get the first Search Report object
    IDictionaryEnumerator ide = recSrchRpts.GetEnumerator();
    ide.MoveNext();
    ESearchReport searchReport = (ESearchReport)ide.Value;
    // Get the PropertyMap of word interperations
```

```
    PropertyMap wordMap = searchReport.WordInterps;
    // Get property iterator
    System.Collections.IList props = wordMap.EntrySet;
    // Loop over properties
    for (int j =0; j < props.Count; j++) {
      // Get individual property
      Property prop = (Property)props[j];
      String propKey = prop.Key.ToString();
      String propVal = prop.Value.ToString();
      // Display word interpretation information
      %><tr>
      <td>Original term: <%= propKey %></td>
      <td>Interpreted as: <%= propVal %></td>
      </tr><%
    }
  }
}
```

Troubleshooting Word Interpretation

This topic provides some corrective solutions for word interpretation problems.

The tokenization used for substitutions depends on the configuration of search characters. If word interpretation is to be used to facilitate highlighting variants of search keywords that appear in displayed search results, then the application should consider that words or phrases appearing in substitutions may not include white space, punctuation, or other configured search characters.

Chapter 12

Using Why Rank

This section describes the tasks involved in implementing the Why Rank feature of the Endeca MDEX Engine.

About the Why Rank feature

The Why Rank feature returns information that describes which relevance ranking modules were evaluated during a query and describes how query results were ranked. This information allows an application developer to debug relevance ranking behavior.

With Why Rank enabled in an application, the MDEX Engine returns records that are augmented with additional dynamically generated properties. The MDEX Engine also a returns summary information (in a `Supplement` object) about relevance ranking for a query. The properties provide information that describe which relevance ranking modules ordered the results and indicate why a particular record was ranked in the way that it was.

Enabling Why Rank

You enable Why Rank on a per-query basis using the `Nx` (Navigation Search Options) query parameter or the `Dx` (Dimension Search Options) query parameter. No Developer Studio configuration or Dgraph flags are required to enable this feature.

Why Rank API

The MDEX Engine returns relevance ranking information as a `DGraph.WhyRank` property on each record in the search results. The MDEX Engine also returns summary information for all record results in a `Supplement` object. (Note that the information available in a `Supplement` object is not available if you are using the MAX API.)

Per record match information

The following code samples show how to extract and display the `DGraph.WhyMatch` property from a record.

Java example

```
// Retrieve properties from record
PropertyMap propsMap = rec.getProperties();

// Get the WhyRank property value
String wr = (String) propsMap.get("DGraph.WhyRank");

// Display the WR value if one exists
if (wr != null) {
    %>This record was ranked by <%= wr %>
    <%
}
```

.NET example

```
// Retrieve properties from record
PropertyMap propsMap = rec.Properties;

// Get the WhyRank property value
String wr = propsMap["DGraph.WhyRank"].ToString();

// Display the WR value if one exists
if (wr != null) {
    %>This record was ranked by <%= wr %>
    <%
}
```

Summary match information

The `Supplement` object contains a "`Why Summaries`" property whose value is general summary
information for ranking of all the records returned in a query. This information includes the number of
relevance ranking modules that were evaluated, the number of strata per module, processing time per
module, and so on.

Why Rank property format

The `DGraph.WhyRank` property value has a multi-part format that is made up of a list of relevance
ranking modules that were evaluated and strata information for each module. Strata information includes
the evaluation time, rank, description, records per strata, and so on.

The `DGraph.WhyRank` property is returned as a JSON object with the following format:

```
[
    { "<RankerName>" : { "evaluationTime" : "<number>", "stratumRank" :
"<number>", "stratumDesc" : "<Description>", "rankedField" : "<FieldName>"
 }},
    ...
]
```

where the `<RankerName>` values are any of supported relevance ranking modules. The specific
number of `<RankerName>` values depends on the relevance ranking modules you enabled in the
MDEX Engine and how many of them were used to evaluate the current record.

> **Note:** If a query produces only one record in a result set, the `DGraph.WhyRank` property is
> empty because no relevance ranking was applied.

Here is an example of a query and a `DGraph.WhyRank` property from a record in the result set. Suppose there is a query submitted to an MDEX Engine using the following query parameters: `N=0&Ntk=NoEssay&Ntt=one+two&Ntx=rel+phrase(considerFieldRanks)&Nx=whyrank`. The query produces a result set where one of the records contains the following `DGraph.WhyRank` property:

```
<Property Key="DGraph.WhyRank" Value="[ { "phrase" : { "evaluationTime" :
"0", "stratumRank" : "20", "stratumDesc" : "phrase match", "rankedField" :
 "English" }} ]">
```

Result information for relevance ranking modules

In addition to the basic reporting properties that are common to each `DGraph.WhyRank` property, there are also optional reporting properties that may be included in `DGraph.WhyRank` depending on the relevance ranking module.

The basic reporting properties in `DGraph.WhyRank` that are common to all relevance ranking modules include:

- `evaluationTime` - the time spent evaluating this relevance ranking module.
- `stratumRank` - a value indicating which stratum a record is placed in.
- `stratumDesc` - the description of the relevance ranking module (often, the name of the module, or a description of options for the module).

The following table lists the optional reporting properties that are specific to each relevance ranking module.

Relevance Rank Module Name	Additional DGraph.WhyRank Properties
Exact	`rankedField` - field name for the highest ranked exact or subphrase match described in `stratumDesc`.
Field	`rankedField` - field name for the highest ranked field match.
First	`rankedField` - field name of the highest ranked field described in `stra¬ tumDesc`.
Freq	`perFieldCount` - field-by-field count of occurrences in the format "<X1> in <field1-name>, <X2> in <field2-name>, ...".
Glom	None.
Interp	`rankedField` - field name of the highest ranked field described in `stra¬ tumDesc`.
MaxFields	`rankedField` - field name of the highest ranked field described in `stra¬ tumDesc`.
NTerms	None.
NumFields	`fieldsMatched` - if `considerFieldRanks` is enabled for the module, then `fieldsMatched` is a comma-separated list of: <field-name> + "(" + <field-rank> + ")". Otherwise, `fieldsMatched` is a comma-separated list of the field names that matched.
Phrase	`rankedField` - field name of the highest ranked field (if a phrase match).

Relevance Rank Module Name	Additional DGraph.WhyRank Properties
Proximity	`rankedField` - field name of the highest ranked field (if a field match).
Spell	`rankedField` - field name of a field match that is not a spell corrected match.
Static	• `fieldCompared` - name of field sorted by. If there are multiple fields, names are pipe '|' delimited. • `directionCompared` - direction ("ascending" or "descending") of the sort. If there are multiple fields, directions are pipe '|' delimited • `fieldType` - corresponding field type ("integer", "dimension", "string", etc). If there are multiple fields, types are '|' delimited. ✎ **Note:** The Static module does not return either the `evaluationTime` or the `stratumRank` properties.
Stratify	None.
Stem	`rankedField` - field name of a field match that is not a stemmed match.
Thesaurus	`rankedField` - field name of a field match that is not a thesaurus match
WeightedFreq	None.

Why Rank performance impact

The response times for MDEX Engine requests that include Why Rank properties are more expensive than requests without this feature. The performance cost increases as the number of records returned with the `DGraph.WhyRank` property increases.

This feature is intended for development environments to troubleshoot relevance ranking. The feature is not intended for production environments and is not particularly optimized for performance.

Chapter 13

Using Why Precedence Rule Fired

This section describes the tasks involved in implementing the Why Precedence Rule Fired feature of the Endeca MDEX Engine.

About the Why Precedence Rule Fired feature

The Why Precedence Rule Fired feature returns information that explains why a precedence rule fired. This information allows an application developer to debug how dimensions are displayed using precedence rules.

With the feature enabled in an application, the root dimension values that the MDEX Engine returns are augmented with additional dynamically generated properties. The properties provide information that describe how the precedence rule was triggered (explicitly or implicitly), which dimension ID and name triggered the precedence rule, and the type of precedence rule (standard, leaf, or default).

Enabling Why Precedence Rule Fired

You enable Why Precedence Rule Fired on a per-query basis using the `Nx` (Navigation Search Options) query parameter. No Developer Studio configuration or Dgraph flags are required to enable this feature.

Why Precedence Rule Fired API

The MDEX Engine returns information about why a precedence rule fired as a `DGraph.WhyPrece¬ denceRuleFired` property on each root dimension value.

The following code samples show how to extract and display the `DGraph.WhyPrecedenceRuleFired` property from a root dimension value.

Java example

```
// Retrieve the results object.
Navigation result = results.getNavigation();

// Retrieve the refinements.
DimensionList l = result.getRefinementDimensions();
```

```
// Retrieve the dimension with ID 80000.
Dimension d = l.getDimension(800000);

// Retrieve the root dval for the dimension.
DimVal root = d.getRoot();
PropertyMap propsMap = root.getProperties();

// Get the WhyPrecedenceRuleFired property value
String wprf = (String) propsMap.get("DGraph.WhyPrecedenceRuleFired");

// Display the  value if one exists
if (wprf != null) {

    //Do something
}
```

.NET example

```
// Retrieve the results object.
Navigation result = results.Navigation;

// Retrieve the refinements.
DimensionList l = result.RefinementDimensions;

// Retrieve the dimension with ID 80000.
Dimension d = l.GetDimension(800000);

// Retrieve the root dval for the dimension.
DimVal root = d.Root;
PropertyMap propsMap = root.Properties;

// Get the WhyPrecedenceRuleFired property value
String wprf = propsMap["DGraph.WhyPrecedenceRuleFired"].ToString();

// Display the value if one exists
if (wprf != null) {

    //Do something
}
```

Why Precedence Rule Fired property format

The `DGraph.WhyPrecedenceRuleFired` property value has a multi-part format that is made up of a list of trigger reasons and trigger values that were evaluated for each precedence rule.

The `DGraph.WhyPrecedenceRuleFired` property is returned as a JSON object with the following format:

```
[
  { "triggerReason" : "<Reason>", "triggerDimensionValues" : ["<DimensionID>,
 ... "], "ruleType" : "<Type>", "sourceDimension" : "<DimensionName>",
   "sourceDimensionValue" : "<DimensionID>" },
   ...
]
```

The following table describes the reporting values in the `DGraph.WhyPrecedenceRuleFired`
property. The specific reporting values depend on the precedence rules in the MDEX Engine and how
many rules the MDEX Engine evaluated for the current set of available refinement dimensions.

Reporting Value	Description
<Reason>	The `triggerReason` can have any of the following values: • `explicit` - The precedence rule triggered because a user explicitly selected a trigger dimension value in a navigation query. The `triggerDimensionValues` is a list of dimension IDs that triggered the rule. • `explicitSelection` - The precedence rule triggered because an user explicitly selected a target dimension value, and there are more refinements available. The `triggerDimensionVa¬ lues` is a list of dimension IDs that triggered the rule. • `implicit` - The precedence rule triggered because a user implicitly selected a trigger dimension value. For example, it is implicit because a user could select a dimension value that resulted from a text search rather selecting a refinement from a navigation query. The `triggerDimensionValues` is a list of dimension IDs that triggered the rule. • `implicitSelection` - The precedence rule triggered because a user implicitly selected a target dimension value, and there are more refinements available. • `default` - The precedence rule triggered because it is a default rule that is set up to always trigger. (Forge creates default rules during automatic property mapping.)
<DimensionID>	The `triggerDimensionValues` is followed by a list of integers representing the dimension IDs.
<Type>	The `ruleType` can have any of the following values: • `standard` - Standard precedence rules display the target dimension if the source dimension value or its descendants are in the navigation state. • `leaf` - Leaf precedence rules display the target dimension only after leaf descendants of the source dimension value have been selected.
<DimensionName>	A string representing the name of the dimension.

The `DGraph.WhyPrecedenceRuleFired` property may contain any number of `triggerReason`
reporting values. However, there is one exception in the case where the value of `triggerReason` is
`default`. In that case, there would be a single `triggerReason` value.

Here is an example query that contains at least the following two URL query parameters:
`N=310002&Nx=whyprecedencerulefired`. The value of 310002 is the dimension value ID that
triggers a precedence rule for dimension 300000. The query produces a result with a root dimension
value that contains the following `DGraph.WhyPrecedenceRuleFired` property:

```
<Dimension Id=300000 Name=Number of Digits>
        <Root>
          <DimVal Name="Number of Digits" Id=300000>
            <PropertyList Size=1>
              <Property Key="DGraph.WhyPrecedenceRuleFired" Value="[ {
```

```
"triggerReason" : "explicitSelection", "triggerDimensionValues" : [310002]
} ]">
```

Performance impact of Why Precedence Rule Fired

The Why Precedence Rule Fired feature is intended for a production environment. The response times for MDEX Engine requests that include `DGraph.WhyPrecedenceRuleFired` properties are slightly more expensive than requests without this feature. In general, the feature adds performance throughput costs that are typically observed to be less than 5%.

Part 4

Content Spotlighting and Merchandizing

- *Promoting Records with Dynamic Business Rules*
- *Implementing User Profiles*

Chapter 14

Promoting Records with Dynamic Business Rules

This section describes how to use dynamic business rules for promoting contextually relevant records to application users as they search and navigate within a data set.

Using dynamic business rules to promote records

The rules and their supporting constructs define when to promote records, which records may be promoted, and also indicate how to display the records to application users.

> Note: This chapter applies to applications using the dynamic business rules feature as configured in Developer Studio, Oracle Endeca Workbench 2.1.x with Rule Manager, and the Endeca Presentation API.
>
> - If your application is based on Workbench 2.1.x with Page Builder and the Content Assembler API, read the *Page Builder Developer's Guide* and the *Content Assembler Developer's Guide*.
> - If your application is based on Oracle Endeca Experience Manager 3.1.x and the Endeca Assembler API, read the *Assembler Application Developer's Guide*.

This feature can be referred to in two ways, depending on the nature of your data:

- In a retail catalog application, this activity is called merchandising, because the Endeca records you promote often represent product data.
- In a document repository, this activity is called content spotlighting, because the Endeca records you promote often represent some type of document (HTML, DOC, TXT, XLS, and so on).

You implement merchandising and content spotlighting using dynamic business rules. Here is a simple merchandising example using a wine data set:

1. An application user enters a query with the search term Bordeaux.
2. This search term triggers a rule that is set up to promote wines tagged as Best Buys.
3. In addition to returning standard query results for term Bordeaux, the rule instructs the MDEX Engine to dynamically generate a subset of records that are tagged with both the Best Buy and Bordeaux properties.
4. The Web application displays the standard query results that match Bordeaux, as well as some number of the rule results in an area of the screen set aside for "Best Buy" records. These are the promoted records.

> Note: For the sake of simplicity, this document uses "promoting records" to generically describe both merchandising and content spotlighting.

Comparing dynamic business rules to content management publishing

Endeca's record promotion works differently from traditional content management systems (CMS), where you select an individual record for promotion, place it on a template or page, and then publish it to a Web site.

Endeca's record promotion is dynamic, or rule based. In rule-based record promotion, a dynamic business rule specifies how to query for records to promote, and not necessarily what the specific records are.

This means that, as your users navigate or search, they continue to see relevant results, because appropriate rules are in place. Also, as records in your data set change, new and relevant records are returned by the same dynamic business rule. The rule remains the same, even though the promoted records may change.

In a traditional CMS scenario, if Wine A is "Recommended," it is identified as such and published onto a static page. If you need to update the list of recommended wines to remove Wine A and add Wine B to the static page, you must manually remove Wine A, add Wine B, and publish the changes.

With Endeca's dynamic record promotion, the effect is much broader and requires much less maintenance. A rule is created to promote wines tagged as "Recommended," and the search results page is designed to render promoted wines. In this scenario, a rule promotes recommended Wine A on any number of pages in the result set. In addition, removing Wine A and adding Wine B is simply a matter of updating the source data to reflect that Wine B is now included and tagged as "Recommended." After making this change, the same rule can promote Wine B on any number of pages in the result set, without adjusting or modifying the rule or the pages.

Dynamic business rule constructs

Two constructs make up a dynamic business rule: a trigger and a target.

A trigger is a set of conditions that must exist in a query for a rule to fire. A single trigger may include a combination of dimension values and keywords. A single dynamic business rule may have one or more triggers. When a user's query contains a condition that triggers a rule, the MDEX Engine evaluates the rule and returns a set of records that are candidates for promotion to application users.

A target specifies which records are eligible for promotion to application users. A target may include dimension values, custom properties, and featured records. For example, dimension values in a trigger are used to identify a set of records that are candidates for promotion to application users.

Three additional constructs support rules:

- Zone—specifies a collection of rules to ensure that rule results are produced in case a single rule does not provide a result.
- Style—specifies the minimum and maximum number of records a rule can return. A style also specifies any property templates associated with a rule. Rule properties are key/value pairs that are typically used to return supplementary information with promoted record pages. For example, a property key might be set to "SpecialOffer" and its value set to "BannerAd.gif". A rule's style is passed back along with the rule's results, to the Web application. The Web application uses the style as an indicator for how to render the rule's results. The code to render the rule's results is part of the Web application, not the style itself.
- Rule Group —provides a means to logically organize large numbers of rules into categories. This organization facilitates editing by multiple business users.

The core of a dynamic business rule is its trigger and target values. The target identifies a set of records that are candidates for promotion to application users. The zone and style settings associated with a

rule work together to restrict the candidates to a smaller subset of records that the Web application then promotes.

Query rules and results

Once you implement dynamic business rules in your application, each query a user makes is compared to each rule to determine if the query triggers a rule.

If a user's query triggers a rule, the MDEX Engine returns several types of results:

- Standard record results for the query.
- Promoted records specified by the triggered rule's target.
- Any rule properties specified for the rule.

Two examples of promoting records

The following sections explain two examples of using dynamic business rules to promote Endeca records.

The first example shows how a single rule provides merchandising results when an application user navigates to a dimension value in a data set. The scope of the merchandising coverage is somewhat limited by using just one rule.

The second example builds on the first by providing more broad merchandising coverage. In this example, an application user triggers two additional dynamic business rules by navigating to the root dimension value for the application. These two additional rules ensure that merchandising results are always presented to application users.

An example with one rule promoting records

This example illustrates the "Recommended Chardonnays" rule.

This simple example demonstrates a basic record promotion scenario where an application user navigates to Wine_Type > White, and a dynamic business rule called "Recommended Chardonnays" promotes chardonnays that have been tagged as Highly Recommended. From a merchandising perspective, the marketing assumption is that users who are interested in white wines are also likely to be interested in highly recommended chardonnays.

The "Recommended Chardonnays" rule is set up as follows: The rule's trigger, which specifies when to promote records, is the dimension value Wine_Type > White. The rule's target, which specifies which records to promote, is a combination of two dimension values, Wine_Type > White > Chardonnay and Designation > Highly Recommended. The style associated with this rule is configured to provide a minimum of at least one promoted record and a maximum of exactly one record. The zone associated with this rule is configured to allow only one rule to produce rule results.

The "Recommended Chardonnays" rule is set up as follows:

- The rule's trigger, which specifies when to promote records, is the dimension value Wine_Type > White.
- The rule's target, which specifies which records to promote, is a combination of two dimension values, Wine_Type > White > Chardonnay and Designation > Highly Recommended.
- The style associated with this rule is configured to provide a minimum of at least one promoted record and a maximum of exactly one record.
- The zone associated with this rule is configured to allow only one rule to produce rule results.

When an application user navigates to Wine_Type > White in the application, the rule is triggered. The MDEX Engine evaluates the rule and returns promoted records from the combination of the Chardonnay and Highly Recommended dimension values. There may be a number of records that match these two dimension values, so zone and style settings restrict the number of records actually promoted to one.

The promoted record, along with the user's query and standard query results, are called out in the following graphic:
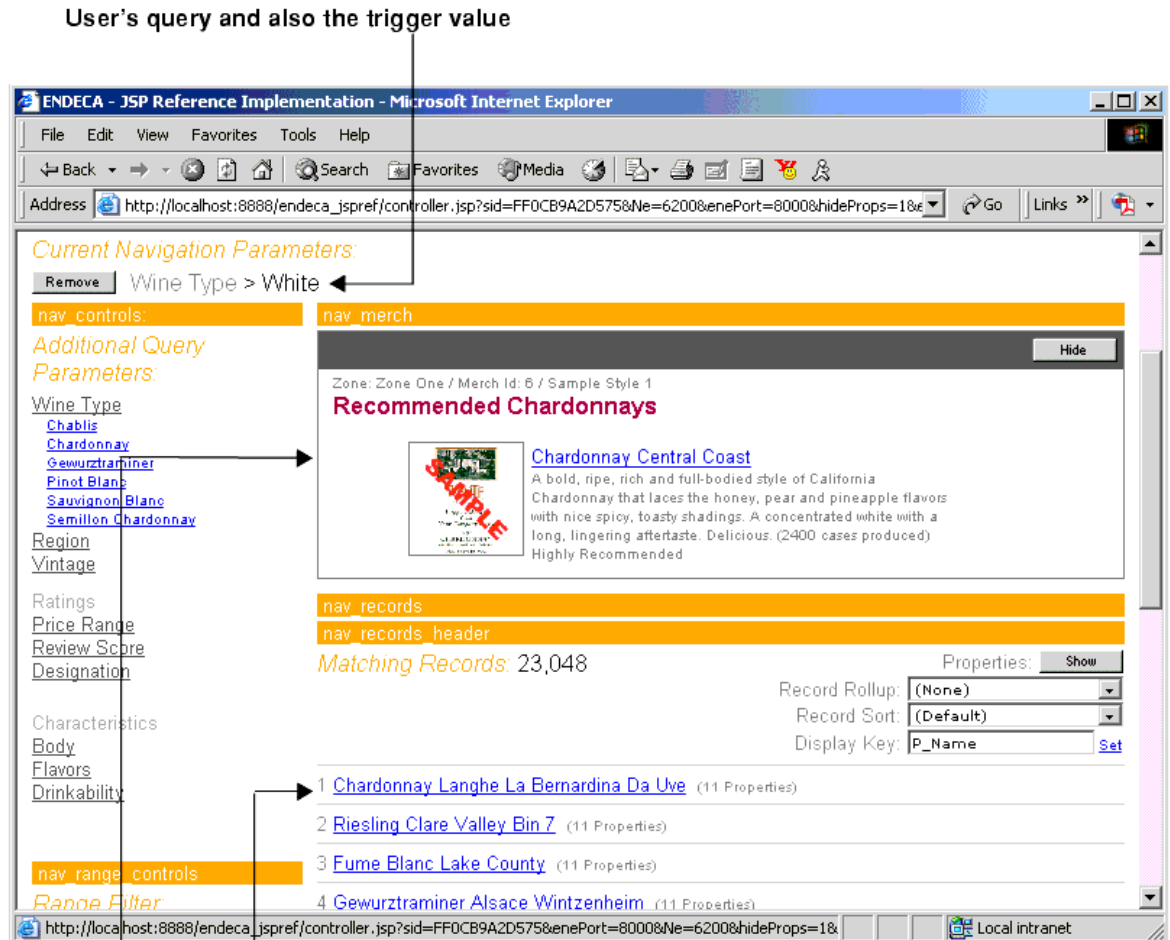
An example with three rules

The following example expands on the previous one by adding two rules called "Best Buys" and "Highly Recommended" to the rule to promote highly recommended chardonnays.

These rules promote wines tagged with a Best Buy property and a Highly Recommended property, respectively. Together, the three rules promote records to expose a broader set of potential wine purchases.

The "Best Buys" rule is set up as follows:

- The rule's trigger is set to the Web application's root dimension value. In other words, the trigger always applies.
- The rule's target is the dimension value named Best Buy.
- The style associated with this rule is configured to provide a minimum of four promoted records and a maximum of eight records.
- The zone associated with this rule is configured to allow only one rule to produce rule results.

The "Highly Recommended" rule is set up as follows:

- The rule's trigger is set to the Web application's root dimension value. In other words, the trigger always applies.
- The rule's target is the dimension value named Highly Recommended.
- The style associated with this rule is configured to provide a minimum of at least one promoted record and a maximum of three records.
- There is the only rule associated with the zone, so no other rules are available to produce results; for details on how zones can be used when more rules are available, see the topic "Ensuring promoted records are always produced."

When an application user navigates to Wine_Type > White, the "Recommended Chardonnays" rule fires and provides rule results as described in "An example with one rule promoting records". In addition, the Highly Recommended and Best Buys rules also fire and provide results because their triggers always apply to any navigation query. The promoted records for each of the three rules, along with the user's query and standard query results, are called out in the following graphic:

Suggested workflow for using Endeca tools to promote records

You can build dynamic business rules and their constructs in Developer Studio.

In addition, business users can use Endeca Workbench to perform any of the following rule-related tasks:

- Create a new dynamic business rule.
- Modify an existing rule.
- Test a rule to a preview application and preview its results.

Because either tool can modify a project, the tasks involved in promoting records require coordination between the pipeline developer and the business user. The recommended workflow is as follows:

1. A pipeline developer uses Developer Studio in a development environment to create the supporting constructs (zones, styles, rule groups, and so on) for rule and perhaps small number of dynamic business rules as placeholders or test rules.
2. An application developer creates the Web application including rendering code for each style.
3. The pipeline developer makes the project available to business users by sending the configuration to Endeca Workbench (with the option Set instance configuration).
4. A business user starts Endeca Workbench to access the project, create new rules, modify rules, and test the rules as necessary.

For general information about using Endeca tools and sharing projects, see the *Endeca Workbench Administrator's Guide*.

Note: Any changes to the constructs that support rules such as changes to zones, styles, rule groups, and property templates have to be performed in Endeca Developer Studio.

Incremental implementation of business rules

Because this is a complex features to implement, the best approach for developing your dynamic business rules is to adopt an incremental approach as you and business users of Endeca Workbench coordinate tasks.

It is also helpful to define the purpose of each dynamic business rule in the abstract (before implementing it in Developer Studio or Endeca Workbench) so that everyone knows what to expect when the rule is implemented. If rules are only loosely defined when implemented, they may have unexpected side effects.

Begin with a single, simple business rule to become familiar with the core functionality. Later, you can add more advanced elements, along with additional rules, rule groups, zones, and styles. As you build the complexity of how you promote records, you will have to coordinate the tasks you do in Developer Studio (for example, zone and style definitions) with the work that is done in Endeca Workbench.

Building the supporting constructs for a business rule

The records identified by a rule's target are *candidates* for promotion and may or may not all be promoted in a Web application. It is a combination of zone and style settings that work together to effectively restrict which rule results are actually promoted to application users.

A zone identifies a collection of rules to ensure at least one rule always produces records to promote. A style controls the minimum and maximum number of results to display, defines any property templates, and indicates how to display the rule results to the Web application. The following topics describe zone and style usage in detail.

Ensuring promoted records are always produced

You ensure promoted records are always produced by creating a zone in Developer Studio to associate with a number of dynamic business rules.

A zone is a logical collection of rules that allows you to have multiple rules available, in case a single rule does not produce a result. The rules in a zone ensure that the screen space dedicated to displaying promoted records is always populated. A zone has a rule limit that dictates how many rules may successfully return rule results.

For example, if three rules are assigned to a certain zone but the "Rule limit" is set to one, only the first rule to successfully provide rule results is evaluated. Any remaining rules in the zone are ignored.

Creating styles for dynamic business rules

You create a style in the Styles view of Endeca Developer Studio.

A style serves three functions:

- It controls the minimum and maximum number of records that may be promoted by a rule
- It defines property templates, which facilitate consistent property usage between pipeline developers and business users of Endeca Workbench
- It indicates to a Web application which rendering code should be used to display a rule's results

Using styles to control the number of promoted records

Styles can be used to affect the number of promoted records in two scenarios.

The first case is when a rule produces less than the minimum number of records. For example, if the "Best Buys" rule produces only two records to promote and that rule is assigned a style that has Minimum Records set to three, the rule does not return any results.

The second case is when a rule produces more than the maximum. For example, if the "Best Buys" rule produces 20 records, and the Maximum Records value for that rule's style is five, only the first five records are returned. If a rule produces a set of records that fall between the minimum and maximum settings, the style has no effect on the rule's results.

Performance and the maximum records setting

The Maximum Records setting for a style prevents dynamic business rules from returning a large set of matching records, potentially overloading the network, memory, and page size limits for a query.

For example, if Maximum Records is set to 1000, then 1000 records could potentially be returned with each query, causing significant performance degradation.

Ensuring consistent property usage with property templates

Rule properties are key/value pairs typically used to return supplementary information with promoted record pages.

For example, a property key might be set to "SpecialOffer" and its value set to "BannerAd.gif".

As Endeca Workbench users and Developer Studio users share a project with rule properties, it is easy for a key to be mis-typed. If this happens, then the supplementary information represented by a property does not get promoted correctly in a Web application. To address this, you can optionally create property templates for a style. Property templates ensure that property keys are used consistently when pipeline developers and Endeca Workbench users share project development tasks.

If you add a property template to a style in Endeca Developer Studio, that template is visible in Endeca Workbench in the form of a pre-defined property key with an empty value. Endeca Workbench users are allowed to add a value for the key when editing any rule that uses the template's associated style. Endeca Workbench users are not allowed to edit the key itself.

Furthermore, pipeline developers can restrict Endeca Workbench users to creating new properties based only on property templates, thereby minimizing potential mistakes or conflicts with property keys. For example, a pipeline developer can add a property template called "WeeklyBannerAd" and then make the project available to Endeca Workbench users. Once the project is loaded in Endeca Workbench, a property template is available with a populated key called "WeeklyBannerAd" and an empty value. The Endeca Workbench user provides the property value. In this way, property templates reduce simple project-sharing mistakes such as creating a similar, but not identical property called "weeklybannerad".

Note: Property templates are associated with styles in Developer Studio, not rules. Therefore, they are not available for use on the Properties tab of the Rule editor.

Using styles to indicate how to display promoted records

You indicate how to display promoted records to users by creating a style to associate with each rule and by creating application-level rendering code for the style.

You create a style in Developer Studio. You create rendering code in your Web application. This section describes how to create styles. Information about rendering code will be described later in the topic "Adding Web application code to render rule results." A style has a name and an optional title. Either the name or title can be displayed in the Web application.

When the MDEX Engine returns rule results to your application, the engine also passes the name and title values to your application. The name uniquely identifies the style. The title does not need to be unique, so it is often more flexible to display the title if you use the same title for many dimension value targets. For example, the title "On Sale" may commonly be used. Without application-level rendering code that uses the specific style or title values, the style and title are meaningless. Both require application-level rendering code in an application.

Grouping rules

Rule groups complement zones and styles in supporting dynamic business rules.

Rule groups serve two functions:

- They provide a means to logically organize rules into categories to facilitate creating and editing rules.
- They allow multiple users to access dynamic business rule simultaneously.

A rule group provides a means to organize a large number of rules into smaller logical categories, which usually affect distinct (non-overlapping) parts of a Web site.

For example, a retail application might organize rules that affect the electronics and jewelry portions of a Web site into a group for Electronics Rules and another group for Jewelry Rules. A rule group also enables multiple business users to access rule groups simultaneously. Each Workbench user can access a single rule group at a time. Once a user selects a rule group, Workbench prevents other users from editing that group until the user returns to the selection list or closes the browser window.

Prioritizing rule groups

In the same way that you can modify the priority of a rule within a group, you can also modify the priority of a rule group with respect to other rule groups.

The MDEX Engine evaluates rules first by group order, as shown in the Rules view of Developer Studio or Endeca Workbench, and then by their order within a given group.

For example, if Group_B is ordered before Group_A, the rules in Group_B will be evaluated first, followed by the rules in Group_A. Rule evaluation proceeds in this way until a zone's Rule Limit value is satisfied. This relationship is shown in the graphic below. In it, suppose zone 1 has a Rule Limit setting of 2. Because of the order of group B is before group A, rules 1 and 2 satisfy the Rule Limit rather than rules 4 and 5.

```
Group B
   Rule 1, Zone 1
   Rule 2, Zone 1
   Rule 3, Zone 2
Group A
   Rule 4, Zone 1
   Rule 5, Zone 1
   Rule 6, Zone 2
```

If you want to further prioritize the rules within a particular rule group, see the topic "Prioritizing rules."

Interaction between rules and rule groups

When creating or editing rule groups, keep in mind the following interactions between rules and rule groups.

- Rules may be moved from one rule group to another. However a rule can appear in only one group.
- A rule group may be empty (that is, it does not have to contain rules).
- The order of rule groups with respect to other rule groups may be changed.

Creating rules

After you have created your zones and styles, you can start creating the rules themselves.

An application has at least one rule group by default. Developer Studio groups all rules in this default group. As mentioned in the topic "Suggested workflow using Endeca tools to promote records," a developer usually creates the preliminary rules and the other constructs in Developer Studio, and then hands off the project to a business user to fine tune the rules and created additional rules in Endeca Workbench. However, the business user can use Endeca Workbench to perform any of the tasks described in the following sections that are related to creating a rule. For details, see Endeca Workbench Help.

Specifying when to promote records

You indicate when to promote records by specifying a trigger on the Triggers tab of the Rule editor.

A trigger can be made up of any combination of dimension values and keywords or phrases that identify when the MDEX Engine fires a dynamic business rule.

Note: A phrase represents terms surrounded in quotes.

If a user's query contains the dimension values you specify in a trigger, the MDEX Engine fires that rule. For example, in a wine data set, you could set up a rule that is triggered when a user clicks Red. If the user clicks White, the MDEX Engine does not fire the rule. If the user clicks Red, the MDEX Engine fires the rule and returns any promoted records.

If a user's query contains the keyword or phrase you specify in a trigger, the MDEX Engine fires that rule. Keywords in a trigger require that the zone associated with the rule have "Valid for search" enabled on the Zone editor in Developer Studio. Keywords in a trigger also require a match mode that specifies how the query keyword should match in order to fire the rule. There are three match modes:

- Phrase—A user's query must match all of the words of the keyword value, in the same order, for the rule to fire.
- All—A user's query must match all of the keywords in a trigger, without regard for order, for the rule to fire.
- Exact—A user's query must exactly match the keyword or words for the rule to fire. Unlike the other two modes, a user's query must exactly match the keywords in the number of words and cannot be a superset of the keywords.

Note: All modes allow the rule to fire if the spelling auto-correction and auto-phrasing, and/or stemming corrections of a user's query match the keywords or the phrase (terms surrounded in quotes).

In addition to triggers, a user profile can also be associated with a rule to restrict when to promote records. A user-profile is a label, such as premium_subscriber, that identifies an application user. If a user who has such a profile makes a query, the query triggers the associated rule. For more information, see the topic *"Implementing User Profiles."*

Multiple triggers

A rule may have any number of triggers. Adding more than one trigger to a rule is very useful if you want to promote the same records from multiple locations in your application.

Each trigger can describe a different location where a user's query can trigger a rule; however, the rule promotes records from a single target location.

Global triggers

Triggers can also be empty (no specified dimension values or keywords) on the Triggers tab.

In this case, there are two options to determine when an empty trigger fires a rule:

- Applies everywhere—Any navigation query and any keyword search in the application triggers the rule.
- Applies only at root—Any navigation query and any keyword search from the root dimension value only (N=0) triggers the rule.

Specifying a time trigger to promote records

You can further control when to promote records with time triggers.

A time trigger is a date/time value that you specify on the Time Trigger tab of the Rule editor. A time trigger specified on this tab indicates the time at which to start the rule's trigger and the time at which the trigger ends. Any matching query that occurs between these two values triggers the rule.

A time trigger is useful if you want to promote records for a particular period of time. For example, you might create a rule called "This Weekend Only Sale" whose time trigger starts Friday at midnight and expires on Sunday at 6 p.m. Only a start time value is required for a time trigger. If you do not specify an expiration time, the rule can be triggered indefinitely.

Previewing the results of a time trigger

You can test a time trigger using the Preview feature which is available on the Rule Manager page of Endeca Workbench.

In Endeca Workbench, you can specify a preview time that allows you to preview the results of dynamic business rules as if it were the preview time, rather than the time indicated by the system clock. Once you set a preview time and trigger a rule, you can examine the results to ensure the rule promotes the records that you expected it to. The Preview feature is available to Endeca Workbench users who have Approve, Edit, or View permissions.

Note that temporarily setting the MDEX Engine with a preview time affects only dynamic business rules. The preview time change does not affect any other aspect of the engine, nor does the preview time affect any scheduled updates between now and then, changes to thesaurus entries, changes to automatic phrasing, changes to keyword redirects, and so on. For example, setting the preview time a week ahead does not return records scheduled to be updated between now and a week ahead.

The MDEX Engine supports the use of a parameter called the merchandising preview time parameter as a way to test the results of dynamic business rules that have time triggers. Setting a preview time with the parameter affects only the query that uses the parameter. All other queries are unaffected.

You set a preview time in the MDEX Engine using the Java setNavMerchPreviewTime() method or the .NET NavMerchPreviewTime property in the ENEQuery object. This call requires a string value as input. The format requirement of the string is described in the topic "MDEX Engine URL query parameters for promoting records and testing time triggers."

You can also set this method by sending the necessary URL query parameters to the MDEX Engine via the UrlENEQuery class. For details, see "MDEX Engine URL query parameters for promoting records and testing time triggers".

Related Links

The MDEX Engine evaluates dynamic business rules and keyword redirects only for navigation queries.

Synchronizing time zone settings

The start time and expiration time values do not specify time zones.

The server clock that runs your Web application identifies the time zone for the start and expiration times. If your application is distributed on multiple servers, you must synchronize the server clocks to ensure the time triggers are coordinated.

Specifying which records to promote

You indicate which records to promote by specifying a target on the Target tab of the Rule editor.

A target is a collection of one or more dimension values. These dimension values identify a set of records that are all candidates for promotion. Zone and style settings further control the specific records that are actually promoted to a user.

Adding custom properties to a rule

You can optionally promote custom properties by creating key/value pairs on the Properties tab of the Rule editor.

Rule properties are typically used to return supplementary information with promoted record pages. Properties could specify editorial copy, point to rule-specific images, and so on. For example, a property name might be set to "SpecialOffer" and its value set to "BannerAd.gif." You can add multiple properties to a dynamic business rule. These properties are accessed with the same method calls used to access system-defined properties that are included in a rule's results, such as a rule's zone and style.

For details, see "Adding Web application code to extract rule and keyword redirect results".

Related Links

Adding Web application code to extract rule and keyword redirect results on page 168
>   You must add code to your Web application that extracts rule results or keyword redirect results from the Supplement objects that the MDEX Engine returns.

Adding static records in rule results

In addition to defining a rule's dimension value targets and custom properties, you can optionally specify any number of static records to promote.

These static records are called featured records, and you specify them on the Featured Records tab of the Rule editor. You access featured records in your Web application using the same methods you use to access dynamically generated records. For details, see the topic "Adding Web application code to extract rule and keyword redirect results." The MDEX Engine treats featured records differently than dynamically generated records. In particular, featured records are not subject to any of the following:

• Record order sorting by sort key
• Uniqueness constraints
• Maximum record limits

Order of featured records

The General tab of the Rule editor allows you to specify a sort order for dynamically generated records that the MDEX Engine returns.

This sort order does not apply to featured records. Featured records are returned in a Supplement object in the same order that you specified them on the Featured Records tab. The featured records occur at the beginning of the record list for the rule's results and are followed by any dynamically generated records. The dynamically generated records are sorted according to your specified sort options.

No uniqueness constraints

The Zones editor allows you to indicate whether rule results are unique (across zones) by a specified property or dimension value.

This uniqueness constraint does not apply to featured records even if uniqueness is enabled for dynamically generated rule results. For example, if you enabled "Color" to be the unique property for record results and you have two dynamically generated records with "Blue" as property value, then the MDEX Engine excludes the second record as a duplicate. On the other hand, if you have the same scenario but the two records are featured results not dynamically generated results, the MDEX Engine returns both records.

No maximum record limits

The style associated with a rule allows you to set a maximum number of records that the MDEX Engine may return as rule results.

This Maximum Records value does not apply to featured records. For example, if the Maximum Records value is set to three and you specify five featured records, the MDEX Engine returns all five records. Also, the MDEX Engine returns featured records before dynamically generated records, and the featured records count toward the maximum limit. Consequently, the number of featured records could restrict the number of dynamically generated rule results.

Sorting rules in the Rules view

The dynamic business rules you create in Developer Studio appear in the Rules view.

To make rules easier to find and work with, they can be sorted by name (in alphabetical ascending or descending order) or by priority. The procedure described below changes the way rules are sorted in Rules view only. Sorting does not affect the priority used when processing the rules. Prioritizing rules in Developer Studio is described in the topic "Prioritizing rules."

Prioritizing rules

In addition to sorting rules by name or priority, you can also modify a rule's priority in the Rules view of Developer Studio.

Priority is indicated by a rule's position in the Rules view, relative to the position of other rules when you have sorted the rules by priority. You modify the relative priority of a rule by moving it up or down in the Rules view.

A rule's priority affects the order in which the MDEX Engine evaluates the rule. The MDEX Engine evaluates rules that are higher in the Rules view before those that are positioned lower. By increasing the priority of a rule, you increase the likelihood that the rule is triggered before another, and in turn, increase the likelihood that the rule promotes records before others. It is important to consider rule priority in conjunction with the settings you specify in the Zone editor.

For example, suppose a zone has "Rule limit" set to three. If you have ten rules available for the zone, the MDEX Engine evaluates the rules, in the order they appear in the Rules view, and returns results from only the first three that have valid results. In addition, the "Shuffle rules" check box on the Zone editor overrides the priority order you specify in the Rules view. When you check "Shuffle rules", the MDEX Engine randomly evaluates the rules associated with a zone. If you set up rule groups, you can modify the priority of a rule within a group and modify the priority of a group with respect to other groups. For details, see "Prioritizing rule groups".

Related Links

> In the same way that you can modify the priority of a rule within a group, you can also modify the priority of a rule group with respect to other rule groups.

Controlling rules when triggers and targets share dimension values

The self-pivot feature controls business rules where the trigger and target of the business rule contain one or more identical dimension values.

When enabled, self-pivot allows a business rule to fire even if the user navigates to a location which explicitly contains a dimension value already in the rule target. For example, if a rule is defined as:

Trigger	Target
(No location specified -- this rule applies everywhere)	Price < $10 Region > Napa

And a user navigates to Wine Type > Red, Region > Napa, the rule still fires, despite the fact that the user is already viewing a results list for wines from the Napa region.

When self-pivot is disabled for a rule, the rule does not fire if its targets contain the same dimension values as the user's navigation state. For example, if a rule is defined as:

Trigger	Target
(No location specified -- this rule applies everywhere)	Price < $10 Region > Napa

And a user navigates to Wine Type > Red, Region > Napa, the rule does not fire because the user is already viewing a results list for wines from the Napa region.

Setting self-pivot to false does not necessarily remove all duplicates from search and merchandising results. For example, if a rule is defined as:

Trigger	Target
Wine Type > Red	Price > $10-$20

And a user navigates to Wine Type > Red, the user's navigation state does not include a dimension value from the target and the rule fires. However, because the results list contains all red wines including those in the $10-$20 range, it is still possible to get duplicate results in the merchandising and search results list.

Self-pivot is enabled by default for each new rule created in Endeca Workbench, and the option is not displayed in Endeca Workbench. However, you can change the default and set the check box to display on the Triggers tab of the Rule Manager page in Endeca Workbench. Once the check box is available, you can change self-pivot settings separately for each rule. The option is still available for

rules created or modified in Developer Studio; changing the default setting does not affect Developer Studio behavior.

Changing the default self-pivot setting when running the Endeca HTTP service from the command line

Self-pivot is enabled by default for each new rule created in Endeca Workbench, and the option is not displayed in Endeca Workbench.

In order to change the default behavior, you must set a Java parameter. Once the parameter is set (regardless of the value given for the default) the self-pivot check box displays on the Triggers tab of the Rule Manager page in Endeca Workbench. Previously existing rules are not affected by this change, and this procedure does not affect the behavior of Developer Studio.

To change the default self-pivot setting when running the Endeca HTTP service from the command line:

1. Stop the Endeca Tools Service.
2. Navigate to %ENDECA_TOOLS_ROOT%\server\bin (on Windows) or $ENDECA_TOOLS_ROOT/server/bin (on UNIX).
3. Open the setenv.bat file (on Windows) or setenv.sh (on UNIX).
4. Below "set JAVA_OPTS" add:

    *   (On Windows) CATALINA_OPTS=-Dself-pivot-default=true
    *   (On UNIX) CATALINA_OPTS=-Dself-pivot-default=true export CATALINA_OPTS

    To set the default value as disabled, use: -Dself-pivot-default=false
5. Save and close the file.
6. Run %ENDECA_TOOLS_ROOT%\server\bin\setenv.bat (on Windows) or $ENDECA_TOOLS_ROOT/server/bin/setenv.sh (on UNIX).

The self-pivot check box is now exposed on the Triggers tab of the Rule Manager page in Endeca Workbench. The check box defaults to the value specified in the setenv file.

Changing the default self-pivot setting when running the Endeca Tools Service as a Windows service

Self-pivot is enabled by default for each new rule created in Endeca Workbench, and the option is not displayed in Endeca Workbench.

In order to change the default behavior, you must set a Java parameter. Once the parameter is set (regardless of the value given for the default) the self-pivot check box displays on the Triggers tab of the Rule Manager page in Endeca Workbench. Previously existing rules are not affected by this change, and this procedure does not affect the behavior of Developer Studio.

To enable self-pivot when running the Endeca Tools Service as a Windows service:

1. Stop the Endeca Tools Service.
2. Run the Registry Editor: go to Start > Run and type regedit.
3. Navigate to HKEY_LOCAL_MACHINE > SOFTWARE > Apache Software Foundation > Procrun version > EndecaHTTPService > Parameters > Java.
4. Right click Options.

5. Choose Modify. The Edit Multi-String dialog box displays.

6. Choose Modify. The Edit Multi-String dialog box displays.
    (To set the default value as disabled, use: -Dself-pivot-default=false.)

7. Click OK.

8. Start the Endeca Tools Service.

The self-pivot check box is now exposed on the Triggers tab of the Rule Manager page in Endeca Workbench. The check box defaults to the value specified in the Registry Editor.

Working with keyword redirects

Conceptually, keyword redirects are similar to dynamic business rules in that both have trigger and target values.

However, keyword redirects are used to redirect a user's search to a Web page (that is, a URL).

The trigger of a keyword redirect is one or more search terms; the target of a keyword redirect is a URL. If a user searches with a search term that triggers the keyword redirect, then the redirect URL displays in the application. For example, you can create a keyword trigger of "delivery" and a redirect URL of http://shipping.acme.com. Or you might create a keyword redirect with a keyword trigger of "stores" and a redirect URL of http://www.acme.com/store_finder.htm.

You organize keyword redirects into keyword redirect groups in the same way and for the same reasons that you organize dynamic business rules into rule groups. Groups provide logical organization and multi-user access in Endeca Workbench. For details about how groups work, see the topic "Grouping rules." You can create keyword redirects in both Developer Studio and Endeca Workbench. For details, see the Endeca Developer Studio Help and the Endeca Workbench Help.

Displaying keyword redirects in your web application requires application coding that is very similar to the coding required to display rule results. The MDEX Engine returns keyword redirect information (the URL to display) to the web application in a Supplement object just like dynamic business rule results. The Supplement object contains a DGraph.KeywordRedirectUrl property whose value is the redirect URL. The application developer chooses what to display from the Supplement object by rendering the DGraph.KeywordRedirectUrl property rather than rendering merchandising results. In this way, the application developer codes the redirect URL to take precedence over merchandising results.

Presenting rule and keyword redirect results in a Web application

The MDEX Engine returns rule results keyword redirect results to a Web application in a Supplement object.

To display these results to Web application users, an application developer writes code that extracts the results from the Supplement object and displays the results in the application.

Before explaining how these tasks are accomplished, it is helpful to briefly describe the process from the point at which a user makes a query to the point when an application displays the rule results:

1. A user submits a query that triggers a dynamic business rule or keyword redirect.

2. When a query triggers a rule or keyword redirect, the MDEX Engine evaluates the it and returns rule results in a single Supplement object per rule or per keyword redirect.
3. Web application code extracts the results from the Supplement object.
4. Custom rendering code in your application defines how to display the rule or keyword redirect results.

The following sections describe query parameter requirements and application and rendering code requirements.

MDEX Engine URL query parameters for promoting records and testing time triggers

The MDEX Engine evaluates dynamic business rules and keyword redirects only for navigation queries.

This evaluation also occurs with variations of navigation queries, such as record search, range filters, and so on. Dynamic business rules are not evaluated for record, aggregated record, or dimension search queries. Therefore, a query must include a navigation parameter (N) in order to potentially trigger a rule. No other specific query parameters are required.

To preview the results of a rule with a time trigger, you add the merchandising preview time parameter (Nmpt) and provide a string value that represents the time at which you want to preview the application. The format of the date/time value should correspond to the following W3C format:

```
YYYY-MM-DDTHH:MM
```

The letter T is a separator between the day value and the hour value. Time zone information is omitted. Here is an example URL that sets the date/time to October 15, 2008 at 6 p.m.:

```
/controller.jsp?N=0&Nmpt=2008-10-15T18:00&Ne=1000
```

Note: The merchandising preview time parameter supports string values that occur after midnight, January 1, 1970 and before January 19, 2038. Values outside this range (either before or after the range) are ignored. Also, values that are invalid for any reason are ignored.

Adding Web application code to extract rule and keyword redirect results

You must add code to your Web application that extracts rule results or keyword redirect results from the Supplement objects that the MDEX Engine returns.

Supplement objects are children of the Navigation object and are accessed via the Java getSupplements() method or the .NET Supplements property for the Navigation object. The Java getSupplements() method and the .NET Supplements property return a SupplementList object that contains some number of Supplement objects. For example, the following sample code gets all Supplement objects from the Navigation object.

Java example

```
// Get Supplement list from Navigation object
SupplementList sups = nav.getSupplements();
// Loop over the Supplement list
for (int i=0; i<sups.size(); i++) {
 // Get individual Supplement
 Supplement sup = (Supplement)sups.get(i);
...
}
```

.NET example

```
// Get Supplement list from Navigation object
SupplementList sups = nav.Supplements;
// Loop over the Supplement list
for (int i=0; i<sups.Count; i++) {
 // Get individual Supplement
 Supplement sup = (Supplement)sups[i];
...
}
```

Composition of the Supplement object

Each Supplement object may contain three types of data: records, navigation references, and properties.

- Records—Each dynamic business rule's Supplement object has one or more records attached to it. These records are structurally identical to the records found in navigation record results. These code snippets get all records from a Supplement object. See the sample code sections below for more detail.

```
// Java example:
// Get record list from a Supplement
ERecList supRecs = sup.getERecs();
// Loop over the ERecList and get each record
for (int j=0; j<supRecs.size(); j++) {
 ERec rec = (ERec)supRecs.get(j);
...
}

//.NET example:
// Get record list from a Supplement
ERecList supRecs = sup.ERecs;
// Loop over the ERecList and get each record
for (int j=0; j<supRecs.Count; j++) {
 ERec rec = (ERec)supRecs[j];
...
}
```

- Navigation reference—Each Supplement object also contains a single reference to a navigation query. This navigation reference is a collection of dimension values. These dimension values create a navigation query that may be used to direct a user to a new location (usually the full result set that the promoted records were sampled from.) This is useful if you want to create a link from the rule's title that displays the full result set of promoted records. These code snippets get the navigation reference from a Supplement object. See the sample code sections below for more detail.

```
// Java example:
// Get navigation reference list
NavigationRefsList refs = sup.getNavigationRefs();
// Loop over the references
for (int j=0; j<refs.size(); j++) {
 DimValList ref = (DimValList)refs.get(j);
 // Loop over dimension vals for each nav reference
 for (int k=0; k<ref.size(); k++) {
  DimVal val = (DimVal)ref.get(k);
  ...
 }
}
```

```
// .NET example:
// Get navigation reference list
NavigationRefsList refs = sup.NavigationRefs;
// Loop over the references
for (int j=0; j<refs.Count; j++) {
 DimValList dimref = (DimValList)refs[j];
 // Loop over dimension vals for each nav reference
 for (int k=0; k<dimref.Count; k++) {
  DimVal val = (DimVal)dimref[k];
  ...
 }
}
```

- Properties—Each Supplement object contains multiple properties, and each property consists of a key/value pair. Properties are rule-specific, and are used to specify the style, zone, title, a redirect URL and so on. These code snippets get all the properties from a Supplement object. See the sample code sections below for more detail.

```
// Java example:
// Get property map from the Supplement
PropertyMap propsMap = sup.getProperties();
Iterator props = propsMap.entrySet().iterator();
// Loop over properties
while (props.hasNext()) {
 // Get individual property
 Property prop = (Property)props.next();
 ...
}

// .NET example:
// Get property map from the Supplement
PropertyMap propsMap2 = sup.Properties;
System.Collections.IList  props = propsMap2.EntrySet;
// Loop over properties
for (int j =0; j < props.Count; j++) {
 // Get individual property
 Property prop = (Property)props[j];
 ...
}
```

Properties in a business rule's Supplement object

There are a number of important properties for each business rule's Supplement object.

They include the following:

- Title—The title of a rule as specified on the Name field of the Rule editor.
- Style—The name of the style associated with the rule, as specified in the Style drop-down list of the Rule editor's General tab, or if the object represents a keyword redirect, the style is an empty string.
- Style Title—The title of the style (different than the name of the style) associated with the rule, as specified in the Title field on the Style editor.
- Zone—The name of the zone the rule is associated with, as specified by the Zone drop-down list of the Rule editor's General tab. If the object represents a keyword redirect, the zone is an empty string.
- DGraph.KeywordRedirectUrl—The string representing the URL redirect link for a keyword.
- DGraph.SeeAlsoMerchId—The rule ID. This ID is system-defined, not user-defined.

- DGraph.SeeAlsoPivotCount—This count specifies the total number of matching records that were available when evaluating the target for this rule. This count is likely to be greater than the actual number of records returned with the Supplement object, since only the top N records are returned for a given business rule style.
- DGraph.SeeAlsoMerchSort—If a sort order has been specified for a rule, the property or dimension name of the sort key is listed in this property.
- DGraph.SeeAlsoMerchSortOrder—If a sort key is specified, the sort direction applied for the key is also listed.

In addition to the properties listed above, you can create custom properties that on the Properties tab of the Rule editor. Custom properties also appear in a Supplement object. For details, see the topic "Adding custom properties to a rule."

Extracting rule results from Supplement objects

You can use the following sample code to assist you in extracting rule results from Supplement objects.

Java example

```
<% SupplementList sl = nav.getSupplements();
for (int i=0; i < sl.size(); i++) {
 // Get Supplement object
 Supplement sup = (Supplement)sl.get(i);
 // Get properties
 PropertyMap supPropMap = sup.getProperties();
 String sProp=null;
 // Check if object is merchandising or
 // content spotlighting result
 if ((supPropMap.get("DGraph.SeeAlsoMerchId") != null) &&
  (supPropMap.get("Style") != null) &&
  (supPropMap.get("Zone") != null)) {
 boolean hasMerch = true;
 // Get record list
 ERecList recs = sup.getERecs();
 for (int j=0; j < recs.Size(); j++) {
  // Get record
  ERec rec = (ERec)recs.get(j);
  // Get record Properties
  PropertyMap recPropsMap = rec.getProperties();
  // Get value of property (e.g. Name) from current record
  sProp =(String)recPropsMap.get("Name");
 }
 // Set target link using first Navigation Reference
 NavigationRefsList nrl = sup.getNavigationRefs();
 DimValList dvl = (DimValList)nrl.get(0);
 // Loop over dimension values to build new target query
 StringBuffer sbNavParam = new StringBuffer ();
 for (int j=0; j < dvl.size(); j++) {
 DimVal dv = (DimVal)dvl.get(j);
 // Add delimiter and id
 sbNavParam.append(dv.getId());
 sbNavParam.append(" ");
 // Get specific rule properties
 String style = (String)supPropMap.get("Style");
 String title = (String)supPropMap.get("Title");
 String zone = (String)supPropMap.get("Zone");
 // This is an example of a custom Property Template
 // defined in the Style
```

```
 String customText = (String)supPropMap.get("CustomText");
  Test output in JSP page
  %><b>%=sProp %></b><br><%
  %>Navigation:<%=sbNavParam.toString()%><br><%
  %>Style:<%=style%><br><%
  %>Title:<%=title%><br><%
  %>Zone:<%=zone%><br><%
  %>Text:<%=customText%><br><%
 }
}
%>
```

.NET example

```
// Get supplement list
SupplementList sups = nav.Supplements;
// Loop over Supplement objects
for (int i=0; i<sups.Count; i++) {
 // Get Supplement object
 Supplement sup = (Supplement)merchList[i];
 // Get properties
 PropertyMap supPropMap = sup.Properties;
 // Check if Supplement object is merchandising
 // or content spotlighting
 if ((supPropMap["DGraph.SeeAlsoMerchId"] != null) &&
     (supPropMap["Style"] != null) &&
     (supPropMap["Zone"] != null) &&
     (Request.QueryString["hideMerch"] == null)) {
  // Get Record List
  ERecList supRecs = sup.ERecs;
  // Loop over records
  for (int j=0; j<supRecs.Count; j++) {
   // Get record
   ERec rec = (ERec)supRecs[j];
   // Get property map for record
   PropertyMap propsMap = rec.Properties;
   // Get value of name prop from current record
   String name = (String)propsMap["Name"];
  }
  // Set target link using first navigation reference
  NavigationRefsList nrl = sup.NavigationRefs;
  DimValList dvl = (DimValList)nrl[0];
  // Loop over dimension values to build new target query
  String newNavParam;
  for (int k=0; k<dvl.Count; k++) {
   DimVal dv = (DimVal)dvl[k];
   // Add delimiter and id
   newNavParam += " "+dv.Id;
  }
  // Get specific rule properties
  String style = supPropMap["Style"];
  String title = supPropMap["Title"];
  String zone = supPropMap["Zone"];
  String customText = supPropMap["CustomText"];
 }
}
```

Adding Web application code to render rule results

In addition to Web application code that extracts rule results from Supplement objects, you must also add application code to render the rule results on screen.

(Rendering is the process of converting the rule results into displayable elements in your Web application pages.) Rendering rule results is a Web application-specific development task. The reference implementations come with three arbitrary styles of rendering business rule results, but most applications require their own custom development that is typically keyed on the Title, Style, Zone, and other custom properties. For details, see the topic "Adding Web application code to extract rule and keyword redirect results."

Filtering dynamic business rules

Dynamic business rule filters allow an Endeca application to define arbitrary subsets of dynamic business rules and restrict merchandising results to only the records that can be promoted by these subsets.

If you filter for a particular subset of dynamic business rules, only those rules are active and available in the Dgraph to fire in response to user queries. Rule filters support Boolean syntax using property names, property values, rule IDs, and standard Boolean operators (AND, OR, and NOT) to compose complex combinations of property names, property values, and rule IDs.

For example, a rule filter can consist of a list of workflow approval states in a multi-way OR expression. Such a filter could filter rules that have a workflow state of pending OR approved. You specify a rule filter using the Java ENEQuery.setNavMerchRuleFilter() method and the .NET ENEQuery.NavMerchRuleFilter property, and you pass the filter directly to the Dgraph as part of an MDEX Engine query.

Rule filter syntax

The syntax for rule filters supports prefix-oriented Boolean operators (AND, OR, and NOT) and uses comma-separated name/value pairs to specify properties and numeric rule IDs. The wildcard operator (*) is also supported.

Here are the syntax requirements for specifying rule filters:

- The following special characters cannot be a part of a property name or value: ( ) : , *
- Property names are separated from property values with a colon (:). The example `<applica¬ tion>?N=0&Nmrf=state:approved` filters for rules where state property has a value of approved.
- Name/value pairs are separated from other name/value pairs by a comma. The example `<appli¬ cation>?N=0&Nmrf=or(state:pending,state:approved)` filters for rules where state property is either approved or pending.
- Rule IDs are specified by their numeric value. The example `<application>?N=0&Nmrf=5` filters for a rule whose ID is 5.
- Multiple rule IDs, just like multiple name/value pairs, are also separated by a comma. The example `<application>?N=0&Nmrf=or(1,5,8)` filters for rules where the value of the rule ID is either 1, 5, or 8.
- Boolean operators (AND, OR, and NOT) are available to compose complex combinations of property names, property values, and rule IDs. The example `<application>?N=0&Nmrf=and(im¬ age_path:/common/images/book.jpg,alt_text:*)` filters for rules where the value of the image_path property is book.jpg and alt_text contains any value including null.

- Wildcard operators can substitute for any property value (not property name). The example `<ap¬ plication>?N=0&Nmrf=and(not(state:*),not(alt_text:*))` filters for rules that contain no value for both the state property and alt_text property.

Additional Boolean usage information

- Boolean operators are not case-sensitive.
- Boolean operators are reserved words, so property names or values such as "and," "or," and "not" are not valid in rule filters. However, properties can contain any superset of the Boolean operators such as "andrew", "bread and butter", or "not yellow".
- Although the Boolean operators in rule filters are not case-sensitive, property names and values in the filter are case sensitive.

MDEX URL query parameters for rule filters

The Nmrf query parameter controls the use of a rule filter.

`Nmrf` links to the Java `ENEQuery.setNavMerchRuleFilter()` method and the .NET `ENEQuery.NavMerchRuleFilter` property. The `Nmrf` parameter specifies the rule filter syntax that restricts which rules can promote records for a navigation query.

Performance impact of dynamic business rules

Dynamic business rules require very little data processing or indexing, so they do not impact Forge performance, Dgidx performance, or the MDEX Engine memory footprint.

However, because the MDEX Engine evaluates dynamic business rules at query time, rules affect the response-time performance of the MDEX Engine. The larger the number of rules, the longer the evaluation and response time. Evaluating more than twenty rules per query can have a noticeable effect on response time. For this reason, you should monitor and limit the number of rules that the MDEX Engine evaluates for each query.

In addition to large numbers of rules slowing performance, query response time is also slower if the MDEX Engine returns a large number of records. You can minimize this issue by setting a low value for the Maximum Records setting in the Style editor for a rule.

Rules without explicit triggers

Dynamic business rules without explicit triggers also affect response time performance because the MDEX Engine evaluates the rules for every navigation query.

Applying relevance ranking to rule results

In some cases, it is a good idea to apply relevance ranking to a rule's results.

For example, if a user performs a record search for Mondavi, the results in the Highly Rated rule can be ordered according to their relevance ranking score for the term Mondavi. In order to create this effect, there are three requirements:

- The navigation query that is triggering the rule must contain record search parameters (Ntt and Ntk). Likewise, the zone that the rule is assigned to must be identified as Valid for search. (Otherwise, the rule will not be triggered.)
- The rule's target must be marked to Augment Navigation State.
- The rule must not have any sort parameters specified. If the rule has an explicit sort parameter, that parameter overrides relevance ranking. Sort parameters for a rule are set on the General tab of the Rule editor.

If these three requirements are met, then the relevance ranking rules specified with MDEX Engine startup options are used to rank specific business rules when triggered with a record search request (a keyword trigger).

About overloading Supplement objects

Recall that dynamic business rule results are returned to an application in Supplement objects.

Each rule that returns results does so via a single Supplement object for that rule. However, not all Supplement objects contain rule results.

Supplement objects are also used to support "Did You Mean" suggestions, record search reports, and so on. In other words, a Supplement object can act as a container for a variety of features in an application. One Supplement object instance cannot contain results for two features. For example, one Supplement object cannot contain both rule results and also "Did You Mean" suggestions. For that reason, if you combine dynamic business rules with these additional features, you should check each Supplement object for specific properties such as DGraph.SeeAlsoMerchId to identify which Supplement object contains rule results.

Chapter 15

Implementing User Profiles

This section describes how to create user profiles that can be used in your Endeca application.

About user profiles

A user profile is a character-string-typed name that identifies a class of end users.

User profiles enable applications built on the Endeca Information Access Platform to tailor the content displayed to an end user based on that user's identity.

User profiles can be used to trigger dynamic business rules, where such rules are optionally constructed with an additional trigger attribute corresponding to a user profile. Oracle Endeca Guided Search can accept information about the end user, and use that information to trigger pre-configured rules and behaviors.

You set up user profiles in Developer Studio. Both Developer Studio and Oracle Endeca Workbench allow a user profile to be associated with a business rule's trigger.

This feature discusses how you create user profiles and then implement them as dynamic business rule triggers. Before reading further, make sure you are comfortable with the information in the "Promoting Records with Dynamic Business Rules" section.

> ✎ **Note:** Each business rule is allowed to have at most one user profile trigger.

Related Links

> This section describes how to use dynamic business rules for promoting contextually relevant records to application users as they search and navigate within a data set.

Profile-based trigger scenario

This topic shows how a dynamic business rule would utilize a user profile.

In the following scenario, an online clothing retailer wants to set up a dynamic business rule that says: "For young women who are browsing stretch t-shirts, also recommend cropped pants." We follow the shopping experience of a customer named Jane.

In order to set up this rule, a few configuration steps are necessary:

1. In Endeca Developer Studio, the retailer creates a user profile called young_woman, which corresponds to the set of customers who are female and are between the ages of 16 and 25.
2. In Endeca Workbench, a dynamic business rule that uses the profile as a trigger is created, as shown below. No complex Boolean logic programming is necessary here. The business user simply selects a user profile from a set of available profiles to create the business rule.

```
young_woman X DVAL(stretch t-shirt) => DVAL(cropped pants)
```

3. In the Web application that's driving the customer's experience, there needs to be logic that identifies the user and tests to see if he or she meets the requirements to be classified as a young_woman. Alternatively, the profile young_woman may already be stored along with Jane's information (such as age, address, and income) in a database or LDAP server.

The user's experience would go something like this:

1. Jane accesses the clothing retailer's Web site and is identified by a cookie on her computer. By looking up a few database tables, the application knows that it has interacted with her before. The database indicates that she is 19 years old and female.

   At this point, the database may also indicate the user profiles that she belongs to: young_woman, r_and_b_music_fan, college_student. Alternatively, the application logic may test against her information to see which profiles she belongs to, as follows: "Jane is between 16 and 25 years old and she is female, so she belongs in the young_woman profile."

2. As Jane is browsing the site, the Endeca MDEX Engine is driving her catalog experience. As each query is being sent to the Endeca MDEX Engine, it is augmented with user profile information. Here is some sample Java code:

```
profileSet.add("young_woman");
eneQuery.setProfiles(profileSet);
```

3. As Jane clicks on a stretch t-shirt link, the Endeca MDEX Engine realizes that a dynamic business rule has been triggered: young_woman X DVAL(stretch t-shirt). Therefore, it returns a cropped pants record in one of the dynamic business rule zones.
4. Jane sees a picture of cropped pants in a box labeled, "You also might like..."

User profile query parameters

There are no URL MDEX query parameters associated with user profiles.

In many live application scenarios, the URL query is exposed to the end user, and it is usually not appropriate for end users to see or change the user profiles with which they have been tagged.

API objects and method calls

These Java and .NET code samples demonstrate how to implement user profiles in the Web application.

In the following code samples, the application recognizes the end user as Jane Smith, looks up some database tables and determines that she is 19 years old, female, a college student and likes R&B music. These characteristics map to the following Endeca user profiles created in Endeca Developer Studio:

- young_woman
- r_and_b_music_fan

- `college_student`

User profiles can be any string. The user profiles supplied to `ENEQuery` must exactly match those configured in Endeca Developer Studio.

Java example of implementing user profiles

```
// User profiles can be any string. The user profiles must
// exactly match those configured in Developer Studio.
// Add this import statement at the top of your file:
// import java.util.*;
Set profiles = new HashSet();
// Collect all the profiles into a single Set object.
profiles.add("young_woman");
profiles.add("r_and_b_music_fan");
profiles.add("college_student");
// Augment the query with the profile information.
eneQuery.setProfiles(profiles);
```

.NET example of implementing user profiles

```
// Make sure you have the following statement at the top
// of your file:
// using System.Collections.Specialized;
StringCollection profiles = new StringCollection();
// Collect all the profiles into a single StringCollection object.
profiles.Add("young_woman");
profiles.Add("r_and_b_music_fan");
profiles.Add("college_student");
// Augment the query with the profile information.
eneQuery.Profiles = profiles;
```

Performance impact of user profiles

An application using this feature may experience additional memory costs due to user profiles being set in an ENEQuery object.

In addition, the application may require additional Java `ENEConnection.query()` or .NET `HttpENEConnection.Query()` response time, because the MDEX Engine must do additional work to receive profile information and check if business rules fire. However, in typical application scenarios that set one to five user profile strings of at most 20 characters in the `ENEQuery` object, the performance impact is insignificant.

Part 5

Other Features

- *Using Internationalized Data*
- *MDEX Engine Logging Variables*

Chapter 16

Using Internationalized Data

This section describes how to include internationalized data in an Endeca application.

Using internationalized data in your Endeca application

The MDEX Engine supports the following 22 languages:

Catalan, Chinese (Simplified), Chinese (Traditional), Czech, Dutch, English, French, German, Greek, Hebrew, Hungarian, Italian, Japanese, Korean, Polish, Portuguese, Romanian, Russian, Spanish, Swedish, Thai, and Turkish.

This chapter describes language support concepts and how to configure Forge, Dgidx, and the Dgraph to process source data in any of the supported languages.

The chapter makes the following assumptions:

- If working with Chinese, you are familiar with the encoding and character sets (Traditional versus Simplified, Big5, GBK, and so on).
- If working with Chinese or Japanese, you know that these languages do not use white space to delimit words.
- If working with Japanese, you are familiar with the shift_jis variants and how the same character can be represent either the Yen symbol or the backslash character.

About language identifiers

American English (`en`) is the default language of the MDEX Engine.

If your application contains text in non-English languages, you should specify the language of the text to the MDEX Engine, so that it can correctly perform language-specific operations.

You use a language ID to identify a language. Language IDs must be specified as a valid RFC-3066 or ISO-639 code, such as the following examples:

- `da` – Danish
- `de` – German
- `el` – Greek
- `en` – English (United States)
- `en-GB` – English (United Kingdom)
- `es` – Spanish

- `fr` – French
- `it` – Italian
- `ja` – Japanese
- `ko` – Korean
- `nl` – Dutch
- `pt` – Portuguese
- `zh` – Chinese
- `zh-CN` – Chinese (simplified)
- `zh-TW` – Chinese (traditional)

A list of the ISO-639 codes is available at:

`http://www.w3.org/WAI/ER/IG/ert/iso639.htm`

About language collations

During both indexing and query processing, the text of a language is collated (sorted) according to a collation setting. You specify the collation setting as an argument to the `--lang` flag of Dgidx.

There are two primary types of collations--the `endeca` collation and the `standard` collation.

The `endeca` collation sorts text with lower case before upper case and does not account for character accents and punctuation. For example, the `endeca` collation sorts text as follows:

```
0 < 1 < ... < 9 < a < A < b < B < ... < z < Z
```

In applications where English is the global language, the `endeca` collation performs better during indexing and query processing than the `standard` collation primarily because the `endeca` collation is optimized for unaccented languages. In applications with other languages, the collation results for accented characters may not be what is expected. (There may be cases where an application has text with accented characters and you choose the `endeca` collation for performance reasons.) The `endeca` collation is the default collation.

The `standard` collation sorts data according to the International Components for Unicode (ICU) standard for the language you specify with `--lang <lang id>`. For details about standard collation for a particular language, see the Unicode Common Locale Data Repository at `http://cldr.unicode.org/`.

In applications that include internationalized data, the `standard` collation is typically the more appropriate choice because it accounts for character accents during sorting.

In addition to the `endeca` and `standard` collations, there are two other language-specific ICU collations supported by Dgidx and the Dgraph:

- `de-u-co-phonebk` (A German collation that sorts according to phone book order rather than by dictionary order.)
- `es-u-co-trad` (A Spanish collation that sorts the ch and ll characters in the traditional order rather than the standard order.)

Configuring Forge with the encoding of source data

You must specify the encoding of incoming source data in order for Forge to process it correctly.

The format of the source data determines how you specify the encoding value.

To specify the encoding of source data:

- If the format is Delimited, Vertical, Fixed-width, Exchange, ODBC, JDBC Adapter, or Custom Adapter, specify the encoding in the **Encoding** field of the Record Adapter editor in Developer Studio, for example, UTF-8, Latin1, etc.. For a list of valid source data encodings, see the ICU Converter Explorer at *http://demo.icu-project.org/icu-bin/convexp?s=ALL*
- If the format is XML, specify the encoding in the DOCTYPE declaration of the XML source data document. The **Encoding** value are ignored.
- If the format is Binary, no specification is required. The **Encoding** value is ignored because encoding only applies to text-based files.

Configuring Dgidx to process internationalized data

The following topics describe how to configure Dgidx for character mapping and language sorting (collation).

Configuring language-specific analysis

Dgidx and the Dgraph can use either Latin-1 language analysis or Oracle Language Technology (OLT) analysis. This topic compares the two types of language analysis, including their effects on various Endeca features, and provides general guidance about which type of language analysis may be appropriate for your application.

Latin-1 language analysis

Latin-1 language analysis is available for all languages except Chinese (Simplified, Traditional), Japanese, and Korean. Latin-1 analysis provides a language agnostic approach to indexing text and supports the following features:

- Whitespace tokenization: the process of dividing text into words, phrases, symbols, or other meaningful elements delimited by spaces.
- Wildcard search
- Phrase search
- Search characters
- Diacritic folding
- Static stemming
- Stop words

Latin-1 language analysis is enabled by default for the following languages: Dutch, English, French, German, Italian, Portuguese, and Spanish.

Oracle Language Technology (OLT) language analysis

Oracle Language Technology analysis provides language specific, dictionary based, linguistic analysis features including the following:

- Segmentation: the process of breaking up the non-whitespace language's text into meaningful units.
- Tokenization: the process of breaking a stream of text up into words, phrases, symbols, or other meaningful elements.

- Orthographic normalization: the creation of a standard indexed form for diacritic marks.
- Decompounding: the decomposition of compound word forms into their base terms.
- Dynamic stemming: the process of determining the base form of a word form. The process is based on dictionary entries and language specific rules.
- Stop words

🖉 **Note:** The disadvantage of OLT analysis is its compatibility with other Endeca record and dimension search features. If you index your data using Dgidx with OLT analysis, there is no support for Wildcard search, phrase search, search characters, and diacritic folding. If these features are required in your application, use Latin-1 analysis.

Oracle Language Technology analysis is enabled by default for the following languages: Catalan, Chinese (Simplified), Chinese (Traditional), Czech, Greek, Hebrew, Hungarian, Japanese, Korean, Polish, Romanian, Russian, Swedish, Thai, and Turkish.

Comparing stemming feature differences

Keep in mind the following distinctions about the two types of stemming:

- *Static* stemming provides an explicit list of words and their corresponding stemmed variants. You can easily supplement or replace the stemming word lists as necessary. Static stemming is enabled with the Latin-1 analyzer. In the context of stemming, the advantage of the Latin-1 analyzer is its compatibility with other Endeca record and dimension search features.
- *Dynamic* stemming provides richer stemming matches by applying linguistic rules, algorithmic rules, and orthographic normalization to match word forms within a particular language. Dynamic stemming is enabled with the OLT analyzer.

Language analysis support

The following table lists each language that the MDEX Engine supports and its corresponding default language analyzer. With the exception of Chinese, Japanese, and Korean, you can modify the default configuration to use either OLT or Latin-1. CJK languages are set to OLT only. No matter which analyzer you use, specify the `--lang` flag to Dgidx with the appropriate `<lang Id>` argument for your application.

Language	Default Language Analyzer	Enablement	Modifying the Default
Catalan	OLT	Specify `--lang ca` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `ca_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. (In this case, the existence of the file switches language analysis modes without any other required settings.)
Chinese (Simplified)	OLT	Specify `--lang zh-CN` to Dgidx.	Not supported.
Chinese (Traditional)	OLT	Specify `--lang zh-TW` to Dgidx.	Not supported.

Language	Default Language Analyzer	Enablement	Modifying the Default
Czech	OLT	Specify `--lang cs` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `cs_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. Then, set `USE_STATIC_WORD¬FORMS="TRUE"` in `<app name>.stemming.xml`.
Dutch	Latin-1	Specify `--lang nl` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"` in `<app name>.stemming.xml`.
English	Latin-1	Specify `--lang en` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"` in `<app name>.stemming.xml`.
French	Latin-1	Specify `--lang fr` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"` in `<app name>.stemming.xml`.
German	Latin-1	Specify `--lang de` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"` in `<app name>.stemming.xml`.
Greek	OLT	Specify `--lang el` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `el_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. Then, set `USE_STATIC_WORD¬FORMS="TRUE"` in `<app name>.stemming.xml`.
Hebrew	OLT	Specify `--lang he` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `he_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming` directory. (In this case, the existence of the file switches language analysis modes without any other required settings.)
Hungarian	OLT	Specify `--lang hu` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `hu_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. Then, set `USE_STATIC_WORD¬FORMS="TRUE"` in `<app name>.stemming.xml`.

Language	Default Language Analyzer	Enablement	Modifying the Default
Italian	Latin-1	Specify `--lang it` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"`in `<app name>stemming.xml`.
Japanese	OLT	Specify `--lang ja` to Dgidx.	Not supported.
Korean	OLT	Specify `--lang ko` to Dgidx.	Not supported.
Polish	OLT	Specify `--lang pl` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `pl_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. Then, set `USE_STATIC_WORD¬FORMS="TRUE"` in `<app name>.stemming.xml`.
Portuguese	Latin-1	Specify `--lang pt` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"`in `<app name>stemming.xml`.
Romanian	OLT	Specify `--lang rm` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `rm_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. (In this case, the existence of the file switches language analysis modes without any other required settings.)
Russian	OLT	Specify `--lang ru` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `ru_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. Then, set `USE_STATIC_WORD¬FORMS="TRUE"` in `<app name>.stemming.xml`.
Spanish	Latin-1	Specify `--lang es` to Dgidx.	Switch to OLT by setting `USE_STAT¬IC_WORDFORMS="FALSE"`in `<app name>stemming.xml`.
Swedish	OLT	Specify `--lang sv` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `sv_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. (In this case, the existence of the file switches language analysis

Language	Default Language Analyzer	Enablement	Modifying the Default
			modes without any other required settings.)
Thai	OLT	Specify `--lang th` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `th_word_forms_collection.xml` and adding it to the `Endeca\MDEX\6.4.0\conf\stemming\custom` directory. (In this case, the existence of the file switches language analysis modes without any other required settings.)
Turkish	OLT	Specify `--lang tr` to Dgidx.	Switch to Latin-1 by creating a static stemming file named `rm_word_forms_collection.xml` and adding it to the `MDEX\6.4.0\conf\stemming\custom` directory. (In this case, the existence of the file switches language analysis modes without any other required settings.)

Configuring language-specific spelling correction

You can enable language-specific spelling correction to prevent queries in one language from being spell-corrected to words in a different language.

This feature works by creating separate dictionaries for each language. The dictionaries are generated from the source data and therefore require that the source data be tagged with a language ID. You should also use a per-query language ID, so that the MDEX Engine can select the appropriate dictionary for a given query.

Note: The language-specific spelling correction feature uses the Espell language engine, which is part of the base product. The Aspell language engine only supports English, and so it is not supported for this feature.

To enable the language-specific spelling correction feature:

1. Using a text editor, create a new file and add the following text to the file:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE SPELL_CONFIG SYSTEM "spell_config.dtd">
<SPELL_CONFIG>
  <SPELL_ENGINE>
    <DICT_PER_LANGUAGE>
      <ESPELL/>
    </DICT_PER_LANGUAGE>
  </SPELL_ENGINE>
</SPELL_CONFIG>
```

Note that Developer Studio does not create this configuration file.

2. Save the file as `db_prefix.spell_config.xml` file (where *db_prefix* is the prefix for your project instance configuration).

3. Place the file in the directory where the project's other XML instance configuration files reside.

4. Run a baseline update and restart the MDEX Engine with the new configuration file.

For more information about the structure of a `spell_config.xml` file, see the `spell_config.dtd` in the Endeca MDEX Engine `conf/dtd` directory.

If a `spell_config.xml` file exists, it overrides the use of the following parameters to the Dgidx `--spellmode` option:

- `espell`
- `aspell`
- `aspell_OR_espell`
- `aspell_AND_espell`

About auxiliary dictionaries for OLT analysis

You can optionally add an auxiliary dictionary to supplement a default OLT dictionary for any supported language. This may be necessary if searches for terms that you know exist in your data are not producing the expected results.

The auxiliary dictionary is a UTF-8 encoded file that is line oriented and tab delimited. Each line in the file represents an entry to supplement the primary dictionary.

Here is the generic syntax for a line:

```
COMMAND value1 value 2 ...
```

Each `value` is tab delimited and depends on the `COMMAND`. In the context of the stemming feature, the `COMMAND` should be set only to `STEM`. Other `COMMAND` values are not described in this guide.

Each line beginning with `STEM` represents a stemming entry that includes the term and one or more part of speech attributes to qualify the term. Each attribute is comma delimited.

```
STEM new_term1 POS, ...
STEM new_term2 POS, ...
STEM new_term3 POS, ...
```

You can specify the part of speech attributes by their full name or abbreviation (in parentheses):

- `noun (N)` - a simple noun, like table, book, procedure
- `nounProper (propN)` - a proper name, for person, place, etc., typically capitalized, like Zachary, Supidito, Susquehanna
- `verb (V)` - any verb in its dictionary form, like deconstruct, upsell, skate
- `adjective (Adj)` - modifiers of nouns, typically can be compared (green, greener, greenest), like fast, trenchant, pendulous
- `adverb (Adv)` - any general modifier of a sentence that may modify an adjective or verb or may stand alone, like slowly, yet, perhaps
- `preposition (Prep)` - a word that forms a prepositional phrase with a noun, like off, beside, from. Used for postpositions too, in languages that have postpositions of similar function.
- `punct (Punct)` - any non-letter symbol that is treated as a unit by itself, like %, $,]
- `pronoun (Pro)` - any pronominal form, including personal pronouns (I,they), demonstrative pronouns (those, this), relative pronouns (who, which, wherever)
- `interrog (Wh)` - an interrogative word, like who, why, when, where, how

- `determiner (Det)` - words that carry grammatical information about a noun group, for example definite/indefinite, like the, a, an
- `particle (Part)` - small, invariant words that convey grammatical information; also used for interjections.
- `conjunction (Conj)` - conjunctions that introduce a subordinate clause, e.g. although, because, while, and conjunctions that introduce a coordinate clause, e.g. and, or, yet
- `numCardinal (Card)` - cardinal numbers, like thirteen, 100, five
- `numOrdinal (Ord)` - ordinal numbers, like thirteenth, 100th, fifth

For example, this German auxiliary dictionary shows three entries. Each entry is marked with the optional attribute `N` to indicate it is a noun:

```
STEM aalglatt N
STEM aalglatte N
STEM aalglatter N
```

Creating an auxiliary OLT dictionary

To create an auxiliary dictionary:

1. Start a text editor that supports UTF-8 characters and enables you to edit the language you want to supplement.
2. Create a new UTF-8 encoded file.
3. Add words to the dictionary. Start each word on a separate line that begins with the command `STEM`, followed by the word or character, any optional attributes, and then a carriage return.
4. Optionally, add comments to the file.

   Comments must begin with a pound sign (#). You can also have blank lines in this file.

5. Save the dictionary file with the filename `dictionary.<RFC3066 language code>.dict` in the `%ENDECA_MDEX_ROOT%\olt` directory on Windows or in `$ENDECA_MDEX_ROOT/olt` on UNIX.

   Note that the dictionary name does not need to include a region code unless you are using either simplified Chinese (`zh-CN`) or traditional Chinese (`zh-TW`). For example: `%ENDECA_MDEX_ROOT%\olt\dictionary.zh-CN.dict`.

6. Re-index your data and specify the `--lang` flag to Dgidx with appropriate `<lang Id>` value.

   Note that the `<lang Id>` value does not need to include a region code unless you are using either simplified Chinese (`zh-CN`) or traditional Chinese (`zh-TW`).

7. Restart the Dgraph.

Mapping accented characters to unaccented characters

Dgidx supports mapping Latin1, Latin extended-A, and Windows CP1252 international characters to their simple ASCII equivalents during indexing. You can optionally specify the `--diacritic-folding` flag on Dgidx to map accented characters to simple ASCII equivalents. This allows the Dgraph to match Anglicized search queries such as *cafe* against result text containing international characters (accented) such as *café*.

Note: This flag is not compatible with the OLT analysis. It is only compatible with Latin-1 analysis that uses an XML configuration file of word forms.

Configuring the MDEX Engine with language identifiers for source data

The following topics describe how to specify language identifiers for the MDEX Engine and how to configure language-specific spelling correction.

Specifying a language ID

You can specify a language ID for source data using one of these methods:

- A *global* language ID can be used if all or most of your text is in a single language.
- A *per-record* language ID should be used if the language varies on a per-record basis.
- A *per-dimension/property* language ID should be used if the language varies on a per-dimension basis.
- A *per-query* language ID should be used in your front-end application if the language varies on a per-query basis.

The following topics describe these methods of specifying the language ID for your data.

Specifying a global language ID and collation order

If most of the text in an application is in a single language, you can specify a global language ID by providing the `--lang` flag and a `<lang-id>` argument to the Dgidx and Dgraph components. The MDEX Engine treats all text as being in the language specified by `<lang-id>`, unless you tag text with a more specific language ID (that is, per-record, per-dimension, or per-query language IDs). The `<lang-id>` defaults to `en` (US English) if left unspecified.

For example, to indicate that text is English (United Kingdom), specify: `--lang en-GB`.

In addition to specifying a language identifier, you can also specify an optional collation order using an argument to the `--lang` flag. A collation is specified in the form:

`--lang <lang-id>-u-co-<collation>`, where:

- `<lang-id>` is the language Id and may also include a sub-tag. If unspecified, the value of `<lang-id>` is `en` (US English).
- `-u` is a separator value between the language identifier portion of the argument and the collation identifier portion of the argument.
- `-co` is a key that indicates a collation value follows.
- `-<collation>` is the collation type of either `endeca`, `standard`, or in some cases, other language-specific ICU collations such as `phonebk` or `trad`. If unspecified, the value of `<colla¬tion>` is `endeca` (that is `en-u-co-endeca`).

For example, `--lang de-u-co-phonebk` instructs Dgidx and the Dgraph to treat all the text as German and collate the text in phonebook order.

Specifying a per-record language ID

If your application data is organized so that all the data in a record is in a single language but different records are in different languages, you should use a per-record language ID.

To specify a language ID, create a Perl manipulator or Java manipulator in your Forge pipeline, and configure it to add a property or dimension named `Endeca.Document.Language` to your records. The value of the property or dimension should be a valid RFC-3066 or ISO-639 language ID.

Specifying a per-dimension/property language ID

Use per-dimension/property language IDs if your application tends to have mixed-language records and the languages are segregated into different dimensions or properties.

For example, your data may have an English property called Description and a Spanish property called Descripción. In this case, because an individual record can have both English and Spanish text, a per-property language ID would be more appropriate than a per-record language ID.

You can specify a per-dimension/property language ID in Developer Studio with the Property or Dimension editors.

To configure a language ID for a property or dimension:

1. In the Project tab of Developer Studio, double-click either **Properties** (to open the Properties view) or **Dimensions** (to open the Dimensions view).
2. From the appropriate view, select a property or dimension and click **Edit**.
3. From the General tab of the Property or Dimension editor, select a language from the **Language** drop-down.
4. Click **OK**.
5. From the File menu, choose **Save**.

The following example shows the Property editor with a property called Descripción. The language for the property is set to Spanish:
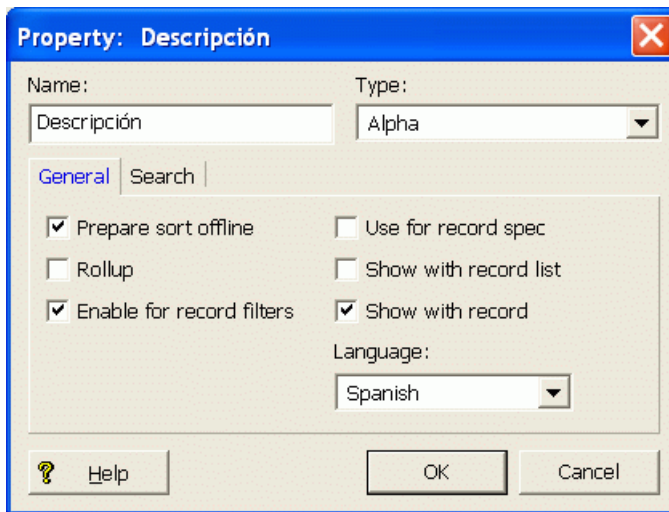
Specifying a per-query language ID

You can use Presentation API calls to specify the language of record queries.

The `ENEQuery` and `UrlENEQuery` classes in the Endeca Presentation API have a Java `setLanguageId()` method and a .NET `LanguageId` property, which you use to tell the MDEX Engine what language record (full-text) queries are in. If you have enabled the language-specific spelling correction feature, a per-query language ID will enable the MDEX Engine to select the appropriate dictionary for a given query.

If no per-query language ID is specified, the MDEX Engine uses the global language ID, which defaults to "en" (US English) if not set specifically.

For details on the `ENEQuery` and `UrlENEQuery` class members, see the *Endeca Presentation API for Java Reference (Javadoc)* or the *Endeca Presentation API for .NET Reference (HTML Help)*.

The following code snippets show how to set French (using its language code of "fr") as the language of any text portion of the query (such as search terms).

Java example of setting a per-query language ID

```
// Create an MDEX Engine query
ENEQuery usq = new UrlENEQuery(request.getQueryString(),"UTF-8");
// Set French as the language for the query
usq.setLanguageId("fr");
// Set other query attributes
...
// Make the request to the MDEX Engine
ENEQueryResults qr = nec.query(usq);
```

.NET example of setting a per-query language ID

```
// Create a query
String queryString = Request.Url.Query.Substring(1);
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");
// Set French as the language for the query
usq.LanguageId = "fr";
// Set other query attributes
...
// Make the request to the Navigation Engine
ENEQueryResults qr = nec.Query(usq);
```

Setting encoding in the front-end application

If you are including internationalized data in your Endeca application, you should be aware of these encoding requirements.

Setting the encoding for URLs

The `UrlENEQuery` and `UrlGen` classes require that you specify a character encoding so that they can properly decode URLs. For example, a URL containing `%E5%8D%83` refers to the Chinese character for "thousand" if using the UTF-8 encoding, but refers to three accented European letters if using the windows-1252 encoding. For details on these classes, see the *Endeca Presentation API Reference (Javadoc)* or the *Endeca Presentation API for .NET Reference (HTML Help)*.

The following code snippets show how to instantiate a `UrlGen` object using the UTF-8 character encoding set.

Java example:

```
// Create request to select refinement value
UrlGen urlg = new UrlGen(request.getQueryString(), "UTF-8");
```

.NET example:

```
// Create request to select refinement value
UrlGen urlg = new UrlGen(Request.Url.Query.Substring(1), "UTF-8");
```

Setting the page encoding

Your application should choose a suitable output encoding for the pages it produces. For example, a multi-lingual European site might choose the windows-1252 encoding, while a Chinese site might choose GB2312 or Big5. If you need to support all languages, Oracle recommends using the UTF-8 encoding.

Viewing MDEX Engine logs

Log messages output by the MDEX Engine binaries are in UTF-8 encoding.

These binaries include Forge, Dgidx, and Dgraph.

Most common UNIX/Linux shells and terminal programs are not set up to display UTF-8 by default and will therefore display some valid characters as question marks (?). If you find unexpected question marks in the data, first validate that it is not simply a display issue. Try the `od` command on Linux, or use a UTF-8 compatible display.

Chapter 17

MDEX Engine Logging Variables

This section describes the MDEX Engine logging variables.

About MDEX Engine logging variables

The MDEX Engine logging variables can be used with the log-enable and log-disable URL config operations to toggle logging verbosity for specified features.

This makes it possible to get detailed information about MDEX Engine processing, to help diagnose unexpected application behavior or performance problems, without stopping and restarting the Dgraph or requiring a configuration update.

Logging variable operation syntax

MDEX Engine logging variables are toggled using the `/config?op=log-enable&name=<variable-name>` and `/config?op=log-disable&name=<variable-name>` operations.

You can include multiple logging variables in a single request. Unrecognized logging variables generate warnings.

For example, this operation:

```
/config?op=log-enable&name=merchverbose
```

turns on verbose logging for the dynamic business rule feature, while this operation:

```
config?op=log-enable&name=textsearchrelrankverbose&name=textsearchspellver¬
bose
```

turns on verbose logging for both the text search relevance ranking and spelling features.

However, this operation:

```
config?op=log-enable&name=allmylogs
```

returns an "unsupported logging setting" message.

In addition, the following operations are supported:

- `/config?op=log-status` returns a list of all logging variables with their values (true or false).
- `/config?op=log-enable` and `/config?op=log-disable` with no arguments return the same thing as `log-status`.

- The special name `all` can be used with `/config?op=log-enable` or `/config?op=log-disable` to set all logging variables.

Supported logging variables

The following table describes the supported logging variables.

Logging variable names are not case sensitive

Variable	Description
verbose	Enables verbose mode.
requestverbose	Prints information about each request to stdout.
updateverbose	Show verbose messages while processing updates.
recordfilterperfverbose	Enables verbose information about record filter performance.
merchverbose	Enables verbose debugging messages during merchandising rule processing.
textsearchrelrankverbose	Enables verbose information about relevance ranking during search query processing.
textsearchspellverbose	Enables verbose output for spelling correction features.
dgraphperfverbose	Enables verbose performance debugging messages during core Dgraph navigation computations.
dgraphrefinementgroupverbose	Enables refinement verbose/debugging messages.

Dgidx Character Mapping

This section lists the character mappings performed by Dgidx.

Diacritical Character to ASCII Character Mapping

The `--diacritic-folding` flag on Dgidx maps accented characters to their simple ASCII equivalent as listed in the table below (characters not listed are not affected by the `--diacritic-folding` option).

Note that capital characters are mapped to lower case equivalents because Endeca search indexing is always case-folded.

ISO Latin1 decimal code	ISO Latin 1 character	ASCII map character	Description
192	À	a	Capital A, grave accent
193	Á	a	Capital A, acute accent
194	Â	a	Capital A, circumflex accent
195	Ã	a	Capital A, tilde
196	Ä	a	Capital A, dieresis or umlaut mark
197	Å	a	Capital A, ring
198	Æ	a	Capital AE diphthong
199	Ç	c	Capital C, cedilla
200	È	e	Capital E, grave accent
201	É	e	Capital E, acute accent
202	Ê	e	Capital E, circumflex accent
203	Ë	e	Capital E, dieresis or umlaut mark
204	Ì	i	Capital I, grave accent
205	Í	i	Capital I, acute accent
206	Î	i	Capital I, circumflex accent
207	Ï	i	Capital I, dieresis or umlaut mark

ISO Latin1 decimal code	ISO Latin 1 character	ASCII map character	Description
208	Đ	e	Capital Eth, Icelandic
209	Ñ	n	Capital N, tilde
210	Ò	o	Capital O, grave accent
211	Ó	o	Capital O, acute accent
212	Ô	o	Capital O, circumflex accent
213	Õ	o	Capital O, tilde
214	Ö	o	Capital O, dieresis or umlaut mark
216	Ø	o	Capital O, slash
217	Ù	u	Capital U, grave accent
218	Ú	u	Capital U, acute accent
219	Û	u	Capital U, circumflex accent
220	Ü	u	Capital U, dieresis or umlaut mark
221	Ý	y	Capital Y, acute accent
222	Þ	p	Capital thorn, Icelandic
223	ß	s	Small sharp s, German
224	à	a	Small a, grave accent
225	á	a	Small a, acute accent
226	â	a	Small a, circumflex accent
227	ã	a	Small a, tilde
228	ä	a	Small a, dieresis or umlaut mark
229	å	a	Small a, ring
230	æ	a	Small ae diphthong
231	ç	c	Small c, cedilla
232	è	e	Small e, grave accent
233	é	e	Small e, acute accent
234	ê	e	Small e, circumflex accent
235	ë	e	Small e, dieresis or umlaut mark
236	ì	i	Small i, grave accent
237	í	i	Small i, acute accent
238	î	i	Small i, circumflex accent
239	ï	i	Small i, dieresis or umlaut mark
240	ð	e	Small eth, Icelandic
241	ñ	n	Small n, tilde

ISO Latin1 decimal code	ISO Latin 1 character	ASCII map character	Description
242	ò	o	Small o, grave accent
243	ó	o	Small o, acute accent
244	ô	o	Small o, circumflex accent
245	õ	o	Small o, tilde
246	ö	o	Small o, dieresis or umlaut mark
248	ø	o	Small o, slash
249	ù	u	Small u, grave accent
250	ú	u	Small u, acute accent
251	û	u	Small u, circumflex accent
252	ü	u	Small u, dieresis or umlaut mark
253	ý	y	Small y, acute accent
254	þ	p	Small thorn, Icelandic
255	ÿ	y	Small y, dieresis or umlaut mark

ISO Latin1 Extended A decimal code	ISO Latin 1 Extended A character	ASCII map character	Description
256		a	Capital A, macron accent
257		a	Small a, macron accent
258		a	Capital A, breve accent
259		a	Small a, breve accent
260		a	Capital A, ogonek accent
261		a	Small a, ogonek accent
262		c	Capital C, acute accent
263		c	Small c, acute accent
264		c	Capital C, circumflex accent
265		c	Small c, circumflex accent
266		c	Capital C, dot accent
267		c	Small c, dot accent
268		c	Capital C, caron accent
269		c	Small c, caron accent
270		d	Capital D, caron accent
271		d	Small d, caron accent
272		d	Capital D, with stroke accent

ISO Latin1 Extended A decimal code	ISO Latin 1 Extended A character	ASCII map character	Description
273		d	Small d, with stroke accent
274		e	Capital E, macron accent
275		e	Small e, macron accent
276		e	Capital E, breve accent
277		e	Small e, breve accent
278		e	Capital E, dot accent
279		e	Small e, dot accent
280		e	Capital E, ogonek accent
281		e	Small e, ogonek accent
282		e	Capital E, caron accent
283		e	Small e, caron accent
284		g	Capital G, circumflex accent
285		g	Small g, circumflex accent
286		g	Capital G, breve accent
287		g	Small g, breve accent
288		g	Capital G, dot accent
289		g	Small g, dot accent
290		g	Capital G, cedilla accent
291		g	Small g, cedilla accent
292		h	Capital H, circumflex accent
293		h	Small h, circumflex accent
294		h	Capital H, with stroke accent
295		h	Small h, with stroke accent
296		i	Capital I, tilde accent
297		i	Small I, tilde accent
298		i	Capital I, macron accent
299		i	Small i, macron accent
300		i	Capital I, breve accent
301		i	Small i, breve accent
302		i	Capital I, ogonek accent
303		i	Small i, ogonek accent
304		i	Capital I, dot accent

ISO Latin1 Extended A decimal code	ISO Latin 1 Extended A character	ASCII map character	Description
305	ı	i	Small dotless i
306		i	Capital ligature IJ
307		i	Small ligature IJ
308		j	Capital J, circumflex accent
309		j	Small j, circumflex accent
310		k	Capital K, cedilla accent
311		k	Small k, cedilla accent
312		k	Small Kra
313		l	Capital L, acute accent
314		l	Small l, acute accent
315		l	Capital L, cedilla accent
316		l	Small l, cedilla accent
317		l	Capital L, caron accent
318		l	Small L, caron accent
319		l	Capital L, middle dot accent
320		l	Small l, middle dot accent
321	Ł	l	Capital L, with stroke accent
322	ł	l	Small l, with stroke accent
323		n	Capital N, acute accent
324		n	Small n, acute accent
325		n	Capital N, cedilla accent
326		n	Small n, cedilla accent
327		n	Capital N, caron accent
328		n	Small n, caron accent
329		n	Small N, preceded by apostrophe
330		n	Capital Eng
331		n	Small Eng
332		o	Capital O, macron accent
333		o	Small o, macron accent
334		o	Capital O, breve accent
335		o	Small o, breve accent
336		o	Capital O, with double acute accent

ISO Latin1 Extended A decimal code	ISO Latin 1 Extended A character	ASCII map character	Description
337		o	Small O, with double acute accent
338	Œ	o	Capital Ligature OE
339	œ	o	Small Ligature OE
340		r	Capital R, acute accent
341		r	Small R, acute accent
342		r	Capital R, cedilla accent
343		r	Small r, cedilla accent
344		r	Capital R, caron accent
345		r	Small r, caron accent
346		s	Capital S, acute accent
347		s	Small s, acute accent
348		s	Capital S, circumflex accent
349		s	Small s, circumflex accent
350		s	Capital S, cedilla accent
351		s	Small s, cedilla accent
352	Š	s	Capital S, caron accent
353	š	s	Small s, caron accent
354		t	Capital T, cedilla accent
355		t	Small t, cedilla accent
356		t	Capital T, caron accent
357		t	Small t, caron accent
358		t	Capital T, with stroke accent
359		t	Small t, with stroke accent
360		u	Capital U, tilde accent
361		u	Small u, tilde accent
362		u	Capital U, macron accent
363		u	Small u, macron accent
364		u	Capital U, breve accent
365		u	Small u, breve accent
366		u	Capital U with ring above
367		u	Small u with ring above
368		u	Capital U, double acute accent

ISO Latin1 Extended A decimal code	ISO Latin 1 Extended A character	ASCII map character	Description
369		u	Small u, double acute accent
370		u	Capital U, ogonek accent
371		u	Small u, ogonek accent
372		w	Capital W, circumflex accent
373		w	Small w, circumflex accent
374		y	Capital Y, circumflex accent
375		y	Small y, circumflex accent
376	Ÿ	y	Capital Y, diaeresis accent
377		z	Capital Z, acute accent
378		z	Small z, acute accent
379		z	Capital Z, dot accent
380		z	Small Z, dot accent
381	Ž	z	Capital Z, caron accent
382	ž	z	Small z, caron accent
383		s	Small long s

Index