Endeca® Latitude

Developer's Guide

Version 2.1.0 • June 2011

Contents

Copyright and disclaimer

Preface

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

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

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

About this guide

This guide describes the core features of the Endeca MDEX Engine that you can access via applications built with Latitude Studio. It covers working with records and attributes, and search configuration. In addition, this guide describes the Endeca Conversation Web Service APIs that are used by Latitude Studio for querying the MDEX Engine.

Who should use this guide

This guide is intended for developers who are building applications based on Endeca® Latitude.

Conventions used in this guide

This guide uses the following typographical conventions:

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

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

Contacting Endeca Customer Support

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

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

Part 1

Introduction

Chapter 1

MDEX Engine Interfaces

The MDEX Engine is the backbone of all applications running on top of Endeca Latitude software. The Endeca Web services provide the interfaces to the Endeca MDEX Engine. You use them through Latitude Studio components to query the MDEX Engine and manipulate the query results.

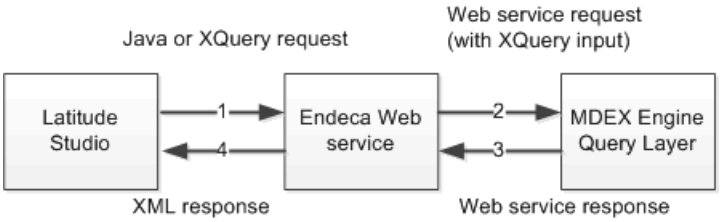
The data flow

In a typical Latitude application, the MDEX Engine communicates with the Web application using the Endeca Web services.

The online portion of a typical Endeca implementation has the following components:

- The MDEX Engine, which receives and processes query requests.
- Latitude Studio components, which you use to query the MDEX Engine and manipulate the query results.
- A Web application in the form of a set of application modules, which receive client requests and pass them to the MDEX Engine through Latitude Studio.

The following diagram illustrates the data flow between these components for a typical Endeca-based application that uses Latitude Studio, Endeca Web services, and the XQuery API:



In this diagram, the following actions take place:

1. A client browser makes a request. The Web application server receives the request and passes it to Latitude Studio.
2. Latitude Studio components use the Endeca Web services to pass the request to the Endeca MDEX Engine, utilizing the XQuery API.
3. The MDEX Engine executes the query and returns its results.
4. Endeca Web services retrieve and manipulate the query results and transfer them in XML format to the Latitude Studio application. The application in Latitude Studio performs formatting of the query results and returns them to the client browser, via the Web application server.

Note: For security reasons, you should never allow Web browsers to connect directly to your MDEX Engine. Browsers should always connect to your application through an application server.

MDEX Engine overview

The Endeca MDEX Engine is the indexing and query engine that provides the foundation for all Endeca solutions.

The MDEX Engine uses proprietary data structures and algorithms that allow it to provide real-time responses to client requests. The MDEX Engine stores the indices that were created after your source data is ingested. After the indices are stored, the MDEX Engine receives client requests via the application tier, queries the indices, and then returns the results.

The MDEX Engine is designed to be stateless. This design requires that a complete query be sent to the MDEX Engine for each request. The stateless design of the MDEX Engine facilitates the addition of MDEX Engine servers for high availability, load balancing and redundancy. Because the MDEX Engine is stateless, any replica of an MDEX Engine on one server can reply to queries independently of a replica on other MDEX Engine servers.

Consequently, adding replicas of MDEX Engines on additional servers provides redundancy and improved query response time. That is, if any one particular server goes down, a replica of an MDEX Engine provides redundancy by allowing other servers in the implementation to continue to reply to queries. In addition, total response time is improved by using load balancers to distribute queries to a replica MDEX Engine on any of the additional servers. If a replica of the MDEX Engine is configured as part of a cluster, this provides high availability of request processing by ensuring that at least one of the MDEX instances running on one of the nodes in the cluster continues to process queries.

The Dgraph is the name of the process for the MDEX Engine. Because the Dgraph is key to every Endeca implementation, its performance is critical. A typical Endeca implementation includes one or more Dgraphs.

Basic queries

While the queries you send to an Endeca MDEX Engine can become quite complex, you should be familiar with a few basic queries.

Listed below are these queries and the response types they return. Note that the Conversation Web Service returns all of the query results in a `Results` type:

Query	Returned type
Navigation query	`NavigationMenu`
Endeca record query	`RecordList`
value search query	`ValueSearch`

Endeca Web services

The MDEX Engine is installed with a set of Web services for loading, configuring, and querying the data. These Web services provide the API to the MDEX Engine.

List of Web services installed with the MDEX Engine

The Web services installed with the MDEX Engine are:

- Data Ingest Web Service (Documented in the *Latitude Data Ingest API Guide*)
- Configuration Web Service
- Conversation Web Service (An internal interface that is subject to change, used by Latitude Studio)
- Configuration Web Service (read-only version)
- Administration Web Service (Documented in the *Latitude Administrator's Guide*)
- MDEX Web Service (For internal use only by the MDEX Engine. It is not used to interact directly with the MDEX Engine.)

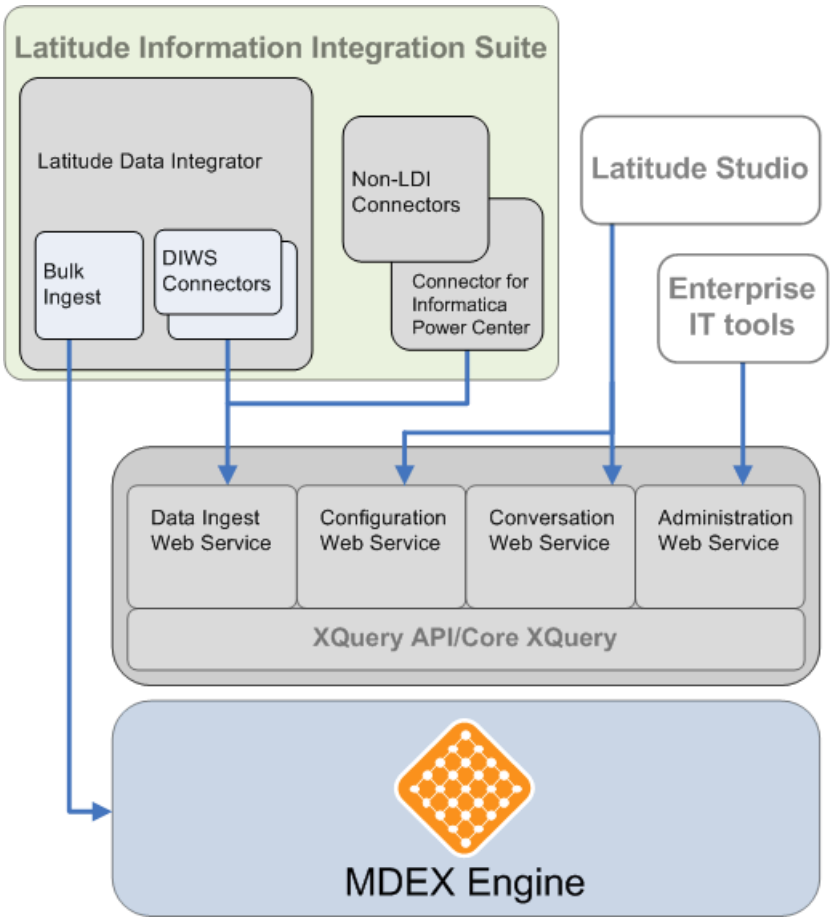
Flow for using the Web services

As you build your application, you use these Web services roughly as follows:

1. Use the Latitude Information Integration Suite to load data into the MDEX Engine. The Latitude Information Integration Suite contains connectors that use the Data Ingest Web Service to communicate with the MDEX Engine.
2. Use the Configuration Web Service to configure the record schema and MDEX Engine features.
3. Set up the front-end application using Latitude Studio. Latitude Studio uses the Conversation Web Service to communicate with the MDEX Engine.
4. Use the Administration Web Service to set up monitoring and backups.

How Web services correspond to Latitude modules

The following diagram lists the Endeca Web services and shows their position in relation to Latitude modules:



How each Web service interacts with the MDEX Engine

Each Endeca Web service can be described in the context of how it interacts with the MDEX Engine.

Web service	Function
Data Ingest Service	The Data Ingest Web Service is used to load data into the MDEX Engine. It serves as the basis for various batch processes. It is designed for easy integration with ETL tools.
Configuration Web Service	The Configuration Web Service supports the process of refining the records schema and adjusting your configuration in the development environment. **Note:** The read-only version of the Configuration Web Service is used for read-only services, such as providing information about the records schema. This version cannot be used for any updating operations to the schema or configuration.
Conversation Web Service	The Conversation Web Service is used to query the MDEX Engine and to provide summarizations. It is used by Latitude Studio.

Web service	Function
	Unlike the Data Ingest and Configuration Web Services, it is not used to update the MDEX Engine index. For example, you can use the Conversation Web Service to search for records, where you obtain a list of attributes from the MDEX Engine, but you cannot use the Conversation Service to make changes to an attribute.
Administration Web Service	The Administration Web Service is used by IT engineers and administrators to integrate the MDEX Engine server and its reporting with third-party IT tools. The Administration Web Service interacts with the MDEX Engine server outside of the MDEX Engine data layer. The other Endeca Web services interact with the indexes within the MDEX Engine data layer.

Latitude Studio components and the MDEX Engine

This topic maps each Latitude Studio component supported in this release to the MDEX Engine features that provide the backbone for the component.

Latitude Studio component	MDEX Engine features
Guided Navigation	Working with refinements, value search, working with attribute groups
Search Box	All record search configuration features, such as search interfaces, making records or attributes searchable, wildcard search, match modes, record search, value search
Breadcrumbs	Working with refinements, record search, DYM (Did You Mean) and auto-correction
Data Explorer	Working with records
Results List	Working with records, working with refinements
Results Table	Working with records, working with refinements, snippeting
Record Details	Working with records, working with attribute groups
Chart	Latitude Query Language in the MDEX Engine
Cross Tab	Latitude Query Language in the MDEX Engine, working with refinements
Metrics Bar	Latitude Query Language in the MDEX Engine
Alerts	Latitude Query Language in the MDEX Engine, working with refinements
Compare	Working with records, working with refinements
Tag Cloud	Working with refinements

About the MDEX Engine API Reference

This automatically generated reference provides information about Web services that are packaged with the MDEX Engine.

The *MDEX Engine API Reference* is the documentation generated from the two types of files that describe a Web service:

- A WSDL document
- An XML Schema definition (XSD)

The *MDEX Engine API Reference* is complemented by the *Developer's Guide* (this guide).

The *MDEX Engine API Reference* is located in the `doc` directory of the MDEX Engine installation.

MDEX Engine configuration items

You can use the Latitude Data Integrator to modify system records and configuration documents.

Using the Latitude Data Integrator, you can modify the following global configuration items:

- Dimension Description Records
- Property Description Records
- Global Configuration Record
- dimsearch_config
- precedence_rules
- recsearch_config
- relrank_strategies
- stop_words
- thesaurus

For information on using the Latitude Data Integrator for schema and configuration changes, and for information on XML configuration documents, see the *Latitude Data Integrator Guide*.

Chapter 2

MDEX Engine Concepts

This section introduces basic concepts associated with the MDEX Engine and how data is structured and configured in the MDEX Engine data model.

MDEX Engine data model

The data model in the MDEX Engine consists of records and attributes.

- Records are the fundamental units of data.
- Attributes are the fundamental units of the schema. For each attribute, a record may be assigned one or more attribute values.

Records

Records are the fundamental units of data in the MDEX Engine. Almost all information that is consumed by the MDEX Engine, including raw data and the data schema, is represented by records.

The MDEX Engine includes the following types of records:

Record type	Description
Source records	The data that is input into the Endeca Latitude application. Latitude supports source records in a variety of formats.
Data records	In most applications, you are primarily concerned with data records. Data records are the business records you want to explore using your Latitude application.
Primordial records	*Primordial records* are created automatically and used internally by the MDEX Engine. They represent the most basic infrastructure of an MDEX Engine.

Record type	Description
System records	*System records* represent the record schema. They are created in the MDEX Engine using the schema from the primordial records. You use these records for data modeling—changing these records controls the behavior of your records schema and thus affects your data model.

Attributes

An *attribute* is the basic unit of a record schema. Attributes describe records in the MDEX Engine.

For a data record, an attribute provides information about that record. For example, for a list of books, the Author attribute contains the author of the book.

For an attribute schema record, an attribute provides configuration information. For example, each attribute is itself represented by an attribute schema record. Each attribute in that attribute schema record controls an aspect of the attribute, such as the format of the attribute values and whether the attribute can be used for searches.

The term attribute collectively refers to both standard attributes and managed attributes.

- Standard attributes are described by system records. The system records that describe standard attributes are known as Property Description Records (PDRs).
- Managed attributes are also described by system records. The system records that describe managed attributes are known as Property Description Records (PDRs) and Dimension Description Records (DDRs).

Each attribute is identified by a unique name.

Related Links

Assignments on standard attributes on page 22
Records are assigned standard attribute values. An *assignment* indicates that a record has a value for a standard attribute.

Unique attributes on page 23
A *unique attribute* is a standard attribute for which each record must have a unique value.

Primary key attributes on page 23
Each set of records must have at least one primary key standard attribute.

Attribute types on page 23
The attribute type identifies the type of data allowed for the standard attribute value (key value pair).

Assignments on standard attributes

Records are assigned standard attribute values. An *assignment* indicates that a record has a value for a standard attribute.

A record typically has assignments for multiple standard attributes. For each assigned attribute, the record may have one or more values. An assignment on a standard attribute is known as a *key value pair (KVP)*.

Not all standard attributes will have an assignment for every record. For example, for a publisher that sells both books and magazines, the "ISBN number" standard attribute would be assigned for book records, but not assigned (empty) for magazine records.

Standard attributes may be single-assign or multi-assign:

- A single-assign attribute is an attribute for which each record can only have one value. For example, for a list of books, the ISBN number would be a single-assign attribute. Each book only has one ISBN number.
- A multi-assign attribute is an attribute for which a single record can have more than one value. For the same list of books, because a single book may have multiple authors, the Author attribute would be a multi-assign attribute.

By default, all standard attributes are single-assign. To make a standard attribute multi-assign, you must update the attribute configuration.

Unique attributes

A *unique attribute* is a standard attribute for which each record must have a unique value.

For the MDEX Engine to identify a record to update, the record must have at least one assignment from a unique attribute.

By default, a standard attribute is not unique. To make a standard attribute unique, you must update the standard attribute configuration.

A schema can have more than one unique attribute. This allows the MDEX Engine to handle different record "buckets," each of which can have a meaningful identifying standard attribute.

For example, a store carries multiple types of printed publications such as books, magazines, or newspapers. The records representing books can have a unique attribute, ISBN. They may also have another unique attribute, publication-id, that is used for all publication types. You could then identify a specific book by providing to the MDEX Engine either its ISBN or its publication-id.

Primary key attributes

Each set of records must have at least one primary key standard attribute.

Primary key attributes are used to uniquely identify a record. In order for a standard attribute to be a primary key attribute:

- The attribute must be unique
- The attribute must be single-assign

You can use more than one attribute in the MDEX Engine as a primary key, as long as each attribute is both unique and single-assign.

Attribute types

The attribute type identifies the type of data allowed for the standard attribute value (key value pair).

The MDEX Engine supports the following standard attribute types:

Attribute type	Description
`mdex:string`	XML-valid character strings.
`mdex:int`	A 32-bit signed integer. `mdex:int` values accepted by the MDEX Engine on all platforms can be up to the value of 2,147,483,647.

Attribute type	Description
`mdex:long`	A 64-bit signed integer. `mdex:long` values accepted by the MDEX Engine on all platforms can be up to the value of 9,223,372,036,854,775,807.
`mdex:double`	A floating point value.
`mdex:time`	A 32-bit unsigned integer that represents the time of day in milliseconds.
`mdex:dateTime`	A 64-bit signed integer that represents the date and time in milliseconds since the epoch (January 1, 1970).
`mdex:duration`	A 64-bit signed integer that represents a length of time in milliseconds.
`mdex:boolean`	A Boolean. Valid Boolean values are `true` (or `1`, which is a synonym for `true`) and `false` (or `0`, which is a synonym for `false`).
`mdex:geocode`	A latitude and longitude pair. The latitude and longitude are both double-precision floating-point values.

XML representation of records and attributes

In XML, each record is represented as a collection of attribute value assignments (key value pairs).

In all of the MDEX Engine interfaces, a record is represented in XML as a record element. The record element contains attribute elements (these attributes should not be confused with the term "attribute" used in the XML standard set of terms). Each attribute element contains the attribute values for the specified attribute.

If a record does not have a value for an attribute, the attribute is not included for that record.

If a record has multiple values for an attribute, there is a separate attribute element for each value.

The following XML represents a single data record with three standard attributes (`Description`, `WineID`, and `WineType`):

```
<Record>
<Description type="mdex:string">Dense and vegetal, with peach and
spice flavors.</Description>
<WineID type="mdex:int">12345</WineID>
<WineType type="mdex:string">white</WineType>
</Record>
```

Examples of records and standard attributes

The following examples of records demonstrate different configurations of standard attributes and their values (key value pairs).

About these examples

In the examples, each row in the table represents a single record, in this case, a bottle of wine. The column headings are standard attributes, and each cell contains a standard attribute value (key value pair).

Example 1: all records have a single assignment from each attribute

In this example:

- The Name attribute is both unique and single-assign. Each record has exactly one assignment on the Name attribute, and the Name attribute is unique across the data set.
- No records have multiple assignments.
- Every record has an assignment for every attribute.

Name	Wine Type	Wine ID	Region	Vintage	Price
Ravenswood Merlot 2007	Merlot	4038	California	2007	$20.00
Veuve Cliquot 2001	Champagne	5213	France	2001	$50.00
2008 Elk Cove Pinot Gris	Pinot Gris	8765	Oregon	2008	$16.99
Yellow Tail Shiraz 2008	Shiraz	4035	Australia	2008	$6.99
Concha y Toro Casillero Del Diablo Cabernet Sauvignon 2008	Cabernet Sauvignon	3421	Chile	2008	$8.99

The XML representation of the Ravenswood wine record may look similar to the following example:

```
<Record>
<Name type="mdex:string">Ravenswood Merlot 2007</Description>
<WineID type="mdex:int">4038</WineID>
<WineType type="mdex:string">Merlot</WineType>
<Region type="mdex:string">California</Region>
<Vintage type="mdex:int">2007</Vintage>
<Price type="mdex:string">20.00</Price>
</Record>
```

All lines in this XML example represent key value pairs.

Also, notice the primary key attribute, which in this case is the `WineID`. This primary key attribute is used by the MDEX Engine to uniquely identify this record. At the data loading stage you decide which of your standard attributes is going to be the primary key attribute.

Example 2: records with no assignments and with multiple assignments on an attribute

This example uses the same data as the previous example, but adds a Review score attribute. For the Review score attribute, some records have multiple assignments, and some have no assignments.

For example, the Ravenswood record has multiple review scores, and the Yellow Tail record has no review scores.

Name	Wine Type	Region	Review score	Vintage	Price
Ravenswood Merlot 2007	Merlot	California	20, 35, 40	2007	$20.00
Veuve Cliquot 2001	Champagne	France	80, 82	2001	$50.00
2008 Elk Cove Pinot Gris	Pinot Gris	Oregon		2008	$16.99
Yellow Tail Shiraz 2008	Shiraz	Australia		2008	$6.99
Concha y Toro Casillero Del Diablo Cabernet Sauvignon 2008	Cabernet Sauvignon	Chile		2008	$8.99

The XML representation of the Ravenswood and Yellow Tail wines may look similar to the following example:

```
<Record>
<Name type="mdex:string">Ravenswood Merlot 2007</Description>
<WineType type="mdex:string">Merlot</WineType>
<Region type="mdex:string">California</Region>
<ReviewScore type="mdex:int">20</ReviewScore>
<ReviewScore type="mdex:int">35</ReviewScore>
<ReviewScore type="mdex:int">40</ReviewScore>
<Vintage type="mdex:int">2007</Vintage>
<Price type="mdex:string">20.00</Price>
</Record>
<Record>
<Name type="mdex:string">Yellow Tail Shiraz 2008</Description>
<WineType type="mdex:string">Shiraz</WineType>
<Region type="mdex:string">Australia</Region>
<Vintage type="mdex:int">2008</Vintage>
<Price type="mdex:string">6.99</Price>
</Record>
```

The XML for the Ravenswood record contains three `ReviewScore` elements, one for each score. Because the Yellow Tail record does not have any review scores, it does not include a `ReviewScore` element.

Managed attributes

A *managed attribute* is a collection of metadata about standard attribute values, most importantly the hierarchy of those attribute values (key value pairs).

Managed attributes are used to support hierarchical navigation. In other words, associating a managed attribute with a standard attribute enables hierarchical navigation of records based on the standard attribute values. For example, you can navigate a collection of books using the Library of Congress Classification standard attribute and refine by Literature > American > 19th century.

When you create a managed attribute, you load a taxonomy definition that enumerates a hierarchy where each standard attribute value (in a key value pair for the standard attribute) is a node in the hierarchy (called a *managed attribute value, or mval*).

Managed attributes are described by system records — Property Dimension Records and Dimension Description Records.

Managed attributes:

- Must share a name with existing standard attributes.
- Can be made available for navigation.
- Can be made available for search.
- Have a defined hierarchy. Managed attribute values in this hierarchy are represented by standard attribute values.

It is important to understand that managed attributes do not exist independently of the standard attributes whose hierarchy they describe. Managed attributes simply define additional metadata for those standard attributes.

In other words, you must first define records and their standard attributes to the MDEX Engine. Only then can you add managed attribute hierarchies for selected standard attributes. Together, the records, standard attributes, and managed attributes make up your *records schema*.

System records

The MDEX Engine uses *system records* to store configuration information.

You can configure the following system records:

- **Property Description Records (PDRs)**, used to define the format and behavior of standard attributes.
- **Dimension Description Records (DDRs)**, used to define managed attributes and thus enable the creation of hierarchical standard attribute values.
- The **Global Configuration Record (GCR)**, used to control various aspects of the global configuration.

To avoid naming collisions with customer-created records and attributes, the keys for system records are prefixed with MDEX-specific namespaces, such as `mdex-property`.

Related Links

Property Description Record (PDR) on page 27
A *Property Description Record (PDR)* is a system record that defines a record for a standard attribute in the MDEX Engine.

Dimension Description Record (DDR) on page 30
A *Dimension Description Record (DDR)* defines a managed attribute, similar to how a PDR defines a standard attribute.

Global Configuration Record (GCR) on page 31
The *Global Configuration Record* (GCR) is a single record used to identify and store global configuration information.

Property Description Record (PDR)

A *Property Description Record (PDR)* is a system record that defines a record for a standard attribute in the MDEX Engine.

About PDRs

The MDEX Engine uses a PDR to store metadata about the standard attribute, and must have a PDR in order to build a schema for your data records.

As records, PDRs themselves have required attributes, and can also have arbitrary, user-defined attributes.

For each standard attribute, the attributes in the associated PDR define the attribute's features, including:

- Name and type
- Display name
- Configuration parameters. For example, whether an attribute is searchable.
- Navigability settings. For example, whether to show record counts for available refinements, whether to enable multi-select, and how to sort refinements.

Creating and updating PDRs

When the MDEX Engine acquires a new record, it stores it and constructs a PDR for any attributes that it finds in the record. To create a PDR, use the Latitude Data Integrator.

Updating a PDR immediately changes the navigation behavior of the MDEX Engine. To change a PDR, you can use:

- The Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.
- The Data Ingest Web Service.

Required schema attributes of a PDR

PDRs have the following required attributes:

Schema attribute	Type	Description
`mdex-property_Key`	`string`	The name of the standard attribute. The key name must be an NCName.
`mdex-property_DisplayName`	`string`	The name of the standard attribute in an easy-to-understand format. The display name can use a non-NCName format.
`mdex-property_Type`	`string`	The data type of the standard attribute. The possible values are `string`, `int`, `int64`, `double`, `boolean`, `dateTime`, `time`, `duration`, and `geocode`. The default is `string`.
`mdex-property_IsSingleAssign`	`boolean`	If set to `true`, each record can have at most one value for the standard attribute. If set to `false`, each record may have more than one value for the standard attribute. The default is `true`.
`mdex-property_IsUnique`	`boolean`	If set to `true`, then each record must have a unique value for the standard attribute. If set to `false`, then multiple records can have the same value. The default is `false`.
`mdex-property_IsTextSearchable`	`boolean`	If set to `true`, then the standard attribute is enabled for text search. If set to `false`, the standard attribute does not support text search. The default is `false`.
`mdex-property_TextSearchAllowsWildcards`	`boolean`	If set to `true`, then wildcard search is enabled for this standard attribute. If set to `false`, then wildcard search is not enabled.

Schema attribute	Type	Description
		If this is set to `true`, then `mdex-property_IsTextSearchable` must be set to `false`. The default is `false`.
`mdex-property_IsPropertyValueSearchable`	`boolean`	If set to `true`, the standard attribute is enabled for value search. If set to `false`, the attribute is not value-searchable. The default is `true`. This schema attribute can be changed only for the standard attributes of type string. This schema attribute does not apply for managed attributes, for which value search is always enabled and cannot be disabled.
`system-navigation_Select`	`string`	Used to configure the multi-select feature for a standard attribute. The allowed values are: • `single`. Users can select only one refinement from this attribute. • `multi-and`. Users can select multiple refinements from the attribute. The returned records must have assignments from all of the selected refinements (from A AND B). • `multi-or`. Users can select multiple refinements from this attribute. The returned records must have assignments from at least one of the selected refinements (from A OR B). The default is `single`.
`system-navigation_Sorting`	`string`	The order in which to display refinements in the navigation menu. The allowed values are: • `lexical` sorts refinements alphabetically or by number. • `record-count` sorts refinements in descending order, by the number of records available for each refinement. The default is `lexical`.
`system-navigation_ShowRecordCounts`	`boolean`	Whether to show record counts for a refinement. If set to `true`, the record counts are shown.

Schema attribute	Type	Description
		If set to `false`, the record counts are not shown. The default is `true`.
`system-property_GroupMember¬ship`	`string`	The groups to which the attributes belong.

User-defined schema attributes of a PDR

You can use the arbitrary, user-defined schema attributes in a Property Description Record to display various aspects of how your data records are organized.

Dimension Description Record (DDR)

A *Dimension Description Record (DDR)* defines a managed attribute, similar to how a PDR defines a standard attribute.

About DDRs

The Dimension Description Record has the same name as the associated standard attribute. It used to enable the creation of hierarchical standard attribute values.

Required schema attributes of a DDR

A Dimension Description Record has the following required schema attributes:

Schema attributes	Type	Description
`mdex-dimension_Key`	`string`	The name of the managed attribute.
`mdex-dimension_EnableRefinements`	`boolean`	If set to `true`, then refinements are displayed. If set to `false`, refinements are not displayed. In other words, the managed attribute is hidden. The default is `true`.
`mdex-dimension_IsDimensionSearch¬Hierarchical`	`boolean`	If set to `true`, then during value searches, the search matches both the assigned values and the ancestors of those values. If set to `false`, then the search matches only the assigned values. The default is `false`.
`mdex-dimension_IsRecordSearchHier¬archical`	`boolean`	If set to `true`, then during record searches, the search matches

Schema attributes	Type	Description
		records with both the assigned values and the ancestors of those values. If set to `false`, then the search only matches records with the assigned values. The default is `false`.

Global Configuration Record (GCR)

The *Global Configuration Record* (GCR) is a single record used to identify and store global configuration information.

Definition

The Global Configuration Record is created automatically in the MDEX Engine, and can be modified if needed. This information persists if you restart the MDEX Engine.

The Global Configuration Record controls the following areas of the MDEX Engine configuration:

Area of global configuration	What you can do...
Wildcard search enablement	Specify whether wildcard search should be enabled or disabled for value search in the MDEX Engine. By default, it is disabled.
Search characters	List which characters you want to identify as search characters to the MDEX Engine.
Merge policy	Optionally, change the policy that the MDEX Engine uses in the background to merge its index generations. The default policy – `balanced` – is recommended, and is optimized for best performance.
Spelling correction settings	Control which words are eligible for the spelling dictionary. To do this, for each attribute assignment on a record and for each managed attribute value, you specify the following parameters: • Minimum word occurrence • Minimum word length • Maximum word length  **Note:** If you change the spelling settings in the Global Configuration Record, you must run the `admin?op=updatea¬spell` command in order for them to take effect.

Modifying the settings in the Global Configuration Record

To change the Global Configuration Record settings, use the Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.

Required attributes of the GCR

The Global Configuration Record has required attributes, but it cannot have arbitrary, user-defined attributes.

The required attributes are:

Attribute	Type	Description
`mdex-config_Key`	`String`	The only value for this attribute is `global`. This attribute is unique and single-assign.
`mdex-config_EnableValueSearch¬Wildcard`	`Boolean`	If set to `true`, then wildcard search is enabled for value search. If set to `false`, then wildcard search is disabled. The default value is `false`.
`mdex-config_MergePolicy`	`String`	The allowed values are `balanced` or `aggressive`. The default is `balanced`.
`mdex-config_SearchChars`	`String`	The characters to use as search characters in the MDEX Engine. The allowed values are strings that are listed sequentially and are not separated by commas or spaces. Each string is a search character.
`mdex-config_SystemRecordVer¬sion`	`String`	The version of the system records in the MDEX Engine. This attribute is used by the MDEX Engine and should not be modified.
`mdex-config_SpellingRecordMin¬WordOccur`	`Int`	The minimum number of times a word must occur in a standard attribute value (record assignment on a standard attribute, in a key value pair) for it to be indexed for spelling correction. The default value is 4.
`mdex-config_SpellingRecordMin¬WordLength`	`Int`	The minimum number of characters that a word can contain in a standard attribute value for it to be indexed for spelling correction. The default value is 3.

Attribute	Type	Description
mdex-config_SpellingRecordMax¬WordLength	Int	The maximum number of characters that a word can contain in a standard attribute value for it to be indexed for spelling correction. The default value is 16.
mdex-config_SpellingDValMinWor¬dOccur	Int	The minimum number of times a word must occur in a managed attribute value for it to be indexed for spelling correction. The default value is 1.
mdex-config_SpellingDValMin¬WordLength	Int	The minimum number of characters that a word must contain in a managed attribute value for it to be indexed for spelling correction. The default value is 3.
mdex-config_SpellingDValMax¬WordLength	Int	The maximum number of characters that a word may contain in a managed attribute value for it to be indexed for spelling correction. The default value is 16.

Validating the GCR settings

During record updates, the MDEX Engine validation process validates the configuration of the Global Configuration Record, and returns errors if its requirements are not met.

The requirements are as follows:

- The `mdex-config_Key` attribute must be unique and single-assign. The value must be `global`.
- The Global Configuration Record must contain valid allowable values for all of its attributes. None of its attributes can be omitted.
- The Global Configuration Record cannot have any arbitrary, user-defined attributes.

Part 2

Endeca Web Services

Chapter 3

Using the Configuration Web Service

This section describes the Configuration Web Service.

About the Configuration Web Service

The Configuration Web Service provides an interface that allows ergonomic interaction with both the MDEX Engine configuration and record schema.

Overview

The Configuration Web Service is used by the Latitude Data Integrator, and allows you to manipulate schema and configuration.

Two versions of the Configuration Web Service exist:

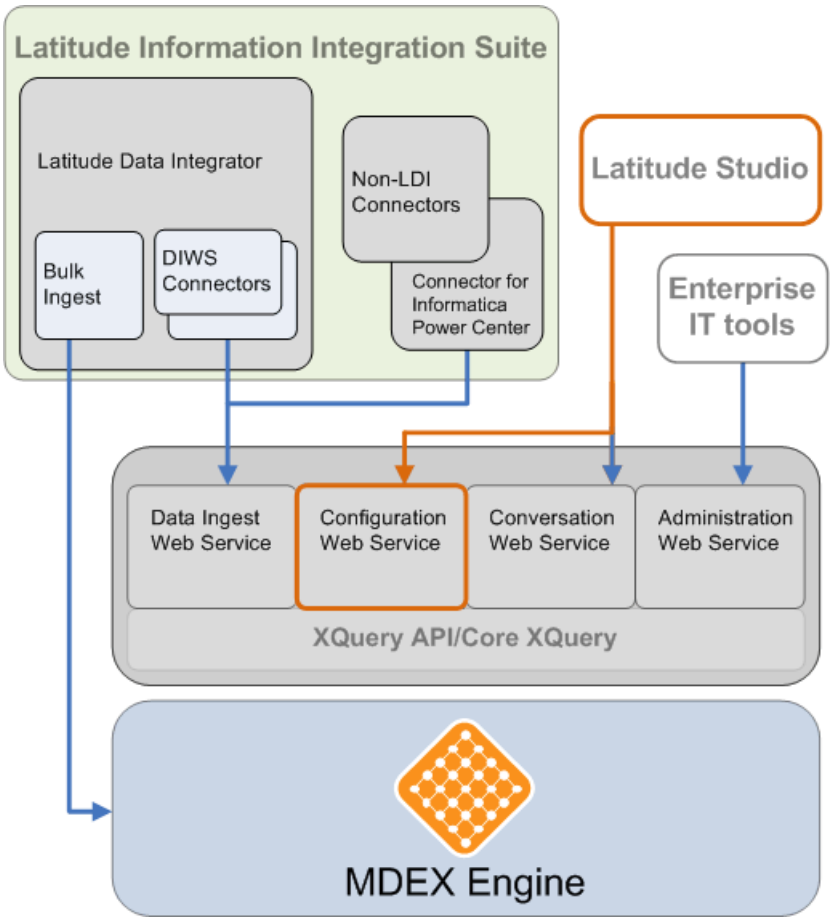
- The full-featured Configuration Web Service
- The read-only Configuration Web Service

The full-featured Configuration Web Service is declared in `config.wsdl`. You can access it at this URL: `http://localhost:<port>/ws/config`, similar to other packaged Web services.

The read-only version of the Configuration Web Service is declared in `config_read_only.wsdl`. This version is a subset of a full-featured version. The read-only version contains only the operations that read from the current schema or configuration; it does not contain operations that update the schema or configuration. This read-only version of the Configuration Web Service is used by Latitude Studio in cases when a server running Latitude Studio must access the server running the MDEX Engine in a read-only mode (such configuration is possible when multiple MDEX Engine servers are running in a cluster. For more information on the cluster of MDEX Engine nodes, see the *Latitude Administrator's Guide*).

Note: When the documentation mentions the Configuration Web Service, it refers to the full-featured version of the service. When the read-only version is mentioned, it is specifically characterized as the "read-only version of the Configuration Web Service".

The following diagram shows how the Configuration Web Service fits into the larger picture of packaged Web services that serve the MDEX Engine:



As shown in this diagram, the Configuration Web Service is used by Latitude Studio during the process of defining the Endeca application.

Function description

A request to the Configuration Web Service consists of a `configTransaction` element, which contains a series of operations. Operations can be combined arbitrarily in a single service request; each of the operations can appear at most once. The operations perform actions on PDRs, DDRs, groups, the GCR, and on XML configuration documents.

The effect of a full-featured Configuration Web Service request that contains `put` operations is to add attributes, XML configuration documents, or the Global Configuration Record to the MDEX Engine:

- If a record with the specified key already exists in the MDEX Engine, it is replaced.
- If a record does not exist, it is created.

Request

The input to the Configuration Web Service depends on the operation used. It can include attribute schema records (PDRs and DDRs), Global Configuration Record, groups, and a set of XML configuration documents.

Response

Not all operations in the Configuration Web Service return data.

If the operation returns data, the response to the Configuration Web Service is a results element, within which each of the submitted operations produces an element showing its own results.

If any operation does not succeed, the whole Web service transaction returns a SOAP fault and none of the operations are applied.

Operations for PDRs

The operations on PDRs are the following:

Operation	Description
`exportProperties`	Return all Property Description Records (PDRs)
`listProperties`	Return the key of each standard attribute present in the MDEX Engine
`getProperties`	Return PDRs for the specified attribute keys. Attribute keys are obtained from `listProperties`
`putProperties`	Add the PDRs (specified as an argument) to the MDEX Engine. If an with the same key exists, it is replaced.  **Note:** This operation is not supported by the read-only version of the service.
`updateProperties`	Lets you add or modify specified assignments on the PDR. As an argument, specify an attribute key associated with an existing PDR and zero or more assignments. The operation replaces the assignment on the PDR with a new assignment if it is provided as an argument. There is no requirement to specify the entire PDR to this operation; there is a requirement to specify the standard attribute key.  **Note:** This operation is not supported by the read-only version of the service.

Operations for DDRs

The operations on DDRs are the following:

Operation	Description
`exportDimensions`	Return all Dimension Description Records
`listDimensions`	Return the key of each managed attribute present in the MDEX Engine
`getDimensions`	Return DDRs for specified managed attribute keys. Managed attribute keys are obtained from `listDimensions`
`putDimensions`	Add the DDRs (specified as arguments) to the MDEX Engine. If a managed attribute with the same key exists, it is replaced.  **Note:** This operation is not supported by the read-only version of the service.

Operation	Description
`updateDimensions`	Lets you add or modify specified assignments on the DDR. As an argument, specify a managed attribute key associated with an existing DDR and zero or more assignments. The operation replaces the assignment on the DDR with a new assignment if it is provided as an argument. There is no requirement to specify the entire DDR to this operation; there is a requirement to specify the managed attribute key.  **Note:** This operation is not supported by the read-only version of the service.

Group operations

The operations on refinement groups are the following:

Operation	Description
`importGroups`	Remove any existing groups and add the specified ones  **Note:** This operation is not supported by the read-only version of the service.
`exportGroups`	Return the full representation of each group
`listGroups`	Return a summary of each group
`getGroups`	Return the specified groups. This operation returns groups in the order in which you specify the keys for each group. This operation creates a summary of each existing group which includes the group key, the display name (if it exists), and the cardinality of the group. This operation returns attributes for all user-specified groups and attributes that do not belong to any user-specified groups. To request all attributes that do not belong to any user-specified groups, specify the key `system-navigation_InternalGroup`.
`putGroups`	Add or replace each of the specified groups  **Note:** This operation is not supported by the read-only version of the service.
`deleteGroups`	Delete each of the specified groups  **Note:** This operation is not supported by the read-only version of the service.

Operations for Global Configuration Record

The operations for Global Configuration Record are the following:

Operation	Description
`getGlobalConfigRecord`	Obtain the Global Configuration Record from the MDEX Engine
`putGlobalConfigRecord`	Add the Global Configuration Record in the MDEX Engine. If the GCR already exists, it is replaced.  **Note:** This operation is not supported by the read-only version of the service.

Operations for XML configuration documents

The operations for managing the XML configuration documents are the following:

Operation	Description
`listConfigDocuments`	Return the name of each XML configuration document
`getConfigDocuments`	Return the requested XML configuration documents
`putConfigDocuments`	Add or replace each of the specified XML configuration documents  **Note:** This operation is not supported by the read-only version of the service.

Global operations

The Configuration Web Service has the following global operations:

Operation	Description
`export`	Export all attributes, groups, configuration documents, and the Global Configuration Record
`import`	Import all attributes, groups, configuration documents, and the Global Configuration Record  **Note:** This operation is not supported by the read-only version of the service.

Loading an attribute schema

You can use the Configuration Web Service to load the schema for your standard and managed attributes.

If you load your attribute schema before loading your source records, you can modify the resulting PDRs and DDRs as needed. After they have been configured as desired, you can then use the Data Ingest Web Service to load your source records.

- The `putProperties` element loads the standard attributes schema.
- The `putDimensions` element loads the managed attributes schema.

To illustrate the use of this operation, this `configTransaction` simple example will be used:

```
<config:configTransaction xmlns:config="http://www.endeca.com/MDEX/config/ser¬
vices/config/2010"
    xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
  <config:putDimensions>
    <mdex:record>
     <mdex-dimension_EnableRefinements type="mdex:int">1</mdex-dimension_En¬
ableRefinements>
     <mdex-dimension_IsDimensionSearchHierarchical type="mdex:int">1</mdex-
dimension_IsDimensionSearchHierarchical>
      <mdex-dimension_IsRecordSearchHierarchical type="mdex:int">1</mdex-
dimension_IsRecordSearchHierarchical>
      <mdex-dimension_Key type="mdex:string">WineType</mdex-dimension_Key>

    </mdex:record>
  </config:putDimensions>
  <config:putProperties>
    <mdex:record>
     <mdex-property_IsSingleAssign type="mdex:int">1</mdex-property_IsSin¬
gleAssign>
      <mdex-property_IsTextSearchable type="mdex:int">0</mdex-property_Is¬
TextSearchable>
      <mdex-property_IsUnique type="mdex:int">1</mdex-property_IsUnique>
      <mdex-property_Key type="mdex:string">WineID</mdex-property_Key>
      <mdex-property_TextSearchAllowsWildcards type="mdex:int">0</mdex-
property_TextSearchAllowsWildcards>
      <mdex-property_Type type="mdex:string">mdex:int</mdex-property_Type>

    </mdex:record>
    <mdex:record>
     <mdex-property_IsSingleAssign type="mdex:int">0</mdex-property_IsSin¬
gleAssign>
      <mdex-property_IsTextSearchable type="mdex:int">1</mdex-property_Is¬
TextSearchable>
      <mdex-property_IsUnique type="mdex:int">0</mdex-property_IsUnique>
     <mdex-property_Key type="mdex:string">Description</mdex-property_Key>

      <mdex-property_TextSearchAllowsWildcards type="mdex:int">1</mdex-
property_TextSearchAllowsWildcards>
     <mdex-property_Type type="mdex:string">mdex:string</mdex-property_Type>

    </mdex:record>
    <mdex:record>
     <mdex-property_IsSingleAssign type="mdex:int">0</mdex-property_IsSin¬
gleAssign>
      <mdex-property_IsTextSearchable type="mdex:int">1</mdex-property_Is¬
TextSearchable>
      <mdex-property_IsUnique type="mdex:int">0</mdex-property_IsUnique>
      <mdex-property_Key type="mdex:string">WineType</mdex-property_Key>
      <mdex-property_TextSearchAllowsWildcards type="mdex:int">1</mdex-
property_TextSearchAllowsWildcards>
     <mdex-property_Type type="mdex:string">mdex:string</mdex-property_Type>

    </mdex:record>
  </config:putProperties>
</config:configTransaction>
```

The example creates three standard attributes (WineID, WineType, and Description) and one managed attribute (WineType). The WineID attribute is configured as a single-assign, unique attribute, so that it can be used as an attribute key for source records. The WineType attribute is the standard attribute record used for the creation of the WineType managed attribute.

To load an attribute schema into the MDEX Engine:

1. Make sure that the MDEX Engine and its Configuration Service are running.
2. Make a SOAP request to the Configuration Service with the schema.

If the request is successful, the response will look like this example:

```
soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

   <soapenv:Header/>
   <soapenv:Body>
      <config-service:results xmlns:config-service="http://www.ende¬
ca.com/MDEX/config/services/config/2010">
         <config-service:successPutDimensions/>
         <config-service:successPutProperties/>
      </config-service:results>
   </soapenv:Body>
</soapenv:Envelope>
```

Chapter 4

Using the Conversation Web Service

This section describes the role of the Conversation Web Service in the MDEX Engine.

About the Conversation Web Service

The Conversation Web Service provides the primary means of querying data in the MDEX Engine.

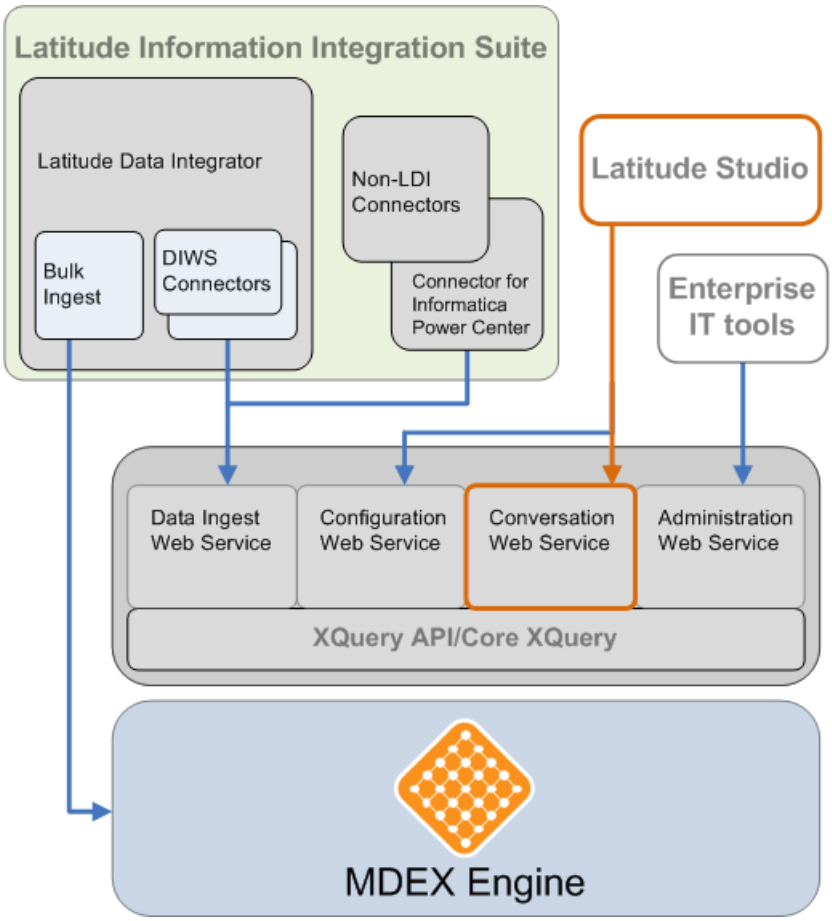
Overview

The Conversation Web Service is used by Latitude Studio components to send queries (such as navigation or search queries) to the MDEX Engine. The service is a WS-I compliant SOAP/HTTP Web service.

Important: The Conversation Web Service is an internal interface that is subject to change. Since the Conversation Web Service is used by Latitude Studio components, Endeca recommends using Latitude Studio as the primary interface to the MDEX Engine and not using the Conversation Web Service directly.

The Conversation Web Service is declared in `conversation.wsdl`. The service uses several library helper modules (such as `queryBuilder.xq`) that are located in the `lib` directory.

The following diagram shows how the Conversation Web Service fits into the larger picture of Endeca Web Services that serve the MDEX Engine.



The Conversation Web Service has an easy-to-program interface that insulates application-tier developers from the low-level XQuery APIs.

The service supports fundamental MDEX Engine behavior, such as:

- Guided Navigation
- Record and value searches
- Communication between the front-end application client and the MDEX Engine
- Support for a range of summarizations
- Management of the conversation state

Conversation Web Service interface

The Conversation Web Service interface provides an operation that queries the MDEX Engine. The information in this topic is provided for reference purposes only, since you use Latitude Studio components, (which utilize the Conversation Service) as a means of querying the MDEX Engine.

This topic provides a brief overview of the Conversation Web Service interface. For information on schema elements, see the *MDEX Engine API Reference*.

Function description

The Conversation Web Service supports the `query` operation:

```
<operation name= "query">
  <input name="request" message="cs:ConversationRequest"/>
  <output name="response" message="cs:ConversationResponse"/>
  <fault name="fault" message="cs:ConversationFault"/>
</operation>
```

At a high level, the Conversation Web Service facilitates a dialog with users about data. The Conversation Web Service manages filter state, content elements, and operators:

- The filter state reflects the currently selected records and the selections that were used to reach them.
- Content elements provide information about the currently selected records and the selections.
- The operators represent requests to change the filter state or reconfigure content elements, typically as a result of user actions. Operators can be specified for refinements, breadcrumbs, and other aspects of the front-end application available for navigation.

The sequence of actions in the Conversation Web Service "dialog" is as follows:

1. A user issues a query using the front-end application.

   This query is used to construct an initial filter state (typically, empty, or containing a simple record filter), and a number of content elements. This filter state and content element configurations are sent in a Conversation Web Service request.

2. The request returns the filter state and the resulting content elements.

   The content elements contain a number of operators, each representing a user action that might occur from this state. For instance, a record list content element contains operators to re-sort the records, or to view different pages; a navigation menu content element contains an operator for each refinement.

3. When the user chooses a particular action, the application can submit a new request through the Conversation Web Service, returning the old filter state and passing in the operators corresponding to the user actions.
4. The response returns a transformed query along with new filter state and new content element contents (response data).

To summarize, this is the "conversation" underlying this Web service: The Conversation Web Service offers a list of content elements and a number of operators, the front-end application selects some operators, and the Conversation Web Service offers new content elements and new operators.

Request

The `query` operation takes a request as its input. The schema for the `Request` is:

```
<complexType name="Request">
  <sequence>
    <element name="FilterState" type="cs:FilterState"/>
   <element name="Operator" type="cs:Operator" minOccurs="0" maxOccurs="un¬
bounded"/>
    <element name="ContentElementConfig" type="cs:ContentElementConfig"
minOccurs="0"
             maxOccurs="unbounded"/>
    <element name="PassThrough" type="cs:CatchAll" minOccurs="0"/>
  </sequence>
</complexType>
```

A request consists of a filter state and a list of "content elements" to compute. Each request specifies:

- A filter state, which contains inputs that affect the set of records to operate on. Currently it contains selected refinements, search terms, and record filters.
- A configuration for user interaction elements (called content elements). The configuration is an object that encapsulates some description or summarization of a filter state or the data therein. For example, a set of breadcrumbs, a navigation menu, or the data for a grid or chart. The content element is conceptually a "memo" to the MDEX Engine asking it to provide certain information relative to a certain filter state.
- A sequence of operators to apply that will transform the filter state and configuration.
- A pass-through, which is used to allow unrestricted schema in several places where user-defined XML is valuable.

Response

The `query` operation outputs a `Results` response. The response contains the `Request` element that generated it, as well as any components that were requested. Each component is returned only if its corresponding configuration was supplied in the request.

In other words, a response from the Conversation Web Service contains operators for refinements, breadcrumbs, and other aspects of the front-end application available for navigation.

The schema for the `Results` response is:

```
<complexType name="Results">
  <sequence>
    <element name="Request" type="cs:Request"/>
    <element name="ContentElement" type="cs:ContentElement" minOccurs="0"
maxOccurs="unbounded"/>
    <element name="PassThrough" type="cs:CatchAll" minOccurs="0"/>
  </sequence>
</complexType>
```

Error example

On failure, the SOAP fault is thrown.

Its `faultstring` element contains information about the request that caused the error, and the `detail` element includes pointers to the location of errors in the request, as in the following example of the SOAP fault:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

   <soapenv:Header/>
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode>soapenv:Client</faultcode>
         <faultstring>Contents of body must be a single request.</fault¬
string>
         <detail>
            <cs:Fault xmlns:cs="http://www.endeca.com/MDEX/conversa¬
tion/2010">
               <stack-trace:span uri="conversation.xq" xmlns:stack-
trace="http://www.endeca.com/XQuery/stacktrace/2009">
                  <stack-trace:start line="31" column="3"/>
                  <stack-trace:end line="31" column="76"/>
               </stack-trace:span>
            </cs:Fault>
         </detail>
      </soapenv:Fault>
```

```
   </soapenv:Body>
</soapenv:Envelope>
```

Part 3

Record Features

Chapter 5

Working with Endeca Records

This section provides information on handling Endeca records in your Web application.

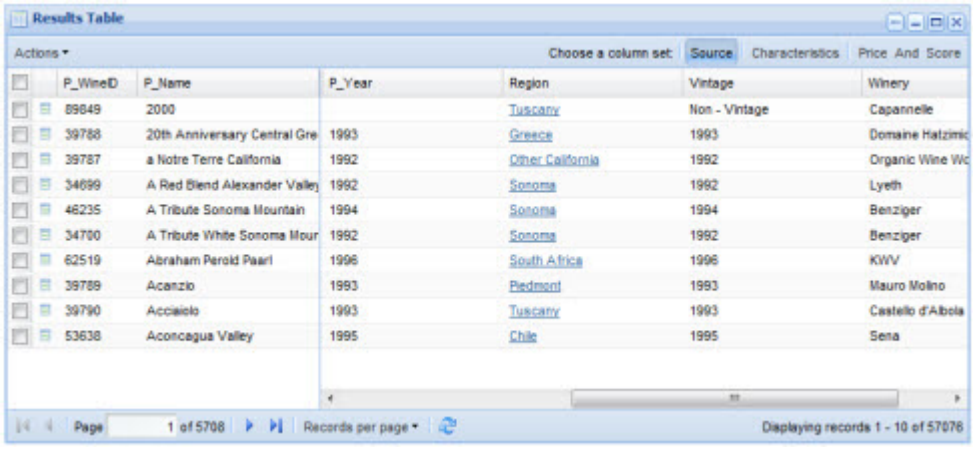
Displaying Endeca records with Latitude Studio

The **Results Table** component provides displaying records in a tabular format. The **Results List** component displays records in a format similar to regular Web search results. The **Data Explorer** component displays each record as a set of key value pairs.

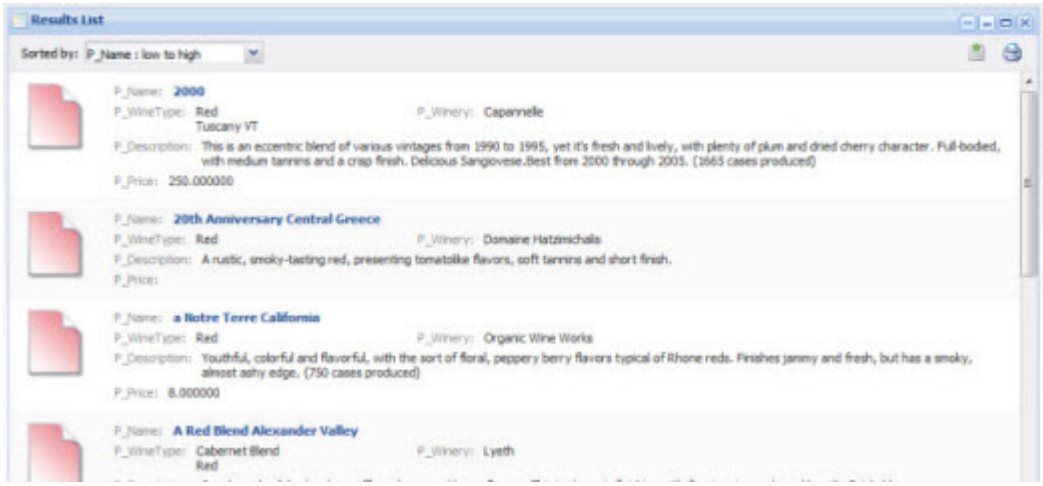
The **Results Table** component can show results from Analytics and non-Analytics queries. The records are displayed with selected attributes and attribute values.

The component provides a list of records in table form, and allows users to view a record's details. Users can page through, sort, and scroll across large tables, switch between column sets, and drill down on selected attributes. Locked columns help users keep track of the data they are viewing.

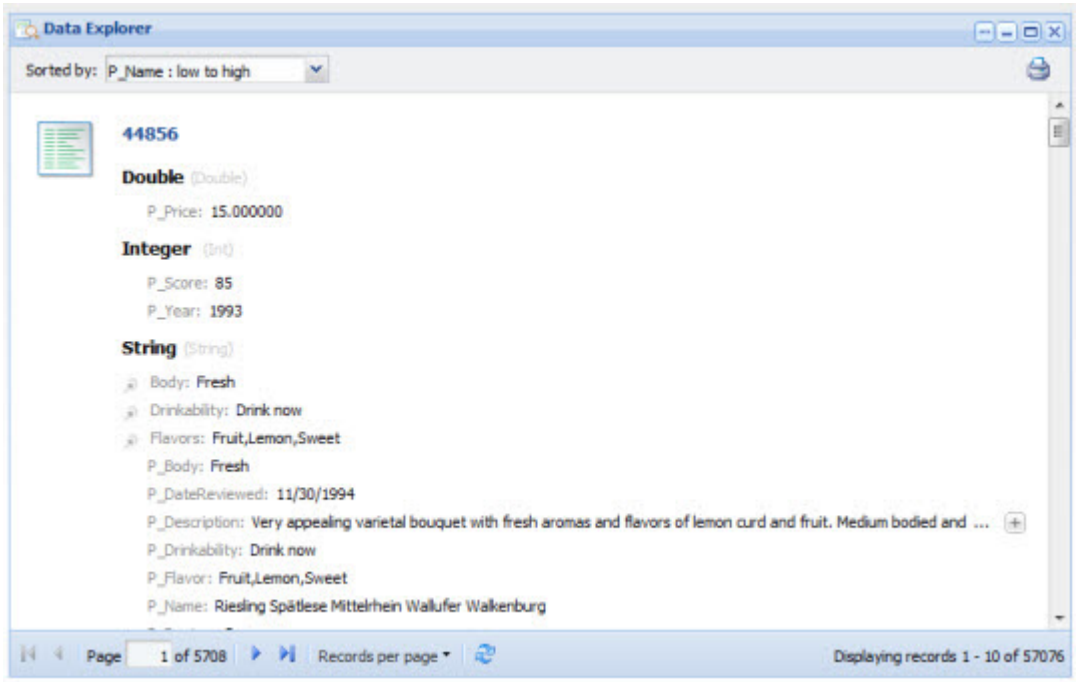
The following is an example of a **Results Table** component displaying a record list:



The **Results List** component displays a list of records in a format similar to regular search results. It displays a selected set of attributes for each record. It does not support Analytics. From the **Results List** component, users can navigate through the list, display record details, and drill down on attributes.



The **Data Explorer** component displays each record as a complete set of key value pairs. It also allows users to display the details for a record. It does not support Analytics, and is most useful for verifying newly added or updated data.



For details on adding and configuring a **Results Table**, **Results List**, or **Data Explorer** component in your Latitude Studio application, see the *Latitude Studio Power User's Guide*.

Implementing export of records in Latitude Studio

Once the records are returned by the Conversation Web Service, you can export them to a file from Latitude Studio components.

For information on configuration options, including setting the number of records to export, see the *Power User's Guide*.

Displaying Endeca records with the API

This section describes how to display a record list using the Conversation Service API.

For more information on the Conversation Service interface, see the *MDEX Engine API Reference*.

Related Links

Record list configuration on page 55
The `RecordListConfig` type defines the configuration for the returned record list.

RecordList result on page 56
The records returned from the query are contained in the `RecordList` element.

Paging through a record set on page 57
A paging UI control is helpful if many records are returned.

Retrieving large numbers of records on page 59
To obtain a large number of records that can later be exported, you request them as part of the `RecordListConfig` element in the Conversation Web Service.

Performance impact of requesting large numbers of records on page 60
Requesting a large number of records at once can reduce memory usage in your front-end application if the response is handled by a streaming parser, as it is in Latitude Studio.

Record list configuration

The `RecordListConfig` type defines the configuration for the returned record list.

Record search queries include a `RecordListConfig` component that lets you configure aspects of the list of records that is returned from the MDEX Engine. For example, you can figure the sort order and which attributes should be returned.

The format of the `RecordListConfig` type is shown in this example:

```
<ContentElementConfig xsi:type="RecordListConfig"
      HandlerFunction="RecordListHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
     Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      MaxPages="40">
   <Column>WineType</Column>
   <Column>Price</Column>
   <RecordsPerPage>5</RecordsPerPage>
   <Page>6</Page>
   <Sort Key="Num" Direction="Ascending" />
</ContentElementConfig>
```

The meanings of the `RecordListConfig` elements and attributes are as follows:

Element/Attribute	Meaning
`HandlerFunction`	Specifies the `RecordListHandler` handler function for this `ContentElementConfig`. Required.
`HandlerNamespace`	Specifies the namespace for the handler function. Required.
`Id`	An arbitrary identifier for this `ContentElementConfig`. Required.

Element/Attribute	Meaning
`MaxPages`	Optionally specifies an integer that is the maximum number of record pages to be returned. If this attribute is omitted, a default value of 20 is used for the query.
`Column`	Optionally specifies an attribute that should be returned with the record. You can specify multiple instances of the `Column` element. Note that you do not have to specify the primary key, because it is automatically returned. If no `Column` elements are specified, then all the record's attributes are returned.
`RecordsPerPage`	Optionally specifies an integer that is the maximum number of records (`Record` elements) to be displayed in the `ContentElement` of the result. If this element is omitted, a default value of 10 is used.
`Page`	Optionally specifies an integer that is the page to be displayed (that is, it provides an offset into the overall list of pages). The offset is a zero-based index, which means that 0 (zero) specifies the first page. This element allows users to page through a long result set, either directly or step by step. If an offset is greater than the total number of pages, then the record list returned will not include records. If this element is omitted, a default value of 0 is used.
`Sort Key Direction`	Optionally specifies a sort order for the record list. `Key` specifies the attribute used for the sort. `Direction` specifies an `Ascending` (the default) or `Descending` sort order.

RecordList result

The records returned from the query are contained in the `RecordList` element.

A list of records is returned with every MDEX Engine query result. The list of records is a `RecordList` type that is returned in a `Results` response by the Conversation Web Service. Each record is returned in a `Record` element.

The following sample snippet shows a `RecordList` with one record, one pagination control, and one column:

```
<cs:ContentElement xsi:type="cs:RecordList"
    Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <cs:NumRecords>19</cs:NumRecords>
   <cs:TotalPages>4</cs:TotalPages>
   <cs:RecordRange First="1" Last="5"/>
   <cs:RecordListEntry>
      <cs:Record>
         <Description type="mdex:string">Dense and vegetal, with peach and
 spice flavors.</Description>
         <Flavors type="mdex:string">nine</English>
         <WineID type="mdex:int">101</WineID>
         <WineType type="mdex:string">white</WineType>
      </cs:Record>
      <cs:ComputedProperties/>
   </cs:RecordListEntry>
   ...
   <cs:PaginationControl Label="First" Active="false">
     <cs:Operator OwnerId="RecordList" Page="0" xsi:type="cs:PageOperator"/>
```

```
   </cs:PaginationControl>
   ...
   <cs:Column ColumnKey="WineID" DisplayName="Wine ID" SpecColumn="true">
      <cs:SortControl Key="WineID" Direction="Ascending" Active="true"
xsi:type="cs:SortControl">
         <cs:Operator OwnerId="RecordList" xsi:type="cs:SortOperator"
Key="WineID" Direction="Ascending"/>
      </cs:SortControl>
      <cs:SortControl Key="WineID" Direction="Descending" Active="false"
xsi:type="cs:SortControl">
         <cs:Operator OwnerId="RecordList" xsi:type="cs:SortOperator"
Key="WineID" Direction="Descending"/>
      </cs:SortControl>
   </cs:Column>
   ...
</cs:ContentElement>
```

The elements in the `RecordList` contain the following information:

- `NumRecords` specifies the total number of records (`Record` elements) that were returned from the query.
- `TotalPages` lists the total number of pages of records.
- `RecordRange` lists the starting and ending records for this page set.
- Each `RecordListEntry` contains a specific record in a `Record` element and a `ComputedProperties` element that has any computed attributes (such as geocode distance or snippets) for that record.
- `PaginationControl` is a control (a `PageOperator`) for a specific record page.

In addition, the attributes on the `Column` element contain the following information for a specific attribute on a record:

- `Key` identifies the name (in an NCName format) of the attribute.
- `DisplayName` specifies the name of the attribute in an easy-to-understand format.
- `SpecColumn` identifies whether the standard attribute is the primary key for the records. If set to **true** identifies this attribute as the primary key attribute for the records.
- `SortControl` identifies the sort order (Ascending or Descending) of the attributes.

The `SpecColumn` allows you to select a record for viewing its record details.

Paging through a record set

A paging UI control is helpful if many records are returned.

An MDEX Engine query may return more records than can be displayed all at once. A common user interface mechanism for overcoming this is to create pages of results, where each page displays a subset of the entire result set.

The `RecordList` in the `Results` response includes pagination controls (the `PaginationControl` type) that you can use for paging.

The following is an example of a `RecordList` with a total of seven record pages and three records per page:

```
<cs:ContentElement xsi:type="cs:RecordList" Id="RecordList"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <cs:NumRecords>19</cs:NumRecords>
   <cs:TotalPages>7</cs:TotalPages>
```

```
   <cs:RecordRange First="1" Last="3"/>
   <cs:RecordListEntry>
   ...
   </cs:RecordListEntry>
   <cs:PaginationControl Label="First" Active="false">
     <cs:Operator OwnerId="RecordList" Page="0" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   <cs:PaginationControl Label="Previous" Active="false">
      <cs:Operator OwnerId="RecordList" Page="-1" xsi:type="cs:PageOpera¬
tor"/>
   </cs:PaginationControl>
   <cs:PaginationControl Label="1" Active="false">
     <cs:Operator OwnerId="RecordList" Page="0" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   <cs:PaginationControl Label="2" Active="true">
     <cs:Operator OwnerId="RecordList" Page="1" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   <cs:PaginationControl Label="3" Active="true">
     <cs:Operator OwnerId="RecordList" Page="2" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   <cs:PaginationControl Label="Next" Active="true">
     <cs:Operator OwnerId="RecordList" Page="1" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   <cs:PaginationControl Label="Last" Active="true">
     <cs:Operator OwnerId="RecordList" Page="6" xsi:type="cs:PageOperator"/>

   </cs:PaginationControl>
   ...
</cs:ContentElement>
```

The `RecordList` is the initial access point for providing the paging controls for the entire record set. By default, the query returns a maximum of ten records for display. To override this setting, use the `RecordsPerPage` element in the `RecordListConfig` type, as in this example that sets five records for display:

```
<ContentElementConfig xsi:type="RecordListConfig"
     HandlerFunction="RecordListHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

    Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <RecordsPerPage>5</RecordsPerPage>
</ContentElementConfig>
```

The `NumRecords` element in the `RecordList` lists the total number of records being returned by the query:

```
<cs:NumRecords>20</cs:NumRecords>
```

The default page offset for a record set is zero, meaning that the first ten records are displayed. The default offset can be overridden with the `PageOperator` type, as in this example that sets the offset to the third page of records:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="PageOperator" OwnerId="RecordList" Page="3"/>
```

If the number of total pages is 1:

```
<cs:TotalPages>1</cs:TotalPages>
```

then no paging controls are needed.

If the number of total pages is 2 or greater, you can use the `PaginationControl` elements in the `RecordList` to go to the appropriate page, as indicated in the following table.

Page Label	Result
First	Goes to the first record page (which is page 0).
Previous	Goes to the previous record page.
Next	Goes to the next record page.
Last	Goes to the last record page.
1	Goes to the first record page (which is page 0).
2 or greater	Goes to the Nth record page.

Note that the `Active` attribute in a `PaginationControl` element indicates whether that paging control is relevant within the context of the current state. For example, if you are on the last record page, then neither the **Next** or **Last** paging controls will be active.

Retrieving large numbers of records

To obtain a large number of records that can later be exported, you request them as part of the `RecordListConfig` element in the Conversation Web Service.

A query that requests a large number of records that could later be exported is the same as any valid navigation query requesting a list of records. This topic contains examples of Conversation Web Service request and response formats for such a query. No configuration is necessary to request a large number of records. Any record that is returned as part of the `RecordListConfig` request, is available to be exported.

When creating the navigation query for a list of records that will be exported, you do not need to specify the number of Endeca records that should be returned. The Conversation Web Service returns records in the record list as it would for any other request for records. The settings that limit the number of records for export are configured in Latitude Studio. For information on configuring these settings, see the *Latitude Studio Power User's Guide*.

Example request

To request a record list with a Conversation Web Service request, use `ContentElementConfig` of type `RecordListConfig`.

There is no requirement to specify any new parameters in the `RecordListConfig`. Simply set the `RecordsPerPage` to the number of records desired for export, and `Page` to 0."

In this abbreviated example, you can see the format for `RecordListConfig`:

```
<ContentElementConfig xsi:type="RecordListConfig"
      HandlerFunction="RecordListHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
     Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      MaxPages="40">
```

```
   <Column>WineType</Column>
   <Column>Price</Column>
   <RecordsPerPage>20</RecordsPerPage>
   <Page>0</Page>
   <Sort Key="Num" Direction="Ascending" />
</ContentElementConfig>
```

Example response

The following abbreviated example shows a returned list:

```
<cs:ContentElement xsi:type="cs:RecordList"
    Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <cs:NumRecords>19</cs:NumRecords>
   <cs:TotalPages>40</cs:TotalPages>
   <cs:RecordRange First="1" Last="19"/>
   <cs:RecordListEntry>
      <cs:Record>
         <Description type="mdex:string">Dense and vegetal, with peach and
 spice flavors.</Description>
         <Flavors type="mdex:string">nine</English>
         <WineID type="mdex:int">101</WineID>
         <WineType type="mdex:string">white</WineType>
      </cs:Record>
      <cs:ComputedProperties/>
   </cs:RecordListEntry>

   ...
</cs:ContentElement>
```

Performance impact of requesting large numbers of records

Requesting a large number of records at once can reduce memory usage in your front-end application if the response is handled by a streaming parser, as it is in Latitude Studio.

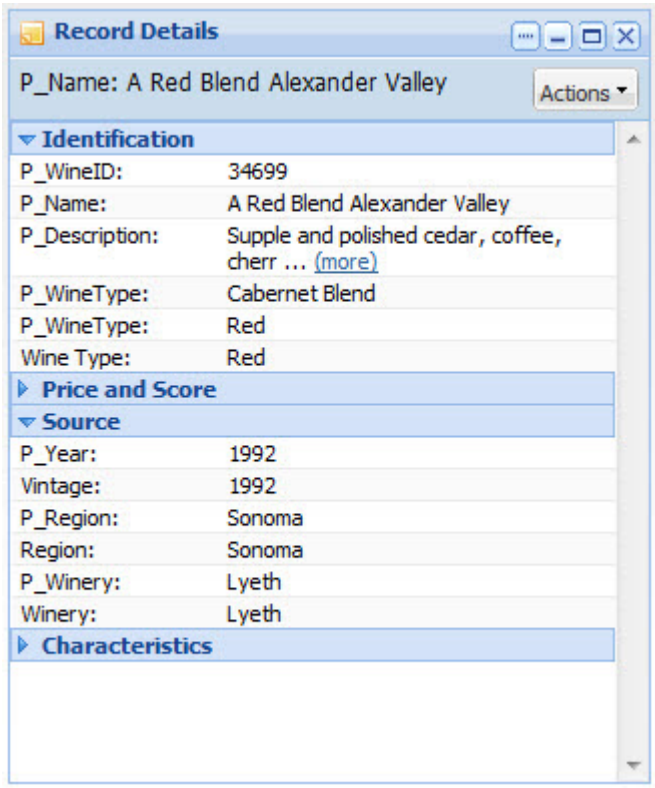
Without this approach, application developers who want to export large amounts of data are required to split up the task and deal with a few records at a time to avoid running out of memory in the application server's threads. This division of exports adds query processing overhead to the MDEX Engine which reduces system throughput and slows down the export process.

Displaying attribute values for records in Latitude Studio

Latitude Studio includes components that can display attribute values for records.

The **Results Table**, **Results List**, and **Data Explorer** components each contain a list of records with their attributes and attribute values.

The **Record Details** component displays the attribute values for a specific selected record.



Displaying attribute values with the API

Attribute values on records can be retrieved via the Conversation Service API.

Records are returned in `Record` elements. Each managed attribute value on a record is returned in a format like this example:

```
<WineType cs:ValueName="Red" type="mdex:string">/3</WineType>
```

where:

- `WineType` is the name of the managed attribute.
- `Red` is the name of the managed attribute value.
- `/3` is the managed attribute value specifier.

This example shows a record with five managed attribute values:

```
<cs:Record>
   <Decimal cs:ValueName="53" type="mdex:string">/1-100/51-60/53</Decimal>

   <English type="mdex:string">five three</English>
   <FirstDigit cs:ValueName="5" type="mdex:string">/5</FirstDigit>
   <Hex cs:ValueName="0035" type="mdex:string">/0001-0100/0031-
0040/0035</Hex>
   <LastDigit cs:ValueName="3" type="mdex:string">/3</LastDigit>
   <Num type="mdex:int">53</Num>
   <NumberOfDigits cs:ValueName="2" type="mdex:string">/2</NumberOfDigits>

   <Spanish type="mdex:string">cinco tres</Spanish>
</cs:Record>
```

Your application front end can iterate through the record, extract the attribute values for the record, and display a table containing the results.

Performance impact when displaying attribute values

Displaying too many attribute values can cause a performance hit.

The main purpose of attribute values is to enable navigation through the records. Passing attribute values through the system consumes resources. Therefore, the default behavior of the MDEX Engine is to return attribute values on records only when a record query request has been made (not for navigation query requests). As mentioned above, this behavior can be changed. However, the developer should exercise caution when passing attribute values through to the record list, because doing this with too many attributes can cause a performance hit.

Chapter 6

Sorting Endeca Records

The sorting functionality allows the user to define the order of Endeca records returned with each navigation query.

About record sorting

When making a basic navigation request, the user may define a series of attributes and order (Ascending or Descending) pairs.

If the user does not specify sort order as part of the query, the MDEX Engine returns query results in a Descending order on the primary key for returned records (i.e., the standard attribute listed in the `SpecColumn` element in the `RecordList`). You cannot change the default record sort order for the system; you can only specify a different sort order on a per-query basis.

All of the records corresponding to a particular navigation state are considered for sorting, not just the records visible in the current request. For example, if a navigation state applies to 100 bottles of wine, all 100 bottles are considered when sorting, even though only the first ten bottles may be returned with the current request.

Record sorting only affects the order of records. It does not affect the ordering of attributes or attribute values that are returned for query refinement.

Note that all attributes are automatically enabled for record sorting when they are created. Therefore, no attribute configuration is required for sorting.

Global sort order of records

This topic discusses the global sort order of records.

Once the records have been added to the MDEX Engine, the MDEX Engine maintains the index of records in memory. The following rules apply to how the records are sorted in the results returned by the Conversation Web Service in response to queries:

- Records are sorted according to the sort order that you specified, if any.
- Even if you specified a sort order, it may not have uniquely determined the resulting order of records — this usually happens when some records only differ in attributes that were not included in the sort specification. In such cases, the MDEX tie-breaks the sorting results at random.
- Subsequent requests with the same query will result in the same order (the tie-break is consistent) unless you have modified the records in any way between requests. For example, the order will

change if you delete any of the records and add them to the MDEX Engine again, even if they are identical.

Note that when a sorted record result list is requested, string values will be sorted case-insensitively, with ties broken with a case-sensitive comparison (upper-cased words will rank above lower-cased words). For example, for the six records **A**, **B**, **C**, **a**, **b**, and **c**, the resulting sort order will be:

```
A
a
B
b
C
c
```

Query-time sort ordering

On a per-query basis, you can specify a key on which to sort the records and a sort direction.

You can add a `Sort` type to a `RecordList` configuration that lets you specify a key to sort on and the sort direction. The `Sort` format is:

```
<Sort Key="keyName" Direction="dirOrder"/>
```

where:

- *keyName* is the name of an attribute on which to sort.
- *dirOrder* is either `Ascending` for an ascending order or `Descending` for a descending order.

Note that case is sensitive for both the attribute name and the sort order.

The following example shows an Ascending sort order based on the `WineID` attribute:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
         <FilterState/>
         <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchOperator" Within="false">
           <Search Mode="AllPartial" Key="All">red</Search>
         </Operator>
         <ContentElementConfig xsi:type="RecordListConfig" HandlerFunc¬
tion="RecordListHandler"
            HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
            Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
            <Sort Key="WineID" Direction="Ascending"/>
            <RecordsPerPage>10</RecordsPerPage>
         </ContentElementConfig>
      </Request>
   </soap:Body>
</soap:Envelope>
```

Troubleshooting application sort problems

This topic presents some approaches to solving sorting problems.

If the returned records do not seem to respect the sort key parameter, there are some potential problems:

- Was the attribute specified as a numeric when it is actually alphanumeric? Or vice versa? In this case, the MDEX Engine returns a valid response, but the sorting may be incorrect.
- If a record has multiple attribute values for a single attribute, the MDEX Engine sorts the records based on the first value associated with the key. If the application is displaying the last value, the records will not appear to be sorted correctly. In general, attributes that are used for sorting should only have one value assigned per record.
- If certain records in a record set lack a sort-key value, they will always appear last in a result set. Therefore, if you reverse a sort order on a record set containing such records, the order of the entire record set will not be reversed—the records without a sort-key value always sort at the end of the set.

Chapter 7

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 attribute 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 attribute values joined in a multi-way OR expression. Or, a filter might consist of a complex nested expression of ANDs, ORs, and NOTs on managed attribute IDs and attribute values.

Filter expressions are passed directly as part of an MDEX Engine query. 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 values are restricted to lead only to records contained within the subset.

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

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

Record filter syntax

Record filters are specified with query-based expressions.

Record filters are specified directly within an MDEX Engine query. The query-level syntax supports prefix-oriented Boolean functions (AND, OR, and NOT), colon-separated paths for standard attribute values, and forward-slash-separated paths for managed attribute values.

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-list> ::= <filter>
                | <filter>,<filter-list>
<literal>     ::= <pval>
                | <dval-path>
<pval>        ::= <prop-key>:<prop-value>
<prop-key>    ::= <string>
<prop-value>  ::= <string>
<dval-path>   ::= <string>
                | <string>/<dval-path>
<string>      ::= any character string
```

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

```
( ) , : \ /
```

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 attribute is `Sony` and record assigned managed attribute value is `1001`.
- Value for Manufacturer is `Aiwa` and record is not assigned managed attribute value `1002`.
- Value for Manufacturer attribute is `Denon`.

Using Boolean attributes

Filtering by Boolean attribute assignments is supported. You can specify the Boolean value as `true` (or its synonym of `1`), or as `false` (or its synonym of `1`). For example, assuming `isOdd` is a Boolean attribute, both `isOdd:1` and `isOdd:true` will parse properly and yield the same results.

Record filter result caching

The MDEX Engine caches the results of 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.

Requesting record filters with the API

The `RecordFilterOperator` complex type adds a record filter component to the filter state.

When making a query to the Conversation Web Service, you can use a `RecordFilterOperator` to limit the set of returned records, as in this example:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
      <FilterState />
      <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="RecordFilterOperator">
        <RecordFilter Name="WineTypes">OR(OR(WineType:white),OR(Wine¬
Type:sparkling))</RecordFilter>
      </Operator>
      <ContentElementConfig xsi:type="RecordListConfig" HandlerFunc¬
tion="RecordListHandler"
        HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="RecordList"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <RecordsPerPage>5</RecordsPerPage>
      </ContentElementConfig>
    </Request>
  </soap:Body>
</soap:Envelope>
```

The example sets a record filter for records that have a WineType standard attribute with a value of "white" or "sparkling".

Managed attribute value examples

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

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

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

```
OR(sku/123,OR(sku/456),OR(sku/789))
```

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

```
sku/123
```

Note that the / (forward slash) is used as the delimiter for value paths.

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 attribute 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 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 record 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
- 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.

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, 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 attribute 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 8

Using Range Filters

You can use range filters for navigation queries.

About range filters

Range filters allow a user, at request time, to specify an arbitrary, dynamic range of values that are then used to limit the records returned for a navigation query.

The remaining refinement values for the records in the result set are also returned. For example, a range filter would be used if a user were querying for wines within a price range, say between $10 and $20.

It is important to remember that, similar to record search, range filters are simply modifiers for a navigation query. The range filter acts in the same manner as an attribute value, even though it is not a specific system-defined attribute value.

Only records returned by the basic navigation request are considered when evaluating the range filter. You can use a range filter in a query on any of the record attributes.

Supported attribute types

Range filters can be applied to either standard attributes or managed attributes of the following supported types.

- Standard attributes of type Numeric (Integer, Long, Double, dateTime), type Geocode, or type Boolean
- Managed attributes of type Numeric that contain only Integer or Floating point values

For values of attributes of type Double, you can specify values using both decimal (0.00...68), and scientific notation (6.8e-10).

For standard attributes of type Boolean, `false` is interpreted to be less than `true`. If isOdd is a Boolean attribute, a query of "isOdd|LTEQ false" will return all records with assignments of `false` on attribute isOdd. Likewise, "isOdd|BTWN false true" will return all records with any assignment on isOdd.

No MDEX Engine configuration flags are necessary to enable range filters. All numeric standard attributes and managed attributes and all geocode standard attributes are automatically enabled for use in range filters.

Implementing range filters in Latitude Studio

To use range filters in a Latitude Studio application, you must add a **Range Filters** component and configure at least one attribute.

After adding the **Range Filters** component, you can configure the component to select the attributes to be used for range filter queries. For detailed information, see the *Latitude Studio Power User's Guide*.

Troubleshooting range filter problems

This topic presents some approaches to solving range filter problems.

Similar to record search, the user-specified interaction of this feature allows a user to request a range that does not match any records (as opposed to the system-controlled interaction of Guided Navigation in which the MDEX Engine controls the refinement values presented to the user). Therefore, it is possible for a user to make a dead-end request when using a range filter. Applications implementing range filters need to account for this.

If a range filter request specifies an attribute that does not exist in the MDEX Engine, the query throws a SOAP fault and an error message is written to the MDEX Engine error log:

```
ERROR DGRAPH {dgraph}
Range filter does not specify a legal attribute name- "PropName".
```

If a range filter request does not specify numeric range values, the query also throws an `InvalidDatatypeValueException` in the application. The MDEX Engine error log will output a message similar to this example:

```
Message: Value 'five' does not match any member types (of the union)
```

If the specified attribute exists but is not configured as numeric or geocode, the query will not throw an exception, but the following message will be written to the MDEX Engine error log:

```
ERROR DGRAPH {dgraph} Tried to range filter on unsupported type
```

It is likely that no records will be correctly evaluated against the query and therefore no results will be returned.

You should also be careful of dollar signs or other similar characters in attribute values that would prevent an attribute from being defined as numeric.

Performance impact for range filters

Range filters impact the MDEX Engine response times, but not memory usage.

Because range filters are not indexed, this feature does not impact the amount of memory needed by the MDEX Engine. However, because the feature is evaluated entirely at request time, the MDEX Engine response times are directly related to the number of records being evaluated for a given range filter request. You should test your application to ensure that the resulting performance is compatible with the requirements of the deployment.

Implementing range filters with the API

This section describes how to issue range filter queries using the Conversation Service API. For additional information on the Conversation Web Service interface, see the *MDEX API Reference*.

Operator for range filters

A range filter search requires a `RangeFilterOperator` with a `Search` type.

The syntax for a `RangeFilterOperator` is shown in this example for a geocode range filter:

```
<cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="RangeFilterOperator">
   <cs:RangeFilter AttributeName="Location">
      <cs:LowerBound Inclusive="false">10</LowerBound>
      <cs:UpperBound Inclusive="false">20</UpperBound>
      <cs:GeocodeReferencePoint Latitude="+42.365615"
          Longitude="-71.075647"/>
   </cs:RangeFilter>
</cs:Operator>
```

The meanings of the `RangeFilter` elements and attributes are as follows:

RangeFilter Attribute	Meaning
`AttributeName`	The name of a numeric standard attribute, geocode standard attribute, or numeric managed attribute on which range filtering will be performed. Only a single attribute (standard or managed) can be specified per range filter.
`LowerBound`	A numeric value that is the lower value of the range to search.
`UpperBound`	A numeric value that is the upper value of the range. This value must be equal to or greater than the lower value.
`Inclusive`	A Boolean that determines whether the values for the lower and/or upper bounds are excluded (`false`) or included (`true`) in the range. Note that geocode filters support only exclusive bounds.
`GeocodeReferencePoint`	Specifies latitude and longitude values for a geocode range filter.

The following topics provide details for the various types of range filters.

Note: Multiple range filters can be applied to the same attribute. The results will include the intersection of the result sets of each range filter.

Less-than range filter format

The `UpperBound` element lets you make less-than range filter queries.

To make a less-than query, use only the `UpperBound` element. Because you are specifying only the upper bound of the range, all returned records will fall below this bound (i.e., be less than the upper bound).

In addition, the `Inclusive` attribute determines whether the specified value is included in the range:

- If `Inclusive` is set to `false`, the value for the `UpperBound` element is exclusive. That is, the specified value for the `UpperBound` element is not included in the range.
- If `Inclusive` is set to `true`, the value for the `UpperBound` element is inclusive.

The default for the `Inclusive` attribute is `false` (that is, if you omit the attribute, the query will work as if you had specified `false` for this attribute).

Less-than example

The following is an example of an inclusive less-than query:

```
<cs:Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <cs:State/>
   <cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="RangeFilterOperator">
     <cs:RangeFilter AttributeName="Price">
        <cs:UpperBound Inclusive="true">10</cs:UpperBound>
     </cs:RangeFilter>
   </cs:Operator>
   <cs:ContentElementConfig xsi:type="RecordListConfig"
      HandlerFunction="RecordListHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      Id="RecordList"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
</cs:Request>
```

This example returns all items whose price is up to, and including, $10.

In the example, if `Inclusive` had been set to `false`, the query would return all items whose price is up to, but not including, $10.

Greater-than range filter format

The `LowerBound` element lets you make greater-than range filter queries.

To make a greater-than query, use only the `LowerBound` element. Because you are specifying only the lower bound of the range, all returned records will be above this bound (i.e., be greater than the lower bound).

In addition, the `Inclusive` attribute determines whether the specified value is included in the range:

- If `Inclusive` is set to `false`, the value for the `LowerBound` element is exclusive. That is, the specified value for the `UpperBound` element is not included in the range.
- If `Inclusive` is set to `true`, the value for the `LowerBound` element is inclusive.

The default for the `Inclusive` attribute is `false` (that is, if you omit the attribute, the query will work as if you had specified `false` for this attribute).

Greater-than example

The following is an example of an inclusive greater-than query against a wine data set:

```
<cs:Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <cs:State/>
   <cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="RangeFilterOperator">
     <cs:RangeFilter AttributeName="Score">
        <cs:LowerBound Inclusive="true">90</cs:LowerBound>
     </cs:RangeFilter>
```

```
   </cs:Operator>
   <cs:ContentElementConfig xsi:type="RecordListConfig"
      HandlerFunction="RecordListHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      Id="RecordList"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
</cs:Request>
```

This example returns all wines whose rating score is 90 and above, including wines with a score of 90.

In the example, if `Inclusive` had been set to `false`, the query would all wines whose rating score is greater than 90, but wound not include wines with a score of 90.

Between range filter format

Use both `UpperBound` and `LowerBound` elements to construct between range filter queries.

A between range filter query returns records with a numeric attribute value that falls between a lower bound (the `LowerBound` element) and an upper bound (the `UpperBound` element).

Between range filters must be inclusive. Therefore, the `Inclusive` attribute for both bound elements must be set to true, as shown in the example below.

Between example

The following is an example of a between range filter query:

```
<cs:Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <cs:State/>
   <cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="RangeFilterOperator">
     <cs:RangeFilter AttributeName="Price">
        <cs:LowerBound Inclusive="true">10</cs:LowerBound>
        <cs:UpperBound Inclusive="true">20</cs:UpperBound>
     </cs:RangeFilter>
   </cs:Operator>
   <cs:ContentElementConfig xsi:type="RecordListConfig"
      HandlerFunction="RecordListHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      Id="RecordList"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
</cs:Request>
```

This example returns all wines whose price is between $10 and $20. Because both bound elements are inclusive, the returned records include wines that cost $10 and $20.

Geocode range filter format

When used with a geocode standard attribute, the `GeocodeReferencePoint` element indicates a range filter based on the distance of that geocode standard attribute from a given reference point.

The syntax for a geocode range filter is shown in this snippet sample from the request:

```
<cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:type="RangeFilterOperator">
   <cs:RangeFilter AttributeName="Location">
```

```
       <cs:LowerBound>0</cs:LowerBound>
       <cs:UpperBound>20</cs:UpperBound>
       <cs:GeocodeReferencePoint Latitude="+42.365615"
           Longitude="-71.075647"/>
   </cs:RangeFilter>
   </cs:Operator>
   ...
</cs:Request>
```

The meanings of the `RangeFilter` elements and attributes are described in the following table:

RangeFilter Geocode Attribute	Meaning
`AttributeName`	The name of a geocode standard attribute. Only one geocode standard attribute can be specified per range filter.
`LowerBound`	Used to specify a greater-than distance (in kilometers) from the geocode standard attribute to the reference point.
`UpperBound`	Used to specify a less-than distance (in kilometers) from the geocode standard attribute to the reference point. Note that the range filter must contain at least one of the boundary elements.
`GeocodeReference¬ Point`	Uses the `Latitude` and `Longitude` attributes to specify a reference point for range filtering of geocode attributes.
`Latitude`	The latitude of the location in whole and fractional degrees (positive values indicate north latitude and negative values indicate south latitude).
`Longitude`	The longitude of the location in whole and fractional degrees (positive values indicate east longitude and negative values indicate west longitude).

Note the following about geocode range filters:

- Distance limits (specified via the `LowerBound` and `UpperBound` elements) are exclusive. This means that you cannot use the `Inclusive` attribute set to `true` for these two elements. Because the `Inclusive` default is `false`, you can omit this attribute for geocode range filters.
- Geocode range filters must contain at least one of the bound elements.
- Distance limits in range filters are always expressed in kilometers.

The records are filtered by the distance from the filter key to the latitude/longitude pair.

Between geocode range filters

Use both `UpperBound` and `LowerBound` elements to indicate that the distance from the geocode standard attribute to the reference point is between the two bounds. The example at the beginning of this topic returns only records whose location (in the Location standard attribute) is between 0 and 20 miles from the reference point.

Less-than geocode range filters

If only the `UpperBound` element is used, the distance from the geocode standard attribute to the reference point will be less than the given amount. For example:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:type="RangeFilterOperator">
   <RangeFilter AttributeName="Location">
       <UpperBound>20</UpperBound>
```

```
      <GeocodeReferencePoint Latitude="+42.365615"
         Longitude="-71.075647"/>
   </RangeFilter>
   </Operator>
```

This sample query returns all records whose location is less than 20 miles from the reference point.

Greater-than geocode range filters

If only the `LowerBound` element is used, the distance from the geocode standard attribute to the reference point will be greater than the given amount. For example:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:type="RangeFilterOperator">
   <RangeFilter AttributeName="Location">
      <LowerBound>20</LowerBound>
      <GeocodeReferencePoint Latitude="+42.365615"
         Longitude="-71.075647"/>
   </RangeFilter>
</Operator>
```

This sample query returns all records whose location is greater than 20 miles from the reference point.

Removing range filter operators

The `PopRangeFilterOperator` removes a range filter component from the state.

The syntax of this operator is:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="PopRangeFilterOperator">
  <RangeFilter AttributeName="Price" />
</Operator>
```

The `AttributeName` attribute is the name of the numeric or geocode attribute for which a range filter component had been used.

If the call is successful, the following `Results` component is returned:

```
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
    xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
   <cs:Request>
      <cs:State/>
   </cs:Request>
</cs:Results>
```

The empty State element shows that the range filter component was successfully removed.

Rendering the range filter results

The results of a range filter request can be rendered in the UI like any navigation request.

Because a range filter request is simply a variation of a basic navigation request, rendering the results of a range filter request is identical to rendering the results of a navigation request.

Note, however, that there are no API components to access a list of valid range filter standard or managed attributes. This is because the attributes do not need to be explicitly identified as valid for range filters in the same way that they need to be explicitly identified as valid for record search. Therefore, specific standard or managed attributes that a user is allowed to filter against must be correctly identified as numeric or geocode in the instance configuration.

Examples of range filter parameters

This topic shows some valid examples of range filter queries.

Consider the following examples that use these four records:

Record	WineType managed attribute	Price attribute	Description attribute
1	Red (Dim Value 101)	10	Dark ruby in color, with extremely ripe…
2	Red (Dim Value 101)	12	Dense, rich and complex describes this '96 California…
3	White (Dim Value 102)	19	Dense and vegetal, with celery, pear, and spice flavors…
4	Other (Dim Value 103)	20	Big, ripe and generous, layered with honey…

Example 1

Assume that the following query is created:

```
<cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="RangeFilterOperator">
  <cs:RangeFilter AttributeName="Price">
     <cs:LowerBound Inclusive="false">15</cs:LowerBound>
  </cs:RangeFilter>
</cs:Operator>
```

This request has a range filter specifying the Price standard attribute should be greater than 15 (with no managed attribute values specified). The following objects are returned:

- 2 records (records 3 and 4)
- 2 refinements represented by the attribute values (White and Other)

Example 2

This example uses the following query:

```
<cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="RangeFilterOperator">
  <cs:RangeFilter AttributeName="Red">
     <cs:UpperBound Inclusive="false">11</cs:UpperBound>
  </cs:RangeFilter>
</cs:Operator>
```

This request specifies the Red managed attribute value and a range filter specifying a price less than 11. The following objects are returned:

- 1 record (record 1)
- (No additional refinements)

Example 3

This query:

```
<cs:Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="RangeFilterOperator">
  <cs:RangeFilter AttributeName="Price">
     <cs:LowerBound Inclusive="false">9</cs:LowerBound>
```

```
    <cs:UpperBound Inclusive="false">13</cs:UpperBound>
  </cs:RangeFilter>
</cs:Operator>
```

would return records 1 and 2 from the sample record set.

Part 4

Attribute Features

Chapter 9

Working with Refinements

This section provides information on handling and displaying refinements in your Web application.

About refinements

Endeca standard and managed attributes are referred to as *refinements*. They become refinements once the user utilizes them to refine the result set. Refinements can be used for navigation and search.

Endeca standard and managed attributes are similar in that they both:

- Support navigation.
- Are usually generated from a record's source attributes.
- Consist of key/value pairs (standard attribute name/standard attribute value, managed attribute name/managed attribute value).
- Can be searched and displayed using their display names.
- Can have a multi-level hierarchy.

Displaying refinements in Latitude Studio

Each component that is affected by the **Guided Navigation** component uses refinements and information received from refinements computation, such as the order of refinements or a number of refinements for a given attribute.

For additional information on configuring Latitude Studio components that use refinement information, including the **Guided Navigation** component, see the *Latitude Studio Power User's Guide.*

Configuring managed attributes for query refinement

You must configure a managed attribute for query refinement.

In the managed attribute's DDR (Dimension Description Record), the `mdex-dimension_EnableRe¬finements` attribute must be set to `true`, so that refinements will be displayed. If the configuration attribute is set to `false`, refinements will not be displayed (i.e., the managed attribute will be hidden).

If a managed attribute is created and used to classify records, but no records are classified with any corresponding values, that managed attribute will not be available as a refinement, because it is not related to the resulting record set in any way.

Notes

No configuration is needed for Endeca standard attributes. Assuming that the standard attribute is used to classify records, the corresponding refinement values will be available to create or refine a query.

There are no MDEX Engine configuration flags necessary to enable the basic displaying of refinements.

Configuring the global order of refinements

You configure the global order for the values of a refinement in the Latitude Data Integrator.

The `system-navigation_Sorting` attribute in the PDR for the selected attribute controls the global order of values in a refinement. This attribute can have the following values:

- **lexical** sorts refinement values by natural order.
- **record-count** sorts refinement values in descending order, by the number of records available for each refinement. This is the default.

For information on how to configure the global order of refinements, see the *Latitude Data Integrator Guide*.

Configuring refinement counts

You configure whether to show record counts for an attribute by changing the value in the `system-navigation_ShowRecordCounts` attribute in the PDR, using the Latitude Data Integrator.

The `system-navigation_ShowRecordCounts` attribute in the PDR specifies whether to show record counts for a refinement. The valid settings for this attribute are:

- `true` means that record counts are enabled and will display. This is the default.
- `false` means that record counts are disabled and will not be displayed.

To configure refinement counts for any attribute, see the *Latitude Data Integrator Guide*.

About multi-select attributes

A multi-select attribute is an attribute that can be present on records multiple times, with different values, in a single navigation state.

There are two forms of multi-select attributes:

- if the navigation state contains multiple values from a multi-select-and attribute, then all of the records in that state contain all of the selected values for that attribute.
- If the navigation state contains multiple values from a multi-select-or attribute, then all of the records in that state contain at least one of the selected values.

This has consequences for the refinements that the MDEX Engine generates as well as the navigation state: the MDEX Engine allows you to select additional values for a multi-select attribute.

In the MDEX Engine, you can configure an attribute as multi-select by changing the values of the `system-navigation_Select` attribute on the Property Description Record for this attribute.

The multi-select feature is only fully supported for those attributes (standard or managed) that do not contain a hierarchy.

Related Links

Configuring multi-select attributes on page 87
You configure whether an attribute is multi-select by changing the value in the `system-navigation_Select` attribute of a PDR, using the Latitude Data Integrator.

Handling multi-select attributes in an application on page 87
The behavior of multi-select attributes may require changes in the UI.

Configuring multi-select attributes

You configure whether an attribute is multi-select by changing the value in the `system-navigation_Select` attribute of a PDR, using the Latitude Data Integrator.

The `system-navigation_Select` attribute of a PDR can have the following settings:

- **multi-and** configures the attribute as a multi-select AND attribute. This means that a current navigation state represents the intersection of the records returned from the multiple selections of values on that attribute.
- **multi-or** configures the attribute as a multi-select OR attribute. This means that a current navigation state represents the union of the records returned from the multiple selections of values on that attribute.
- **single** configures the attribute as a single-select attribute. This means that only one value can be selected for an attribute at a time. This is the default.

To configure a multi-select attribute, see the *Latitude Data Integrator Guide*.

Handling multi-select attributes in an application

The behavior of multi-select attributes may require changes in the UI.

The fact that an attribute is tagged as multi-select should be transparent to the application developer. There is no special development required to enable multi-select attributes, and there are no query parameters that are specific to multi-select attributes.

However, the semantics of how the MDEX Engine interprets navigation queries and returns available refinements changes once an attribute is tagged as multi-select. After tagging an attribute as multi-select, the MDEX Engine will then allow multiple attribute values from the same attribute to be added to the navigation state.

The MDEX Engine behaves differently for the two types of multi-select managed attributes:

- Multi-select AND managed attributes. The MDEX Engine treats the list of attribute values selected from a `multi-and` attribute as a Boolean AND operation. That is, the MDEX Engine will return all records that satisfy the Boolean AND of all the attribute values selected from a `multi-and` attribute (that is, all records that have been tagged with "Apple" AND "Apricot"). The MDEX Engine will also continue to return refinements for a `multi-and` attribute. The list of available refinements

will be the set of attribute values that have not been chosen, and are still valid refinements for the results.

- Multi-select OR managed attributes. A `multi-or` managed attribute is analogous to a `multi-and` attribute, except that a Boolean OR operation is performed instead (that is, all records that have been tagged with "Apple" OR "Apricot"). The MDEX Engine will always return all attribute values for a `multi-or` attribute that have not already been selected – the set of refinements does not correlate to the set of remaining records. Also note that as more `multi-or` attribute values are added to the navigation state, the set of record results gets larger instead of smaller, because adding more terms to an OR expands the set of results that satisfy the query.

Avoiding dead-end query results

Be careful when rendering the selected managed attribute values of `multi-or` managed attributes. It is possible to create an interface that might result in dead-ends when removing selected attribute values.

Consider this example: Managed attribute Alpha has been flagged as `multi-or`, and contains values 1 and 2. Attribute Beta contains value 3.

Assume the user's current query contains all three values. The user's current navigation state would represent the query:

```
"Return all records tagged with (1 or 2) and 3"
```

If the user then removes one of the values from Attribute Alpha, a dead end could be reached. For example, if the user removes value 1, the new query becomes:

```
"Return all records tagged with 2 and 3"
```

This could result in a dead end if no records are tagged with both value 2 and 3.

Due to this behavior, it is recommended that the UI be designed so that the user must be forced to remove all values from a `multi-or` managed attribute when making changes to the list of selected attribute values.

Performance impact for multi-select managed attributes

Tagging an attribute as multi-select does not affect performance. However, when making decisions about when to tag an attribute as multi-select, keep the following in mind: Users will take longer to refine the list of results, because each selection from a multi-select managed attribute still allows for further refinements within that attribute. Also, refinements for `multi-or` attribute are more expensive.

Refinement counts for multi-or refinements

Refinement counts on a refinement that is multi-or indicate how many records in the result set will be tagged with the refinement if you select it. When there are no selections made yet, the refinement count equals the total number of records in the result set if that refinement were selected. However, for subsequent refinements, the refinement count may differ from the total results set.

Consider the following example which illustrates this use case. A `cuisine` refinement is configured as multi-or. In the data set, there are 2 records that have assignments only to a `Chinese` attribute, and 3 records that have assignments only to a `Japanese` attribute. There is also 1record that has assignments on both of these attributes.

When the user requests `Chinese` or `Japanese` as refinements during navigation, the resulting record list includes all 6 records, out of which 2 have only `Chinese` attribute, 3 have only `Japanese` attribute, and 1 has both:

Records	Assignment on a `Chinese` attribute	Assignment on a `Japanese` attribute
1	x	
2	x	
3	x	x
4		x
5		x
6		x

If the user first selects only `Chinese`, the navigation state shows that there is one remaining follow-on refinement (Japanese) with the refinement count of 4 records (3 with only `Japanese` assignment on a attribute and 1that has both `Chinese` and `Japanese` attribute assignments on them). When the user navigates on that refinement, the resulting record list includes all 6 records. This illustrates that a record count for a `Japanese` refinement shows the number of records (4) tagged with that refinement, within the entire record set (6).

About externally managed attributes

Endeca applications can use managed attributes created with a taxonomy management tool.

You can also import or otherwise access externally managed attributes. For details on loading these attributes, see the "Loading Records with the Data Ingest Web Service" chapter in the *Latitude Data Ingest API Guide*.

Performance impact for displaying refinements

Run-time performance of the MDEX Engine is directly related to the number of refinement values being computed for display.

Only request refinement values if you are planning to display them in the front-end application. If any refinement values are being computed by the MDEX Engine but not being displayed by the application, this negatively affects performance. Attributes containing large numbers of refinements also affect performance.

The following aspects affect the display of refinements:

- The `Expose` attribute in the `RefinementConfig` element. This attribute is optional and defaults to **false**. Setting this attribute to **true** for a refinement may make a query more expensive to compute (if the refinement has a large number of values).

  Note: The elements for which you specify these attributes belong to the `NavigationMenuConfig` element.

- The `ExposeAllRefinements` in the `ContentElementConfig` element. This attribute is optional and defaults to **false**.

  The worst-case scenario for run-time performance is having a data set with a large number of refinements, each containing a large number of refinement values, and setting the `ExposeAllRe¬`

`finements` attribute to **true**. This would create a page with an overwhelming number of refinement choices for the user.

Performance impact of refinement ordering

You can use the `--esampmin` option with the Dgraph, to specify the minimum number of records to sample during refinement computation.

This option is useful because sampling the entire navigation state during the refinement computation can be one of the more performance intensive operations for the MDEX Engine.

For most applications, larger values for `--esampmin` reduce performance without improving the quality of refinement ordering. For some applications with extremely large, non-hierarchical attributes (if they cannot be avoided), larger values can meaningfully improve refinement ordering quality with minor performance cost.

Performance impact of refinement counts

Dynamic statistics on records are expensive computations for the MDEX Engine.

You should only enable a managed attribute for dynamic statistics if you intend to use the statistics in your Endeca-enabled Web application. Because the Dgraph does additional computation for additional statistics, there is a performance cost for those that you are not using.

Working with refinements using the API

This section provides examples of Conversation Web Service requests and responses that let you retrieve various aspects of the refinement configuration — refinements themselves, their order, counts, and special types of refinements, such as those that are multi-select.

Related Links

Retrieving refinements with the API on page 91
Displaying attribute values for your refinements is the core concept behind Guided Navigation.

Refinements configuration format on page 91
Use the `RefinementConfig` element of the Conversation Web Service request to expose attribute values for refinements.

Retrieving attributes that have refinements on page 94
The first step in displaying refinements is to retrieve the attributes that potentially have refinements.

Creating a new query from refinement attribute values on page 95
Once refinement attribute values have been retrieved, these attribute values typically are used to create additional refinement Navigation queries.

Limiting the number of refinements to be displayed on page 97
If there are too many refinements to be returned per refinement, you can limit the number of displayed refinements.

Retrieving the full path of hierarchical refinements on page 98

For attributes that contain hierarchy, you can request hierarchy information about a refinement with the `ReturnFullPath` attribute.

Retrieving refinements with the API

Displaying attribute values for your refinements is the core concept behind Guided Navigation.

After a user creates a query using record and/or value search, only valid remaining refinement values are provided to the user to refine that query. This allows the user to reduce the number of matching records without creating an invalid query.

To display refinements, they need to be requested, that is, included in the Conversation Web Service request that describes the navigation menu. If the refinements belong to a group, this group needs to be requested. The following example shows the request in which a group of refinements is requested. This example assumes that the group "Wine Characteristics" exists and includes the refinements `WineType`, `Year` and `Scope`:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
    Id="NavigationMenu"
    HandlerFunction="NavigationMenuHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <RefinementGroupConfig Name="Wine Characteristics" Expose="true"/>
  </ContentElementConfig>
</Request>
```

The refinements `WineType`, `Year` and `Scope` will be returned along with the group "Wine Characteristics" in which they are included.

If you would like to retrieve refinements that are not explicitly included in any user-configured groups, you can request a group `system-navigation_InternalGroup`. This group exists in the MDEX Engine and includes all refinements that are not members of any other groups.

Refinements configuration format

Use the `RefinementConfig` element of the Conversation Web Service request to expose attribute values for refinements.

The `RefinementConfig` element of the Conversation Web Service request specifies which attribute, out of all valid attributes returned with a Navigation query, should return actual refinement values. Note that only the top-level refinement values are returned.

For managed attributes, if a managed attribute value is a parent, you can also use the `RefinementConfig` element with that attribute value and return its child attribute values (again, only the top-level child attribute values are returned).

The `RefinementConfig` element is included in the `RefinementGroupConfig`. This parent element returns refinements for groups.

RefinementConfig format

The basic `RefinementConfig` format is shown in this example:

```
<RefinementConfig
   Name="WineType"
   Spec="/"
   Expose="true"
   OrderByRecordCount="true"
   MaximumCount="100">
</RefinementConfig>
```

The descriptions of the attributes are:

Attribute	Description
`Name`	Required. The name of the attribute value. Specifying a root managed attribute value name is the same as specifying a name of the managed attribute.
`Spec`	Optional. The attribute value spec. For a hierarchical managed attribute, refinements will be returned for any child values of this spec.
`Expose`	Optional. Specify `true` to expose refinements (the default) or `false` to just show the root refinement.
`OrderByRecordCount`	Optional. Specify `true` to use dynamic ranking to order by record count or `false` to use the default order from the MDEX Engine.
`MaximumCount`	Optional. An integer that specifies a maximum limit on the number of refinements returned per attribute. If this setting is not specified, the number of refinements returned per attribute in the Conversation Web Service response is dictated by a value specified in the `MaximumRefinementCount` attribute in the `NavigationMenuConfig` element in the Conversation Web Service request. Further, if that value is not specified, the default is 10.

The `RefinementConfig` element is used in a `NavigationMenuConfig` element, as in this example. It exposes refinement values for the WineType attribute that is part of the group "Wine Characteristics":

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <State/>
   <ContentElementConfig xsi:type="NavigationMenuConfig"
     Id="NavigationMenu"
     HandlerFunction="NavigationMenuHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
```

```
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
       <RefinementConfig Name="WineType" Expose="true" MaximumCount="100"
/>
     </RefinementGroupConfig>
   </ContentElementConfig>
</Request>
```

Note that you can use multiple `RefinementConfig` elements in a `RefinementGroupConfig`, as in this example that for the WineType and Designation managed attributes:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
   Id="NavigationMenu"
   HandlerFunction="NavigationMenuHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
     <RefinementConfig Name="WineType" Expose="true" MaximumCount="20" />

     <RefinementConfig Name="Designation" Expose="true" MaximumCount="20"
/>
   <RefinementGroupConfig>
  </ContentElementConfig>
</Request>
```

Notes on RefinementConfig

Keep in mind that the `RefinementConfig` element is an optional query parameter. However, attributes for which `RefinementConfig` is not included will not return refinements. The `Expose` attribute is also optional and defaults to `false`. `Expose="false"` helps improve performance.

For example, in a simple data set with three managed attributes `WineType`, `Year` and `Score` and a user-defined group "Wine Characteristics", the query:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
     Id="NavigationMenu"
     HandlerFunction="NavigationMenuHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <RefinementGroupConfig Name="Wine Characteristics" Expose="true"/>
  </ContentElementConfig>
  </ContentElementConfig>
</Request>
```

will return all attributes in the group but no refinement attribute values. This is faster for the MDEX Engine to compute, and returns only root managed attribute values.

However, this query for the `WineType` managed attribute:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
     Id="NavigationMenu"
     HandlerFunction="NavigationMenuHandler"
```

```
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <RefinementGroupConfig Name="Wine Characteristics" Expose="true"/>
    <RefinementConfig Name="WineType" Expose="true" />
    </ContentElementConfig>
</Request>
```

will return all three managed attributes (since they are included in the "Wine Characteristics" group), as well as the top-level refinement values for the `WineType` managed attribute (such as `Red`, `White`, and `Other`). This is slightly more expensive for the MDEX Engine to compute, and returns the three root managed attribute values (`WineType`, `Year`, and `Score`) as well as the top-level refinement attribute values for `WineType`, but is necessary for selecting a valid refinement.

A more advanced query option returns all the top-level managed attribute value refinements for all attributes requested (instead of a single attribute). This option involves setting the `ExposeAllRefine¬ments` attribute to `true`. If an application sets this attribute to `true`, the query:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
     HandlerFunction="NavigationMenuHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

     Id="NavigationMenu"
     ExposeAllRefinements="true"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <RefinementGroupConfig Name="Wine Characteristics" Expose="true"/>
  </ContentElementConfig>
</Request>
```

will return three managed attributes (`WineType`, `Year`, and `Score`) as well as all valid top-level refinement attribute values for each of these managed attributes (`Red`, `White`, `Other` for `Wine Type`; `1999`, `2001`, `2003` for `Year`; and `70-80`, `80-90`, `90-100` for `Score`).

This is the equivalent of the query:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <State/>
   <ContentElementConfig xsi:type="NavigationMenuConfig"
      Id="NavigationMenu"
      HandlerFunction="NavigationMenuHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
       <RefinementConfig Name="WineType" Expose="true" />
       <RefinementConfig Name="Year" Expose="true" />
       <RefinementConfig Name="Score" Expose="true" />
      </RefinementGroupConfig>
   </ContentElementConfig>
</Request>
```

This is the most expensive type of query for the MDEX Engine to compute, and returns three root managed attribute values as well as the nine top-level refinement managed attribute values, creating a larger network and page size strain. This method, however, is effective for creating custom navigation solutions that require all possible refinement attribute values to be displayed at all times.

Retrieving attributes that have refinements

The first step in displaying refinements is to retrieve the attributes that potentially have refinements.

Types of refinements

Refinement attributes contain refinement attribute values for the current record set, including both *standard refinements* and *implicit refinements*.

- Standard refinements refine the record set when selected.
- Implicit refinements are those attribute values that are assigned to all records in the current result set and whose selection, therefore, does not narrow the results. The attribute values can be from standard attributes or managed attributes.

Retrieving standard and implicit refinements

Refinements (both standard and implicit) are returned in a `NavigationMenu` content element, that in turn contains a`NavigationMenuItemGroup` element with `NavigationMenuItem` elements for each managed attribute that has refinements.

This example shows the `NavigationMenuItem` element for the WineType attribute:

```
<cs:NavigationMenuItem
    Name="WineType"
    Display Name="Wine Type"
    MultiSelect="Or" HasMore="false">
    <cs:ExposureControl Exposed="true">
     <cs:Operator
      OwnerId="NavMenu" xsi:type="cs:RefinementHideOperator"
      Name="Perfect"
      Spec="/"
      Group="Wine Characteristics"/>
    </cs:ExposureControl>
    <cs:Refinement Name="Red" Spec="/Red" Label="Red" Count="40">
     <cs:Operator xsi:type="cs:RefinementOperator" Name="Red" Spec="/Red"/>

    </cs:Refinement>
    <cs:Refinement Name="White" Spec="/White" Label="White" Count="50">
      <cs:Operator xsi:type="cs:RefinementOperator" Name="White"
Spec="/White"/>
    </cs:Refinement>
   <cs:RootDimensionValue DimensionName="Sparkling" Spec="/"/>
</cs:NavigationMenuItem>
```

Each refinement is returned in a `Refinement` element, as shown in this example for the Red managed attribute value:

```
<cs:Refinement Name="Red" Spec="/Red" Label="Red" Count="18">
   <cs:Operator xsi:type="cs:RefinementOperator" Name="Red" Spec="/Red"/>
</cs:Refinement>
```

The `Count` element indicates that eighteen records would be in the result set if you were to refine on this attribute value.

Creating a new query from refinement attribute values

Once refinement attribute values have been retrieved, these attribute values typically are used to create additional refinement Navigation queries.

First, consider this request in which the `WineType` refinement is requested and exposed:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
 <ContentElementConfig xsi:type="NavigationMenuConfig"
```

```
  Id="NavigationMenu"
  HandlerFunction="NavigationMenuHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
    <RefinementConfig Name="WineType" Expose="true"/>
   </RefinementGroupConfig>
  </ContentElementConfig>
</Request>
```

It returns the following query results. Notice that the query results show the `WineType` refinement and the refinement values on it — `White` and `Sparkling`.

```
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
 <cs:Request>
  <State xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <ContentElementConfig xsi:type="NavigationMenuConfig" Id="NavigationMenu"
 HandlerFunction="NavigationMenuHandler" HandlerNamespace="http://www.ende¬
ca.com/MDEX/conversation/handlers/2010" xmlns="http://www.endeca.com/MDEX/con¬
versation/2010" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
    <RefinementConfig Name="WineType" Expose="true"
xmlns:ns="http://www.endeca.com/MDEX/conversation/2010"/>
   </RefinementGroupConfig>
  </ContentElementConfig>
  </cs:Request>
  <cs:ContentElement xsi:type="cs:NavigationMenu" Id="NavigationMenu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <cs:NavigationMenuItemGroup Name="Wine Characteristics" HasRefinablePRop¬
erties="true">
    <cs:NavigationMenuItem Name="WineType" DisplayName="WineType" MultiSe¬
lect="Or" HasMore="false">
        <cs:ExposureControl Exposed="true">
         <cs:Operator OwnerId="NavigationMenu" xsi:type="cs:RefinementHide¬
Operator"
              Name="WineType" S
              Spec="/"
              Group="Wine Characteristics"/>
         </cs:ExposureControl>
        <cs:Refinement Name="WineType" Spec="/Red" Label="Red" Count="18">

          <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red"/>
         </cs:Refinement>
         <cs:Refinement Name="WineType" Spec="/White" Label="White"
Count="40">
          <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red"/>
         </cs:Refinement>
        <cs:Refinement Name="WineType" Spec="/Sparkling" Label="Sparkling"
 Count="50">
          <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red"/>
         </cs:Refinement>
      <cs:RootDimensionValue DimensionName="WineType" Spec="/"/>
    </cs:NavigationMenuItem>
   </cs:NavigationMenuItemGroup>
  </cs:ContentElement>
</cs:Results>
```

Based on this result, a follow-on request creates an additional refinement Navigation query. It uses the refinement operator to request `Red`, to let you further refine to `WineType Red`.

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <State/>
  <ContentElementConfig xsi:type="NavigationMenuConfig" Id="NavigationMenu"
 HandlerFunction="NavigationMenuHandler" HandlerNamespace="http://www.ende¬
ca.com/MDEX/conversation/handlers/2010">
    <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
     <RefinementConfig Name="WineType" Expose="true"/>
    </RefinementGroupConfig>
   </ContentElementConfig>
  <Operator xsi:type="RefinementOperator" Name="WineType" Spec="/Red"/>
</Request>
```

Limiting the number of refinements to be displayed

If there are too many refinements to be returned per refinement, you can limit the number of displayed refinements.

Generally, when the request from the Conversation Web Service asks for attributes to return in response to a query, it asks for all of them that were requested with a `RefinementGroupConfig` element.

To provide a meaningful navigation experience, the MDEX Engine returns only those attributes that have refinements on them and that are not filtered by precedence rules. In other words, those attributes are returned are based on the navigation state.

If there are too many refinements to be returned per attribute, you can limit the number of them that are displayed in the navigation menu of the Conversation Web Service request (`NavigationMenuConfig`).

You can do this in a global setting or per each refinement value. The following statements describe the logic used by the Conversation Web Service to identify the number of refinements to be displayed:

- Per attribute configuration. You can specify a number of refinements to display per attribute. For each, you can optionally specify the number in the attribute `MaximumCount` in the `RefinementConfig` element.
- Global configuration, for all attributes in a particular Navigation Menu. If the number for each attribute is not specified, the Conversation Web Service response uses the global setting that you can specify in the `MaximumRefinementCount` attribute of `RefinementConfig` in `ContentElementConfig`. The setting is per content element, not per query.
- Further, if neither of the settings is specified, the number defaults to 10.

For example, in this configuration for the navigation menu, `MaximumRefinementCount` is set to 15. In addition, for the `WineType` refinement value, `MaximumCount` is set to 40. `MaximumCount` is not set in each of the other refinement values.

```
<ContentElementConfig xsi:type="NavigationMenuConfig"
 HandlerFunction="NavigationMenuHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
 Id="NavigationMenu"
 MaximumRefinementCount="15"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
   <RefinementConfig Name="WineType" MaximumCount="40"/>
   <RefinementConfig Name="Year"/>
   <RefinementConfig Name="Score"/>
  </RefinementGroupConfig>
</ContentElementConfig>
```

This request returns up to 40 refinement values for `WineType`. It returns 15 refinement values for each of the other two refinement values (`Year` and `Score`).

The attribute `HasMore` (with possible boolean values `true` or `false`) in the response specifies whether the total refinement count exceeds the value returned with the `MaximumRefinementCount`. The value from this attribute can be utilized in the settings for Latitude Studio components designed to show refinement counts.

The following example shows a response where `HasMore` attribute is set to `true` in the `NavigationMenuItem` type of the Conversation Web Service response:

```
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
  <cs:Request>
    <FilterState xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <ContentElementConfig xsi:type="NavigationMenuConfig" Id="NavigationMenu"
 HandlerFunction="NavigationMenuHandler" HandlerNamespace="http://www.ende¬
ca.com/MDEX/conversation/handlers/2010" xmlns="http://www.endeca.com/MDEX/con¬
versation/2010" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
       <RefinementConfig Name="WineType" MaximumCount="1" Expose="true"
xmlns:ns="http://www.endeca.com/MDEX/conversation/2010"/>
      </RefinementGroupConfig>
      </ContentElementConfig>
  </cs:Request>
  <cs:ContentElement xsi:type="cs:NavigationMenu" Id="NavigationMenu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <cs:NavigationMenuItemGroup Name="Wine Characteristics" HasRefinablePRop¬
erties="true">
    <cs:NavigationMenuItem Name="WineType" DisplayName="WineType" MultiSe¬
lect="Or" HasMore="true">
         <cs:ExposureControl Exposed="true">
           <cs:Operator OwnerId="NavigationMenu"
              xsi:type="cs:RefinementHideOperator"
              Name="WineType" Spec="/"
              Group="Wine Characteristics"/>
         </cs:ExposureControl>
        <cs:Refinement Name="WineType" Spec="/Red" Label="Red" Count="18">

          <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red"/>
         </cs:Refinement>
      <cs:RootDimensionValue DimensionName="WineType" Spec="/"/>
    </cs:NavigationMenuItem>
   </NavigationMenuItemGroup>
  </cs:ContentElement>
</cs:Results>
```

Retrieving the full path of hierarchical refinements

For attributes that contain hierarchy, you can request hierarchy information about a refinement with the `ReturnFullPath` attribute.

For example, if a Wineries managed attribute contains four levels of hierarchy (Country, State, Region, Winery) and the current query is at the region level (Sonoma Valley), the full path of hierarchical refinements can be represented by the following list:

```
Wineries > United States > California > Sonoma Valley
```

Refinement values, in this case specific wineries, may still exist for the `Sonoma Valley` refinement to refine the query even further.

To request the full path of hierarchical refinements, use the `ReturnFullPath` attribute on `NavigationMenuConfig`. The `ReturnFullPath` has the following values:

`ReturnFullPath`	Specifies whether to return the full path of hierarchical refinements with the response. This setting is relevant in navigation queries for refinements and breadcrumbs. If set to `true`, the returned refinement contains the full path to its parent refinement values, as in `Wine > Red > Merlot`. If set to `false`, returns only the refinement, without the path to its ancestors. The default is `false`.

The format of the `NavigationMenuConfig` is shown in this example. It uses the `ReturnFullPath` attribute set to `true`:

```
 <ContentElementConfig xsi:type="NavigationMenuConfig"
     Id="NavigationMenu"
     HandlerFunction="NavigationMenuHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     ReturnFullPath="true">
     <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
      <RefinementConfig Name="WineType" Expose="true" MaximumCount="3"/>
     </RefinementGroupConfig>
  </ContentElementConfig>
```

For a flat managed attribute with no hierarchy, the refinement parent will always be the attribute root, because there would be no further refinements if a value had already been selected for the attribute.

Refinements for a given managed attribute can only be returned from the MDEX Engine on the same level within the attribute. For example, the MDEX Engine could never return a list of refinement choices that included a mix of countries, states, and regions. In all cases where hierarchy is explicitly defined for a managed attribute, only refinements on an equal level of hierarchy will be returned for a given query.

Example request

The following example request in the Conversation Web Service illustrates how to retrieve a full path of hierarchical refinements for an attribute:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State>
    <SelectedRefinementFilter Name="WineType" Spec="/Red"/>
  </State>
  <ContentElementConfig xsi:type="NavigationMenuConfig"
     Id="NavigationMenu"
     HandlerFunction="NavigationMenuHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     ReturnFullPath="true">
    <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
     <RefinementConfig Name="WineType" Expose="true" MaximumCount="3"/>
    </RefinementGroupConfig>
  </ContentElementConfig>
</Request>
```

Example response

The following response returns a list of hierarchical refinements:

```
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
   <cs:Request>
      <State xmlns="http://www.endeca.com/MDEX/conversation/2010"
xmlns:ns="http://www.endeca.com/MDEX/conversation/2010">
         <SelectedRefinementFilter Name="WineType" Spec="/Red"/>
      </State>
     <ContentElementConfig xsi:type="NavigationMenuConfig" Id="Navigation¬
Menu" HandlerFunction="NavigationMenuHandler" HandlerNamespace="http://www.en¬
deca.com/MDEX/conversation/handlers/2010" ReturnFullPath="true"
xmlns="http://www.endeca.com/MDEX/conversation/2010"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <RefinementGroupConfig Name="Wine Characteristics" Expose="true">

         <RefinementConfig Name="WineType" Expose="true" MaximumCount="3"
xmlns:ns="http://www.endeca.com/MDEX/conversation/2010"/>
         </RefinementGroupConfig>
     </ContentElementConfig>
   </cs:Request>
<cs:ContentElement xsi:type="cs:NavigationMenu" Id="NavigationMenu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <cs:NavigationMenuItemGroup Name="Wine Characteristics" HasRefinablePRop¬
erties="true">
  <cs:NavigationMenuItem Name="WineType" DisplayName="WineType" MultiSe¬
lect="None" HasMore="false">
         <cs:ExposureControl Exposed="true">
            <cs:Operator OwnerId="NavigationMenu"
             xsi:type="cs:RefinementHideOperator"
             Name="WineType" Spec="/"
             Group="Wine Characteristics"/>
         </cs:ExposureControl>
         <cs:Refinement Name="WineType" Spec="/Red/Merlot" Label="Merlot"
Count="19">
            <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red/Merlot"/>
         </cs:Refinement>
         <cs:Refinement Name="WineType" Spec="/Red/Shiraz" Label="Shiraz"
Count="15">
            <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red/Shiraz"/>
         </cs:Refinement>
         <cs:Refinement Name="WineType" Spec="/Red/PinotNoir" Label="Pinot
 noir" Count="8">
            <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
Spec="/Red/PinotNoir"/>
         </cs:Refinement>
         <cs:RootDimensionValue DimensionName="WineType" Spec="/"/>
         <cs:FullPath>
            <cs:DimensionValue>
               <cs:DimensionValue DimensionName="WineType" Spec="/">Wine¬
Type</cs:DimensionValue>
             <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
 Spec="/"/>
            </cs:DimensionValue>
            <cs:DimensionValue>
               <cs:DimensionValue DimensionName="WineType"
Spec="/Red">Red</cs:DimensionValue>
```

```
            <cs:Operator xsi:type="cs:RefinementOperator" Name="WineType"
 Spec="/Red"/>
          </cs:DimensionValue>
        </cs:FullPath>
      </cs:NavigationMenuItem>
    </cs:NavigationMenuItemGroup>
  </cs:ContentElement>
</cs:Results>
```

Retrieving the order of refinements with the API

A core capability of the MDEX Engine is the ability to dynamically order and present the most popular refinement values to the user.

There are two ways in which you can configure the display order of refinements in the Conversation Web Service:

- By specifying the value for `system-navigation_Sorting` in the PDR, for an attribute.
- By using query-time control of the display order specified in the `OrderByRecordCount` attribute in the `RefinementConfig` element of the Conversation Web Service request. Note that by using this method, you can override the `system-navigation_Sorting` settings for a given attribute.

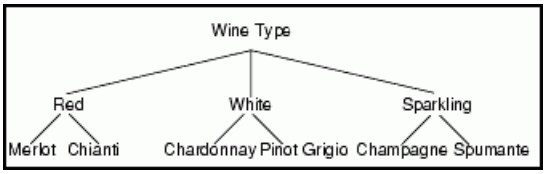
Using query-time control of refinement ordering

You can configure refinement ordering on a per-query basis.

The MDEX Engine lets you switch refinement ordering on and off on a per-query basis.

A use case for this refinement ordering would be an application that renders refinements as a tag cloud. Such an application may adjust the size of the tag cloud at query time, depending on user preferences or from which page the query originates.

You set the refinement ordering at the refinement value level that you want to control. For managed attributes, ordering will be applied to that attribute value and all its children. For example, assume that you have an attribute named WineType that has three child attribute values (named Red, White, and Sparkling), which in turn have two child attribute values each. The managed attribute hierarchy would look like this:



You would set the ordering depending on which level of the hierarchy you want to order and present, for example:

- If you set the ordering on the root attribute value (which has the same name and ID as the attribute itself), the refinements in the Red, White, and Sparkling attribute values will be returned.
- If there are multiple child attribute values, you can set an order on only one sibling. In this case, the refinements from the other siblings will not be exposed. For example, if you set an order on the Red attribute value, only the refinements of the Merlot and Chianti attribute values will be returned. The refinements from the White and Sparkling attribute values will be not be shown, even if you explicitly set orders for them.

The settings of the per query ordering of refinements are not persistent. That is, each query must have its own configuration, because it is not carried over from the previous query. Keep the following items in mind when using this feature:

Enabling the refinement order at query time

The `OrderByRecordCount` attribute sets the refinement order at query time.

Setting the `OrderByRecordCount` attribute to **true** in the `RefinementConfig` element sets the order in which refinements will be displayed, at query time, as in this example:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <State/>
   <ContentElementConfig xsi:type="NavigationMenuConfig"
      Id="NavigationMenu"
      HandlerFunction="NavigationMenuHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <RefinementGroupConfig Name="Wine Characteristics" Expose="true">
        <RefinementConfig
          Name="WineType"
          Expose="true"
          OrderByRecordCount="true"
          MaximumCount="100" />
      </RefinementGroupConfig>
    </ContentElementConfig>
   </Request>
```

This setting overrides the setting for refinement order that you can specify in the `system-naviga¬tion_Sorting` in the PDR for a refinement.

For details on the `RefinementConfig` element, see the "Refinements configuration format" topic in this chapter.

Retrieving refinement counts with the API

The application UI can display the number of records returned for refinements.

Refinement counts represent the number of records (in the current navigation state) available beneath a given refinement value. These counts are dynamically computed at run-time by the MDEX Engine and can be displayed in the user interface.

By providing the user with an indication of the number of records that will be returned for each refinement, refinement counts can enhance the Endeca application's navigation controls by providing more context at each point in the Endeca application.

A *refinement count* is the number of records that would be in the result set if you were to refine on an attribute value.

By default, all types of attributes (standard and managed) are enabled for refinement counts. Therefore, no further configuration is needed to display record counts. So long as there are attribute values returned for a given request, refinement value statistics will be returned as an attribute attached to each attribute value.

You can, however, disable refinement statistics for attributes in the `system-navigation_ShowRecord¬Counts` attribute on the PDR.

Retrieving refinement counts for records

Record counts are returned in a `Count` attribute.

Each refinement is returned in a `Refinement` element, as shown in this example:

```
<cs:Refinement Name="Red" Spec="/Red" Label="2" Count="18">
   <cs:Operator xsi:type="cs:RefinementOperator" Name="Red" Spec="/Red"/>
</cs:Refinement>
```

In the example, a record count of 18 is returned for the Red attribute value.

Retrieving multi-select refinements with the API

The MDEX Engine supports two types of multi-select attributes.

The default behavior of the MDEX Engine permits only a single value from an attribute to be added to the navigation state. By default, after a user selects a leaf refinement from any attribute, that attribute is removed from the list of refinements available for future refinement in the query results. For example, after selecting "Apple" from the Flavors attribute, the Flavors attribute is removed from the navigation controls.

However, sometimes it is useful at navigation time to allow the user to select more than one value from an attribute. For example, you can give a user the ability to show wines that have a flavor of "Apple" and "Apricot".

The MDEX Engine provides support for two types of multi-select attributes that apply Boolean logic to the values selected:

- **multi-and**
- **multi-or**

You can tag the attribute as **multi-and**, **multi-or**, or **single** by configuring the values of the `system-navigation_Select` attribute of a PDR.

Chapter 10

Using Breadcrumbs

The section discusses how to implement breadcrumbs.

About breadcrumbs

Breadcrumbs let you summarize any Guided Navigation selections, keyword searches, or range filters specified by the end user.

Breadcrumbs represent the following information that was passed to the navigation state by the Conversation Web Service response:

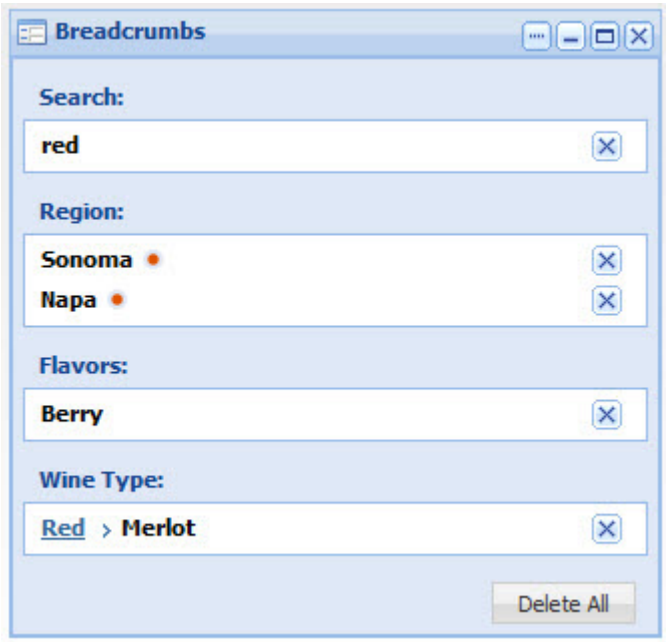
- Selected refinement values that were used to query for the current record set.
- Keyword searches that were used to query for the current record set.
- Range filters that have been selected for the query.

Any standard or managed attribute value available in the MDEX Engine can be selected as a breadcrumb.

Breadcrumbs honor record filters (such as security filters), but do not display them.

Breadcrumbs can reflect spelling correction and DYM information returned by the MDEX Engine in response to keyword search queries.

In Latitude Studio, the **Breadcrumbs** component lets you display breadcrumbs made with navigation queries (when users select refinement values or range filters for navigation), and keyword search queries.



For example, here is how user selections made in the **Guided Navigation** component are reflected in the **Breadcrumbs** component:

- Once the user selects a refinement in the **Guided Navigation** component, it is reflected as a breadcrumb in the **Breadcrumbs** component.
- The user can select an additional breadcrumb in the **Guided Navigation** component, thereby narrowing down the scope of the record set for the query.
- Alternatively, the user can remove a refinement value from the **Breadcrumbs** component, which increases the scope of the record set for the query.

Implementing breadcrumbs in Latitude Studio

This section describes how to configure and use breadcrumbs in a Latitude Studio application.

Related Links

Configuring breadcrumbs on page 106

To use breadcrumbs in Latitude Studio, you add a **Breadcrumbs** component. Note that any component that supports refinement works in conjunction with the **Breadcrumbs** component when displaying results.

Configuring breadcrumbs

To use breadcrumbs in Latitude Studio, you add a **Breadcrumbs** component. Note that any component that supports refinement works in conjunction with the **Breadcrumbs** component when displaying results.

The **Breadcrumbs** component requires a backing data source. There also must be a component that can be used to refine the data, such as the **Guided Navigation** component or **Search Box**.

Latitude Studio power users can add and configure a **Breadcrumbs** component.

To add the **Breadcrumbs** component and set the preferences on it in the Latitude Studio application:

1. In Latitude Studio, point the cursor at the Dock in the upper-right corner of the page.
2. From the drop-down menu, select **Add Component**.
   The **Add Component** dialog box opens.
3. In the **Add Component** dialog box, expand the **Latitude** category.
   A list of the available Latitude components appears.
4. Drag the **Breadcrumbs** component into the main page layout.
   The **Breadcrumbs** component is added with the message `"No refinements have been selected."`
5. To display the edit view for the component, click the **....** button. In the drop-down menu, click **Preferences**.
6. To bind a different data source to the component, select the data source from the drop-down list, then click **Update data source**.
7. In the **Multi-select collapse/expand threshold** field, set the number of selected attribute values after which the list can be collapsed.

   When end users select multiple values for an attribute, if they select more than this number, then on the **Breadcrumbs** component, the list of selected values is initially collapsed.

   End users can then display or hide the full list.
8. To save the changes to the configuration, click **Save Preferences**.
9. To exit the edit view, click **Return to Full Page**.

Implementing breadcrumbs with the API

This section describes how to issue queries requesting breadcrumbs using the Conversation Web Service API.

The Conversation Web Service returns breadcrumb results for these types of queries:

- Navigation
- Search
- Range filters

For more information on the Conversation Web Service interface, see the *MDEX Engine API Reference*.

Related Links

Retrieving breadcrumbs in a navigation query on page 108
An initial Conversation Service request that is made in response to a user-initiated navigation query (in which no selections have been made in the navigation state) does not yet return breadcrumbs. However, a subsequent request (in which the user made selections within the available attribute values) returns breadcrumbs.

Retrieving breadcrumbs in a search query on page 110
Breadcrumbs returned by the Conversation Web Service in response to a search query can reflect spelling correction and DYM (Did You Mean) information.

Retrieving breadcrumbs in a navigation query

An initial Conversation Service request that is made in response to a user-initiated navigation query (in which no selections have been made in the navigation state) does not yet return breadcrumbs. However, a subsequent request (in which the user made selections within the available attribute values) returns breadcrumbs.

The request for breadcrumbs is implemented with the `ContentElementConfig` element with the `BreadcrumbHandler`:

```
<ContentElementConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="BreadcrumbConfig"
ReturnFullPath="false"
HandlerFunction="BreadcrumbHandler"
HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
Id="Breadcrumbs" />
```

This element includes:

`ReturnFullPath`	Specifies whether to return the full path of hierarchical refinements with the response. This setting is relevant only in navigation queries that request breadcrumbs; it is ignored in search or range filter queries requesting breadcrumbs. If set to `true`, the returned breadcrumb contains the full path to its parent refinement values, as in `Wine > Red > Merlot`. If set to `false`, returns only the refinement, without the path to its ancestors. The default is `false`.
`BreadcrumbHandler`	Is the function that facilitates breadcrumb generation in the response. This function is required to return breadcrumbs.

If spelling is enabled in the MDEX Engine, and in addition to breadcrumbs, you want the Conversation Web Service response to contain supplemental information about spelling suggestions and DYM, a second `ContentElementConfig` with`SearchAdjustmentHandler` is required. If this element is included, spelling correction or DYM suggestions are returned with the breadcrumbs in the response.

Note: If spelling is enabled, spelling correction occurs for breadcrumb results even if `ContentElementConfig` with `SearchAdjustmentHandler` is not included; however, while spelling correction takes place, the spelling correction and DYM suggestions are not returned in the response.

In the response, breadcrumbs are returned in the order in which they were added (requested).

To request breadcrumbs for a navigation query:

In the Conversation Web Service request, specify the following:

- The selection for a specific refinement.
- The `ContentElementConfig` element for `BreadcrumbHandler`.
- (Optional). The `ContentElementConfig` element for `SearchAdjustments`.

In this example, the navigation state includes a selection of the `NumberOfDigits` refinement, and two `ContentElementConfig` elements, one for `BreadcrumbHandler` and one for `SearchAdjustmentHandler`:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
    <State/>
    <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="RefinementOperator" Spec="/2"
    Name="NumberOfDigits"/>
    <ContentElementConfig
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="BreadcrumbConfig"
    ReturnFullPath="true" HandlerFunction="BreadcrumbHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="Breadcrumbs"/>
    <ContentElementConfig
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchAdjustmentConfig"
    HandlerFunction="SearchAdjustmentHandler"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
    Id="SearchAdjustments"/>
    <PassThrough> ...</PassThrough>
</Request>
```

The Conversation Web Service result includes the original request with operators for `BreadcrumbHandler` and `SearchAdjustmentHandler` applied, followed by the `ContentElementConfig` element that lists attribute values identified as breadcrumbs, based on the user-selected navigation state.

Note: The result also includes the `GeneralizationOperator`. It enables the removal of the refinement (and thus the breadcrumb) in the user interface of Latitude Studio, if the user chooses to remove the previously selected refinement from the breadcrumb list.

```
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
  <cs:Request>
      <cs:State>
       <cs:SelectedRefinementFilter Name="NumberOfDigits" Spec="/2"/>
      </cs:State>
      <ContentElementConfig xmlns="http://www.endeca.com/MDEX/
      conversation/2010"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="BreadcrumbConfig" ReturnFullPath="true"
      HandlerFunction="BreadcrumbHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="Breadcrumbs"/>
      <ContentElementConfig xmlns="http://www.endeca.com/MDEX/conversa¬
tion/2010" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchAdjustmentConfig" HandlerFunction="SearchAdjustmentHandler"

      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="SearchAdjustments"/>
      <PassThrough>... </PassThrough>
  </cs:Request>
  <cs:ContentElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="cs:Breadcrumbs" Id="Breadcrumbs">
  <cs:RefinementBreadcrumb Name="NumberOfDigits"
     DisplayName="Number Of Digits" Spec="/2">
```

```
     <cs:DimensionValue>
      <cs:DimensionValue DimensionName="NumberOfDigits" Spec="/">
        NumberOfDigits
      </cs:DimensionValue>
     <cs:Operator xsi:type="cs:GeneralizationOperator" Name="" Spec="/"/>
    </cs:DimensionValue>
    <cs:DimensionValue>
     <cs:DimensionValue DimensionName="NumberOfDigits" Spec="/2">2
     </cs:DimensionValue>
    <cs:Operator xsi:type="cs:GeneralizationOperator" Name="" Spec="/2"/>
      </cs:DimensionValue>
      <cs:Operator xsi:type="cs:GeneralizationOperator"
      Name="NumberOfDigits" Spec="/2"/>
   </cs:RefinementBreadcrumb>
  </cs:ContentElement>
 <cs:ContentElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="cs:SearchAdjustments" Id="SearchAdjustments"/>
</cs:Results>
```

Retrieving breadcrumbs in a search query

Breadcrumbs returned by the Conversation Web Service in response to a search query can reflect spelling correction and DYM (Did You Mean) information.

The following requirements must be met to implement breadcrumbs that also return spelling correction and DYM information in response to a search query:

- The spelling must be enabled in the MDEX Engine. To enable spelling, after you install the MDEX Engine and run `mkmdex`, run the `admin?op=updateaspell` command.
- The request must include the `ContentElementConfig` element for the `BreadcrumbHandler`. This ensures that breadcrumbs are returned:

  ```
  <ContentElementConfig xmlns:xsi="http://www.w3.org/2001
  /XMLSchema-instance"
  xsi:type="BreadcrumbConfig"
  ReturnFullPath="false"
  HandlerFunction="BreadcrumbHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
  Id="Breadcrumbs" />
  ```

- If you would like to return DYM and spelling correction results with breadcrumbs, the request must include the `ContentElementConfig` element for the `SearchAdjustmentHandler`:

  ```
  <ContentElementConfig
  xmlns="http://www.endeca.com/MDEX/conversation/2010"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="SearchAdjustmentConfig"
  HandlerFunction="SearchAdjustmentHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
  Id="SearchAdjustments" />
  ```

In the response, breadcrumbs are returned in the order in which they were added.

Request and response with spelling correction
Breadcrumbs information returned by the Conversation Web Service can reflect spelling correction. The following example illustrates this case.

To request breadcrumbs in a search query that returns spelling correction:

In the request, specify the search keyword, and the `ContentElementConfig` elements for `BreadcrumbHandler` and `SearchAdjustmentsHandler`, as in the following abbreviated example.

This request example specifies a navigation state that includes a search for a user-entered word `fife`. It illustrates a search request with a breadcrumb that needs to be corrected for spelling:

```
<Request
xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
   <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="SearchOperator" Within="false">
    <SearchFilter Mode="All" Key="English">
     fife
    </SearchFilter>
   </Operator>
 <ContentElementConfig
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="BreadcrumbConfig" ReturnFullPath="true"
 HandlerFunction="BreadcrumbHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

 Id="Breadcrumbs"/>
 <ContentElementConfig
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="SearchAdjustmentConfig"
 HandlerFunction="SearchAdjustmentHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

 Id="SearchAdjustments"/>
 <PassThrough>...</PassThrough>
</Request>
```

The response from the Conversation Web Service contains the original request with search filter operators applied, the original (not yet spelling-corrected) term `fife`, and the `PopSearchOperator` needed for removing the refinement (if the user decides to remove this breadcrumb). Finally, the response also includes the automatically corrected term `five` in the `ContentElement` for `AppliedAdjustment` .

```
<cs:Results
xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
   <cs:Request>
       <cs:State>
         <SearchFilter
         xmlns="http://www.endeca.com/MDEX/conversation/2010"
          xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          Mode="All"
          Key="English">
           fife
         </SearchFilter>
       </cs:State>
       <ContentElementConfig
       xmlns="http://www.endeca.com/MDEX/conversation/2010"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:type="BreadcrumbConfig" ReturnFullPath="true"
       HandlerFunction="BreadcrumbHandler"
       HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="Breadcrumbs"/>
```

```
       <ContentElementConfig
       xmlns="http://www.endeca.com/MDEX/conversation/2010"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:type="SearchAdjustmentConfig"
       HandlerFunction="SearchAdjustmentHandler"
       HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010" Id="SearchAdjustments"/>
       <PassThrough> ... </PassThrough>
   </cs:Request>
     <cs:ContentElement
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:type="cs:Breadcrumbs" Id="Breadcrumbs">
       <cs:SearchBreadcrumb DisplayName="English">
         <cs:SearchFilter Key="English" Mode="All">
             fife
         </cs:SearchFilter>
       <cs:Operator xsi:type="cs:PopSearchOperator">
           <cs:SearchFilter Key="English" Mode="All">
             fife
           </cs:SearchFilter>
       </cs:Operator>
      </cs:SearchBreadcrumb>
     </cs:ContentElement>
          <cs:ContentElement
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="cs:SearchAdjustments" Id="SearchAdjustments">
         <cs:AppliedAdjustment>
             <cs:SearchFilter Key="English" Mode="All">
                fife
             </cs:SearchFilter>
         <cs:AdjustedTerms>
           five
         </cs:AdjustedTerms>
        </cs:AppliedAdjustment>
     </cs:ContentElement>
</cs:Results>
```

Request and response with DYM

Breadcrumbs information returned by the Conversation Web Service can reflect DYM suggestions. The following example illustrates this case.

To request breadcrumbs in a search query that returns DYM suggestions:

Specify the keyword search entry, the `ContentElementConfig` elements for `BreadcrumbHandler` and `SearchAdjustmentHandler`. The following abbreviated example contains a keyword search `jane`:

```
<Request
xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State />
  <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="SearchOperator" Within="false">
   <SearchFilter Mode="All" Key="Essay">
    jane
   </SearchFilter>
  </Operator>
  <ContentElementConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="BreadcrumbConfig" ReturnFullPath="true"
  HandlerFunction="BreadcrumbHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
```

```
    Id="Breadcrumbs" />
    <ContentElementConfig
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="SearchAdjustmentConfig"
    HandlerFunction="SearchAdjustmentHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

    Id="SearchAdjustments" />
    <PassThrough> ... </PassThrough>
  </Request>
```

The response reflects DYM results. In this example, the response includes the request with operators applied, followed by a DYM suggested term, `can` and by the `ApplySpellingSuggestionOperator` that actually replaces the keyword with the term suggested with DYM:

```
<cs:Results
xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
 <cs:Request>
   <cs:State>
    <SearchFilter xmlns="http://www.endeca.com/MDEX/conversation/2010"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    Mode="All" Key="Essay">
      jane
    </SearchFilter>
   </cs:State>
  <ContentElementConfig
  xmlns="http://www.endeca.com/MDEX/conversation/2010"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="BreadcrumbConfig" ReturnFullPath="true"
  HandlerFunction="BreadcrumbHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

  Id="Breadcrumbs" />
  <ContentElementConfig
  xmlns="http://www.endeca.com/MDEX/conversation/2010"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="SearchAdjustmentConfig"
  HandlerFunction="SearchAdjustmentHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

  Id="SearchAdjustments" />
  <PassThrough> ... </PassThrough>
 </cs:Request>
<cs:ContentElement
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="cs:Breadcrumbs" Id="Breadcrumbs">
  <cs:SearchBreadcrumb DisplayName="Essay">
     <cs:SearchFilter Key="Essay" Mode="All">
         jane
     </cs:SearchFilter>
     <cs:Operator xsi:type="cs:PopSearchOperator">
       <cs:SearchFilter Key="Essay" Mode="All">
         jane
       </cs:SearchFilter>
     </cs:Operator>
   </cs:SearchBreadcrumb>
  </cs:ContentElement>
```

```
  <cs:ContentElement
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="cs:SearchAdjustments" Id="SearchAdjustments">
    <cs:SuggestedAdjustment RecordCountIfApplied="15">
        <cs:SearchFilter Key="Essay" Mode="All">
          jane
        </cs:SearchFilter>
        <cs:SuggestedTerms>
          can
       </cs:SuggestedTerms>
      <cs:Operator xsi:type="cs:ApplySpellingSuggestionOperator">
       <cs:SearchFilter Key="Essay" Mode="All">
         jane
       </cs:SearchFilter>
       <cs:Replacement>
         can
      </cs:Replacement>
     </cs:Operator>
    </cs:SuggestedAdjustment>
  </cs:ContentElement>
</cs:Results>
```

Chapter 11

Using Attribute Groups

This chapter discusses how to implement attribute groups in the MDEX Engine.

About attribute groups

Attribute groups are ordered collections of attributes. They are stored in the MDEX Engine as records.

Attribute groups are useful for organizing a large number of attributes on the user interface of your Latitude Studio application. You can define a set of attribute groups to be displayed, assign attributes to each group, and determine the display order of the groups and attributes.

Because you define the attribute groups, you can group the attributes in any way that makes sense for your data.

You can assign an attribute to more than one of your attribute groups. There is also a default `Other` attribute group containing all of the attributes that you have not assigned to a group.

How attribute groups are used in Latitude Studio

In Latitude Studio, lists of attributes are displayed in attribute groups.

This includes:

- For power users, when configuring Latitude Studio components
- For end users, when viewing components such as the **Guided Navigation** component:

For information on using and configuring components in Latitude Studio, see the *Latitude Studio Power User's Guide*.

About configuring attribute groups

The Latitude Studio **Control Panel** includes an **Attribute Settings** component, which is used to configure attribute groups for each data source.

From the **Attribute Settings** component, power users can:

- Create and delete attribute groups
- Add and remove the attributes in each attribute group
- Set the default display order of the attribute groups and their attributes

For information on using the **Attribute Settings** component to configure attribute groups, see the *Latitude Studio Power User's Guide*.

Implementing attribute groups with the API

This section describes how to issue queries requesting attribute groups or lists of attribute groups using the Conversation Web Service API.

The Conversation Web Service returns attribute groups for those types of queries that return refinements (attributes). Any attributes that are returned from the Conversation Web Service as refinements are returned as part of attribute groups.

For more information on the Conversation Web Service interface, see the *MDEX Engine API Reference*.

Related Links

Retrieving attribute groups on page 117
Any request that asks for refinements is also requesting groups, if the attributes that are going to be returned are configured as part of groups.

Retrieving lists of attribute groups on page 119
To retrieve a lists of groups, use a request with `ContentElementConfig` of type `PropertyGroupListConfig`, which uses the `PropertyGroupListHandler`.

Retrieving attribute groups

Any request that asks for refinements is also requesting groups, if the attributes that are going to be returned are configured as part of groups.

The request for groups is implemented with the `RefinementGroupConfig` element of the Conversation Web Service request. This element contains one or more `RefinementConfig` elements that list which attributes, out of all valid attributes returned with a Navigation query, should return actual refinement values. Note that only the top-level refinement values are returned.

The complex type `RefinementGroupConfig` has the following format:

```
<complexType name="RefinementGroupConfig">
 <sequence>
  <element name="RefinementConfig" type="cs:RefinementConfig" minOccurs="0"
 maxOccurs="unbounded"/>
 </sequence>
  <attribute name="Name" type="cs:NonEmptyString" use="required"/>
  <attribute name="Expose" type="boolean" use="required"/>
  <attribute name="ExposeAllPropertyRefinements" type="boolean"/>
 </complexType>
```

The meanings of the attributes are:

Attribute	Description
`Name`	Required. The name of the group.
`Expose`	Required. Specify `true` to expose all top-level attributes in the group (the default) or `false` to just show the head of the refinement group. **Note:** If an attribute is a managed attribute, it contains a hierarchy of attributes under its root. Whether these nested attributes are exposed is not controlled by this element, and is instead controlled by the `Expose` attribute on the `RefinementConfig` element for each standard attribute within a managed attribute.
`ExposeAllPropertyRefinements`	Optional. If set to `true`, specifies whether to expose all attribute refinements underneath each managed attribute that has them. The default is `false` (if this attribute is not specified). This setting supersedes the `Expose` attribute on the `RefinementConfig` element for each attribute refinement.

Groups are returned in a NavigationMenuItemGroup element that contains one or more NavigationMenu elements, each of which returns refinements in the NavigationMenuItem. Here is the format for the NavigationMenuItemGroup:

```
<complexType name="NavigationMenuItemGroup">
    <annotation>
     <documentation>
      A group of NavigationMenuItems, collected by property group.
     </documentation>
    </annotation>
    <sequence>
     <element name="NavigationMenuItem" type="cs:NavigationMenuItem"
minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="HasRefineableProperties" type="boolean" />
    <attribute name="Name" type="string" use="required" />
   </complexType>
```

The required attribute HasRefineableProperties specifies whether a group has attributes that could be refined further.

Note: From the perspective of controlling the groups behavior in the front-end application, another attribute may be useful. It is the ExposureControl attribute of type boolean, on the NavigationMenuItem. If set to false (the default), it does not expose refinements contained within NavigationMenuItem. If set to true exposes the collection of refinements.

To request groups:

In the Conversation Web Service request, for each group, specify its name and whether to expose all top-level attributes in the group by specifying the value of Expose attribute on the Refinement¬ GroupConfig element. Optionally, you can also use this attribute on the refinements within the group.

In this example, two groups are requested, FlavorGroup and ProvenanceGroup, but exposing top-level attributes is requested for FlavorGroup only:

```
    <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
      <State/>
     <ContentElementConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="NavigationMenuConfig" MaximumRefinementCount="10"
ReturnFullPath="true" ExposeAllRefinements="false" HandlerFunction="Nav¬
igationMenuHandler" HandlerNamespace="http://www.endeca.com/MDEX/conver¬
sation/handlers/2010" Id="Navigation">
        <RefinementGroupConfig Name="FlavorGroup" Expose="true">
         <RefinementConfig Name="Flavors" Expose="true" MaximumCount="2"/>

        </RefinementGroupConfig>
        <RefinementGroupConfig Name="ProvenanceGroup" Expose="false"/>
      </ContentElementConfig>
    </Request>
```

The Conversation Web Service result includes results for one group, FlavorGroup, for which refinements were requested to be exposed:

```
    <cs:ContentElement xsi:type="cs:NavigationMenu" Id="Navigation">
      <cs:NavigationMenuItemGroup Name="FlavorGroup" HasRefineableProper¬
ties="true">
       <cs:NavigationMenuItem Name="Flavors" DisplayName="Flavors" MultiS¬
elect="And" HasMore="true">
         <cs:ExposureControl Exposed="true">
```

```
            <cs:Operator OwnerId="Navigation" xsi:type="cs:RefinementHide¬
Operator" Name="Flavors" Group="FlavorGroup" Spec="/"/>
          </cs:ExposureControl>
        <cs:Refinement Name="Flavors" Spec="Currant" Label="Currant">
          <cs:Operator xsi:type="cs:RefinementOperator" Name="Flavors"
Spec="Currant"/>
        </cs:Refinement>
        <cs:Refinement Name="Flavors" Spec="Oak" Label="Oak">
          <cs:Operator xsi:type="cs:RefinementOperator" Name="Flavors"
Spec="Oak"/>
        </cs:Refinement>
        <cs:RootDimensionValue DimensionName="Flavors" Spec="/"/>
          <cs:FullPath><!-- path information omitted in this example--
></cs:FullPath>
        </cs:NavigationMenuItem>
         <cs:NavigationMenuItem Name="Drinkability" DisplayName="Drinkabil¬
ity" MultiSelect="None" HasMore="true">
            <cs:ExposureControl Exposed="false">
              <cs:Operator OwnerId="Navigation" xsi:type="cs:RefinementEx¬
poseOperator" Name="Drinkability" Group="FlavorGroup" Spec="/"/>
            </cs:ExposureControl>
          <cs:RootDimensionValue DimensionName="Drinkability" Spec="/"/>
          <cs:FullPath><!-- path information omitted in this example --
></cs:FullPath>
        </cs:NavigationMenuItem>
      </cs:NavigationMenuItemGroup>
     <cs:NavigationMenuItemGroup Name="ProvenanceGroup" HasRefineableProp¬
erties="true"/>
    </cs:ContentElement>
```

Retrieving lists of attribute groups

To retrieve a lists of groups, use a request with `ContentElementConfig` of type `PropertyGrou¬pListConfig`, which uses the `PropertyGroupListHandler`.

To retrieve a list of groups:

Use the request similar to the following example:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State/>
  <ContentElementConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-in¬
stance"
  xsi:type="PropertyGroupListConfig" HandlerFunction="PropertyGrou¬
pListHandler"
  HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
 Id="groups"/>
</Request>
```

The Conversation Web Service request contains a list of groups that are currently defined, specifying a group's display name and a number of attributes within each group:

```
<cs:ContentElement xsi:type="cs:PropertyGroupList" Id="groups">
 <cs:PropertyGroup Key="TypeGroup"
  NumberOfProperties="2" DisplayName="Wine Types and Varieties"/>
 <cs:PropertyGroup Key="FlavorGroup"
  NumberOfProperties="4" DisplayName="Flavors and Drinkability"/>
 <cs:PropertyGroup Key="ProvenanceGroup"
  NumberOfProperties="3" DisplayName="Terroire and Wineries"/>
```

```
 <cs:PropertyGroup Key="RatingsGroup" NumberOfProperties="3"
  DisplayName="Ratings and Evaluations"/>
 <cs:PropertyGroup Key="system-navigation_InternalGroup" NumberOfProper¬
ties="48"/>
</cs:ContentElement>
```

Note: You may notice a group `system-navigation_InternalGroup` which contains all of the attributes that are not members of any other groups. This group is used by the MDEX Engine and is not intended to be used in any other way in your application.

Chapter 12

Using Precedence Rules

This chapter describes how to configure and use precedence rules.

About precedence rules

Precedence rules provide a way to delay the display of Endeca attributes until they offer a useful refinement of the navigation state.

Precedence rules are defined in terms of a trigger attribute and a target attribute, where a user's selection of the trigger reveals the previously unavailable target attribute to the user. That is, precedence rules are triggered by implicit or explicit selections of either managed attribute values or standard attribute values. These triggers are able to cause either managed attributes or standard attributes to be included as available refinements.

Precedence rule **triggers** can be expressed as:

- Managed attribute value (mval): triggered when a particular mval is selected. This can be configured to control whether the mval itself must be selected, or whether any child of the mval will trigger the rule. Using a root mval for a managed attribute effectively causes any selection within that managed attribute to trigger the rule.
- Standard attribute value (sval): triggered when a particular sval is selected.
- Standard attribute: triggered when any value in a particular standard attribute is selected

The precedence rule **target** can be a managed attribute or a standard attribute.

Note that either attribute type can trigger the other type. That is, a managed attribute value configured as a trigger can display a standard attribute, while a standard attribute (or standard attribute value) can be a trigger for a managed attribute target.

To illustrate the concept of precedence rules, assume that one might not want both the Country and State managed attributes to appear simultaneously in a geographical data set. A precedence rule could be defined so that the State managed attribute would appear only after a managed attribute value from the Country managed attribute is selected. This simplifies the user's navigation choices and avoids information overload by hiding the State managed attribute until it is relevant to the navigation state.

Treatment of target attributes associated with multiple precedence rules

A target managed or standard attribute associated with more than one precedence rule is exposed when at least one associated trigger is selected.

For example, assume we have three managed attributes: Author, Region, and Language. We have two precedence rules:

```
Region > Author
Language > Author
```

In this case, the Author managed attribute is displayed after a managed attribute value from either the Region or Author managed attribute is selected.

Precedence rules with non-existent sources

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

Precedence rules versus hierarchical managed attributes

The creation of managed attributes can be facilitated with precedence rules. Consider the task of creating a Geography managed attribute as a hierarchy of country, state, and city. The hierarchy would need to be created manually, with Country as the root managed value. Each country managed value would have its corresponding states as children and each state its corresponding cities. In this scenario, the onus is on the knowledge worker to create and maintain this potentially enormous hierarchy.

Precedence rules offer a much simpler solution. The knowledge worker can produce the same results by creating three individual managed (or standard) attributes (Country, State, and City) and configuring precedence rules such that the State attribute is not presented until a country has been chosen and the City attribute is not presented until a state has been chosen. Because each attribute is flat, this solution involves much less initial and maintenance effort. Clearly, creating a managed attribute hierarchy by hand is a much more difficult task than creating the three flat attributes, configuring precedence rules, and letting contraction do the work to give the application the desired behavior (that is, to mimic the hierarchy).

Precedence rule types

Precedence rules are Standard, Leaf, or Property types.

During configuration, you specify a rule type for the trigger. Managed value triggers are either Standard or Leaf, while standard attribute triggers are Property.

Standard versus Leaf precedence rules

Standard precedence rules display the target attribute if the trigger managed value or its descendants are in the navigation state. Leaf precedence rules display the target attribute only after descendants of the trigger managed value have been selected.

The two types differ in how the trigger dimension value is interpreted:

- For the Standard type, if the managed value specified as the trigger, or any of its descendants, are in the navigation state, then the target attribute is displayed.
- For the Leaf type, only leaf managed values (managed values with no children) that are descendants of the specified trigger managed value cause the target attribute to be displayed. The presence of the specified trigger managed value in the navigation state does not cause the target attribute to appear. Hence, a leaf precedence rule requires that the trigger managed value have children.

Standard rule example

In this Standard rule example, we have a **Color** dimension with a child managed value named **blue**. We can construct a Standard precedence rule with **blue** as the trigger managed value and the managed attribute **ShadesOfBlue** as the target.

When the user drills into **Color** and selects **blue**, the target managed attribute **ShadesOfBlue** is displayed in the user interface.

Leaf rule example

For Leaf type rules, we will use a hierarchical managed attribute named **Country** and a second managed attribute named State. The **Country** dimension looks like this:

```
Country
   - North America
       - Canada
       - Mexico
       - United States
   - Europe
       - England
       - Spain
       - Italy
```

Logically, a user should choose a country before choosing a state. We can use a Leaf precedence rule to suppress the display of the State attribute until a leaf value in the Country managed attribute (an actual country as opposed to a continent) has been selected. To achieve this, a Leaf precedence rule is constructed with the **Country** root managed value as the trigger and the **State** managed attribute as the target.

If the user drills into **Country** and selects an intermediate child managed value (North America or Europe), the target State attribute is not displayed. However, once the user has selected a leaf value from the Country managed attribute (United States, Canada, Mexico, England, Spain, or Italy) the State managed attribute appears.

Precedence rule Property type

Standard attribute triggers are designated as Property types.

Standard attributes can be configured as triggers by using the PROPERTY keyword in the PRECE¬DENCE_RULES index configuration. In other words, when the SRC_PROPERTY keyword is used, then the PROPERTY keyword must also be used, as in this example:

```
<PRECEDENCE_RULES>
  <PRECEDENCE_RULE
    SRC_PROPERTY="Region" SRC_PVAL="Italy" TYPE="PROPERTY"
    DEST_DIMENSION="WineType" DEST_DVAL_SPEC="Chianti" />
</PRECEDENCE_RULES>
```

Note that the use of SRC_PVAL is optional and is used when you want the source to trigger the target only when the specific standard attribute value ("Italy" in this example) is selected.

Configuring precedence rules

You configure precedence rules in the MDEX Engine via the PRECEDENCE_RULES index configuration document.

The PRECEDENCE_RULES document configures precedence rules in the MDEX Engine. The format for configuring triggers and targets is explained in the *MDEX Engine Configuration XML Reference* appendix of the *Latitude Data Integrator Guide*.

To configure precedence rules:

1. In any text editor, edit the PRECEDENCE_RULES index configuration document.
2. Use the Latitude Data Integrator to send the PRECEDENCE_RULES document to the MDEX Engine. For information, see the *Latitude Data Integrator Guide*.

Precedence rules and implicit attribute value selection

When all records in the navigation state are assigned a given attribute value, that attribute value is an implicit selection.

In addition to being selected explicitly by the application, attribute values (either standard attribute values or managed attribute values) can be selected implicitly. For example, if all Champagnes are from France, then the explicit selection of **WineType** > **Champagne** causes the implicit selection of **Region** > **France**. Implicit selection is a function of the set of records in the navigation state, regardless of what combination of search, navigation, and record filters was used to obtain them.

Implicitly-selected attribute values trigger precedence rules in exactly the same way as explicitly-selected attribute values. This behavior helps ensure a consistent user experience, by providing the same attributes for refinement of a given result set, regardless of whether that result set was obtained through search, navigation, or a combination of the two.

For this reason, two navigation paths leading to the same set of records will always have exactly the same set of navigation selections (differing only in whether the selections are implicit or explicit). Because of this equivalence, the set of precedence rules fired in both states will be identical.

When precedence rules are overridden

Implicit selection of a precedence rule's trigger attribute value fires the rule. Under some circumstances, implicit selection of the rule's target managed value also fires it. Specifically, when a precedence rule's target managed value is implicit in the navigation state, and when refinements are available underneath that target managed value, the precedence rule fires and the target attribute is displayed. This occurs even when none of the rule's trigger values have been implicitly or explicitly selected. The MDEX Engine treats any precedence rules targeting the parent managed attributes of these managed values as having fired, even though the rules' trigger values have not been selected.

For this reason, precedence rule target attributes may appear when no precedence rule trigger has been selected.

Part 5

Search Features

Chapter 13

Using Record Search

This section discusses record search, which is an Endeca equivalent of full-text search, and is one of the fundamental building blocks of Endeca search capabilities.

Record search overview

Record search allows a user to perform a keyword search against specific attribute values assigned to records.

The resulting records that have matching attribute values are returned, along with any valid refinement values.

Unlike value search, record search returns a complete `Navigation` object, the same object that is returned when a user filters records by selecting a managed attribute value.

Because record search returns a navigation page, it is important to remember that the record search parameter acts as a record filter in the same way that an attribute value does, even though it is not a specific value.

Example of record search

For example, consider the following records:

Rec ID	Attribute value (WineType)	Name of attribute	Description of attribute
1	Red (Dim Value 101)	Antinori Toscana Solaia	Dark ruby in color, with extremely ripe...
2	Red (Dim Value 101)	Chateau St. Jean	Dense, rich, and complex...
3	White (Dim Value 103)	Chateau Laville	Dense and vegetal, with celery, pear, and spice flavors...
4	Other (Dim Value 103)	Jose Maria da Fonseca	Big, ripe, and generous, layered with honey...

When the user performs a record search on the Description attribute using the keyword `dense`, the following objects are returned:

- 2 records (records 2 and 3)
- 2 refinement attribute values (Red and White)

When performing a record search on the Description attribute using the keyword `ripe`, these objects are returned:

- 2 records (records 1 and 4)
- 2 refinement attribute values (Red and Other)

Note: In addition to basic record search, other features affect the behavior of record search, such as spelling support, relevance ranking of results, wildcard syntax, multiple attribute record searches, and attribute group record searches. These are discussed in detail in their respective sections.

Features for controlling record search

The following statements describe various aspects of record search behavior and how you can control it:

- To configure run-time record search behavior, you must create one or more search interfaces. For more information, see the chapter on search interfaces.
- There are no MDEX Engine configuration flags necessary to enable record searching. If an attribute was properly enabled for record searching, it will automatically be available for record searching.
- Multiple MDEX Engine configuration flags are available to manage different controls for record search, such as spelling support and relevance ranking. See specific feature sections for details.

Configuring attributes for record search

The first step in implementing basic record search is to configure a standard attribute for record searching using the Latitude Data Integrator.

The `mdex-property_IsTextSearchable` attribute of a PDR enables the attribute for record searching. The valid settings for this attribute are:

- If set to `true`, the attribute is enabled for record search.
- If set to `false`, the attribute is not enabled for record search. This is the default.

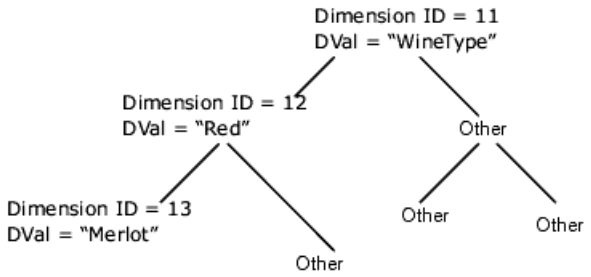
To configure an attribute for record search, see the *Latitude Data Integrator Guide*.

Enabling hierarchical record search

If you want to consider ancestor managed attribute values when matching a record search query, you can enable hierarchical record search.

By default, a record search that uses a managed attribute as the search key returns only those records that are assigned an attribute value whose text matches the search terms. As part of this behavior, record search does not consider implicit ancestor attribute values.

For example, consider the following managed attributes hierarchy:



In this hierarchy, the `Red` attribute (with an ID of 12) is an ancestor of the `Merlot` attribute (ID of 13). A search against the `WineType` attribute for the keyword `merlot` matches any records assigned the attribute value 13. But a search in `WineType` for `red merlot` does not match these records, because record search does not normally consider implicit ancestor attribute value assignments.

In such cases, you may want record search to consider ancestor attribute values when matching a record search query. You can enable this sort of hierarchical record search by setting the `mdex-dimension_IsRecordSearchHierarchical` attribute to `true` in the managed attribute's DDR (Dimension Description Record).

Adding search synonyms to attribute values

You can add synonyms to a managed attribute value so that users can search for other text strings and still get the same records as a search for the original attribute value name.

When a managed attribute is used as the record search key, the text strings considered by record search for matching are the individual names of the attribute values within the attribute. The managed attribute name is automatically added as a searchable string.

You can add synonyms to an attribute value so that users can search for other text strings and still get the same records as a search for the original attribute value name. Synonyms can be added only to child attribute values, not to root attribute values.

You can use the Data Ingest Web Service's `ingestDimensionValues` operation to add synonyms when adding attribute values to the MDEX Engine. For details, see the *Latitude Data Ingest API Guide*.

Implementing record search in Latitude Studio

Record search queries in a Latitude Studio application are made from the **Search Box** component.

To make record search queries in Latitude Studio, you must add and configure the **Search Box** component. For details on this component, see the *Latitude Studio Power User's Guide*.

Implementing record search with the API

This section describes how to issue record search queries using the Conversation Web Service API.

For more information on the Conversation Web Service interface, see the *MDEX Engine API Reference*.

Related Links

Obtaining the available search keys on page 130

The `AvailableSearchKeys` component lets you retrieve a list of the searchable attributes and search interfaces available in the MDEX Engine.

Record search operator on page 131
A basic record search requires a `SearchOperator` with a `SearchFilter` type.

Obtaining the available search keys

The `AvailableSearchKeys` component lets you retrieve a list of the searchable attributes and search interfaces available in the MDEX Engine.

The `AvailableSearchKeys` element contains one or more `AvailableSearchKey` elements. The complex type `AvailableSearchKey` in the Conversation Web Service identifies the items that are searchable — search interfaces and searchable attributes. This type has the following format:

```
<complexType name="AvailableSearchKey">
    <annotation>
     <documentation>
      A key used to identify searchable properties and search interfaces.

     </documentation>
    </annotation>
    <sequence>
     <element name="Key" type="string" use="required"/>
     <element name="DisplayName" type="string" use="required"/>
    </sequence>
    <attribute name="Interface" type="boolean" use="required" />
   </complexType>
```

The `Interface` attribute distinguishes whether the search key is a searchable attribute or a search interface. If the search key is a search interface, the attribute is set to `true`. If the search key is not a search interface and is a searchable attribute, the attribute is set to `false`.

Request for available search keys

To make a request for available search keys, use the `AvailableSearchKeysConfig` component as illustrated in this example:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
   <State/>
   <ContentElementConfig xsi:type="AvailableSearchKeysConfig"
      HandlerFunction="AvailableSearchKeysHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
     Id="MySearchKeys" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 />
</Request>
```

The `Id` attribute is an identifier for the configuration.

Response for available search keys

The response contains an `AvailableSearchKeys` component that lists all of the searchable keys in a single alphabetically ordered list, as shown in this example:

```
dd
<cs:Results xmlns:cs="http://www.endeca.com/MDEX/conversation/2010"
     xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09">
   <cs:Request>
      <State xmlns="http://www.endeca.com/MDEX/conversation/2010"
```

```
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"/>
      <ContentElementConfig xsi:type="AvailableSearchKeysConfig"
        HandlerFunction="AvailableSearchKeysHandler"
        HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
        Id="MySearchKeys" xmlns="http://www.endeca.com/MDEX/conversa¬
tion/2010"
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
   </cs:Request>
   <cs:ContentElement xsi:type="cs:AvailableSearchKeys"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <cs:AvailableSearchKey Interface="true">
         <cs:Key>AllWineSearch</cs:Key>
         <cs:DisplayName>AllWineSearch</cs:DisplayName>
      </cs:AvailableSearchKey>
      <cs:AvailableSearchKey Interface="false">
         <cs:Key>Description</cs:Key>
         <cs:DisplayName>Wine Description</cs:DisplayName>
      </cs:AvailableSearchKey>
      <cs:AvailableSearchKey Interface="false">
         <cs:Key>WineType</cs:Key>
         <cs:DisplayName>Wine Type</cs:DisplayName>
      </cs:AvailableSearchKey>
   </cs:ContentElement>
</cs:Results>
```

Each `AvailableSearchKey` element lists the name of a searchable attribute or search interface (the `Key` sub-element) and the display name (which can have a non-NCName format). If the search key is a search interface, the `Interface` attribute is set to `true`. In this sample response, one search interface, `AllWineSearch`, and two attributes, `Description` and `WineType`, are listed as available search keys.

Record search operator

A basic record search requires a `SearchOperator` with a `SearchFilter` type.

The syntax for a search request is shown in this example:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="SearchOperator" Within="false">
   <SearchFilter Mode="AllPartial" RelevanceRankingStrategy="numfields"
      EnableSnippeting="true" SnippetLength="10"
      Key="Description">spice flavors</SearchFilter>
</Operator>
```

The text content of the `SearchFilter` element contains the search term(s). In the example, a record search is being made for the "spice flavors" keywords. The meanings of the `SearchOperator` and `SearchFilter` attributes are as follows:

Search attribute	Meaning
`Within`	If set to `false`, then the visible parts of the filter state are cleared first.
`Key`	Specifies which attribute will be evaluated when searching. You specify an attribute as a value for this parameter. (You can also specify a search interface as a value.)
`Mode`	Optionally specifies a match mode.

Search attribute	Meaning
`RelevanceRankingStrat¬egy`	Optionally specifies a relevance ranking strategy.
`EnableSnippeting`	Optionally enables snippeting for the query.
`SnippetLength`	Optionally specifies the maximum number of words a snippet can contain.

Search query processing order

This section summarizes how the MDEX Engine processes record search queries.

While this summary is not exhaustive, it covers the processing steps likely to occur is most application contexts. The process outlined here assumes that other features (such as spelling correction and thesaurus) are being used.

The MDEX Engine uses the following high-level steps to process record search queries:

1. Record filtering
2. Endeca Query Language (EQL) filtering
3. Tokenization
4. Auto correction (spelling correction and automatic phrasing)
5. Thesaurus expansion
6. Stemming
7. Primitive term and phrase lookup
8. Did you mean
9. Navigation filtering
10. Analytics
11. Relevance ranking

Note: For Boolean search queries, tokenization, auto correction, and thesaurus expansion are replaced with a separate parsing phase.

Related Links

Step 1: Record filtering on page 133
If a record filter is specified, whether for security, custom catalogs, or any other reason, the MDEX Engine applies it before any search processing.

Step 2: Endeca Query Language filters on page 133
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. If used, this feature is applied after record filtering.

Step 3: Tokenization on page 133
Tokenization is the process by which the MDEX Engine analyzes the search query string, yielding a sequence of distinct query terms.

Step 4: Auto correction (spelling correction and automatic phrasing) on page 134
If spelling correction and automatic phrasing are enabled and triggered, the MDEX Engine implements them as part of the record search processing.

Step 5: Thesaurus expansion on page 134

The tokenized query, as well as each query alternative generated by spelling suggestion, is expanded by the MDEX Engine based on thesaurus matches. This topic describes the behavior of the thesaurus expansion feature.

Step 6: Stemming on page 135
Query terms, unless they are delimited with quotation marks to be treated as exact phrases, are expanded by the MDEX Engine using stemming.

Step 7: Primitive term and phrase lookup on page 135
Primitive term and phrase lookup is the lowest level of search processing performed by the MDEX Engine.

Step 8: Did You Mean on page 135
The MDEX Engine performs the "Did you mean" processing as part of the record search processing.

Step 9: Navigation filtering on page 135
The MDEX Engine performs all filtering based on the navigation state after the search processing. This order is important, because it ensures that the spelling suggestions remain consistent as the navigation state changes.

Step 10: Analytics on page 135
Endeca Analytics builds on the core capabilities of the MDEX Engine to enable applications that examine aggregate information such as trends, statistics, analytical visualizations, comparisons, and so on, all within the Guided Navigation interface. If Analytics is used, it is applied near the end of processing.

Step 11: Relevance ranking on page 136
Relevance ranking is the last step in the MDEX Engine processing for the record search. Each of the navigation-filtered search results is assigned a relevance score, and the results are sorted in descending order of relevance.

Step 1: Record filtering

If a record filter is specified, whether for security, custom catalogs, or any other reason, the MDEX Engine applies it 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.

Step 2: Endeca Query Language filters

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. If used, this feature is applied after record filtering.

For details on this feature, see the Analytics chapter of the *Latitude Studio Power User's Guide*.

Step 3: Tokenization

Tokenization is the process by which the MDEX Engine analyzes the search query string, yielding a sequence of distinct query terms.

Step 4: Auto correction (spelling correction and automatic phrasing)

If spelling correction and automatic phrasing are enabled and triggered, the MDEX Engine implements them as part of the record search processing.

If the spelling correction feature is enabled and triggered, the MDEX Engine creates spelling suggestions by enumerating (for each query term) a set of alternatives, and considering some of the combinations of term alternatives as whole-query alternatives.

Each of these whole-query alternatives is subject to thesaurus expansion and stemming.

For example, if the tokenized query is `employee moral`, then `employee` may generate the set of alternatives {`employer`, `employee`, `employed`}, while `moral` may generate the set of alternatives {`moral`, `morale`}.

The two query alternatives generated as spelling suggestions might be `employer moral` and `employee morale`.

For details on the auto-correction feature, see the section about it.

If automatic phrasing is enabled, then the MDEX Engine automatically combines distinct query terms that match a phrase in the phrase dictionary into a search phrase.

Once distinct terms are grouped as an automatic phrase, the phrase is not subject to additional thesaurus expansion and stemming.

For example, suppose the phrase dictionary contains two phrases `Kenneth Cole` and also `blue jeans`. If the query is `Kenneth Cole blue jeans`, the alternative query might be `“Kenneth Cole” “blue jeans”`.

For details on automatic phrasing, see the section about it.

Step 5: Thesaurus expansion

The tokenized query, as well as each query alternative generated by spelling suggestion, is expanded by the MDEX Engine based on thesaurus matches. This topic describes the behavior of the thesaurus expansion feature.

Thesaurus expansion replaces each expanded query term with an `OR` of alternatives.

For example, if the thesaurus expands `pentium` to `intel` and `laptop` to `notebook`, then the query `pentium laptop` will be expanded to:

```
(pentium OR intel) AND (laptop OR notebook)
```

assuming the match mode is `MatchAll`.

The other match modes (with the exception of MatchBoolean) behave analogously.

If there is a multiple-word thesaurus match, then `OR` is used on the query itself to accommodate the various ways of partitioning the query terms.

For example, if `high speed` expands to `performance`, then the query `high speed laptop` will be expanded to:

```
(high AND speed AND (laptop OR notebook)) OR (performance
AND (laptop OR notebook))
```

Multiple-word thesaurus matches only apply when the words appear in exact sequence in the query. The queries `speed high laptop` and `high laptop speed` do not activate the expansion to `performance`.

For more details on thesaurus expansion, see the section on this feature.

Step 6: Stemming

Query terms, unless they are delimited with quotation marks to be treated as exact phrases, are expanded by the MDEX Engine using stemming.

The expansion for stemming applies even to terms that are the result of thesaurus expansion. A stemmed query term is an `OR` expression of its word forms.

For example, if the query `pentium laptop` was thesaurus-expanded to:

```
(pentium OR intel) AND (laptop OR notebook)
```

it will be stemmed to:

```
(pentium OR intel) AND (laptop OR laptops OR notebook
OR notebooks)
```

assuming that only the improper nouns have plurals in the word form dictionary.

For more details on stemming, see the section on this feature.

Step 7: Primitive term and phrase lookup

Primitive term and phrase lookup is the lowest level of search processing performed by the MDEX Engine.

The MDEX Engine evaluates each search term as is, and matches it to the set of documents containing that precise word or phrase (given the tokenization rules) in the indexes being searched. Search is never case-sensitive, even for phrases.

Step 8: Did You Mean

The MDEX Engine performs the "Did you mean" processing as part of the record search processing.

“Did you mean?” processing is analogous to the spelling correction and automatic phrasing processing, only that the results are not included, but rather the spelling suggestions and automatic phrases themselves are returned.

For details on the “Did you mean?” feature, see the section about it.

Step 9: Navigation filtering

The MDEX Engine performs all filtering based on the navigation state after the search processing. This order is important, because it ensures that the spelling suggestions remain consistent as the navigation state changes.

Step 10: Analytics

Endeca Analytics builds on the core capabilities of the MDEX Engine to enable applications that examine aggregate information such as trends, statistics, analytical visualizations, comparisons, and so on, all within the Guided Navigation interface. If Analytics is used, it is applied near the end of processing.

For more information about this feature, see the *Latitude Power User's Guide*.

Step 11: Relevance ranking

Relevance ranking is the last step in the MDEX Engine processing for the record search. Each of the navigation-filtered search results is assigned a relevance score, and the results are sorted in descending order of relevance.

For details on this feature, see the section about it.

Tips for troubleshooting record search

This topic includes tips for troubleshooting record search.

Due to the user-specified interaction of this feature (as opposed to the system-controlled interaction of Guided Navigation in which the MDEX Engine controls the refinement values presented to the user), a user is allowed to submit a keyword search that does not match any records. Therefore, it is possible for a user to make a dead-end request with zero results when using record search. Applications utilizing record search need to account for this.

In production systems, these Endeca attributes are typically hard-coded at the application level, because the application requires specific search keys to be used for specific functionality.

If an Endeca attribute is not enabled for record searching but an application attempts to perform a record search against this attribute, the MDEX Engine successfully returns a null result set. The MDEX Engine error log, however, outputs the following message: `In fulltext search: [Wed Sep 3 12:28:02 2010] [Warning] Invalid fulltext search key "Description" requested.`

The `-v` flag to the MDEX Engine causes the MDEX Engine to output detailed information about its record search configuration. If you are unsure whether the MDEX Engine is recognizing a particular parameter, start it with the `-v` flag and check the output.

Finally, while implementing record search by enabling record standard attributes for searching is the normal approach, managed attribute values can also be enabled for record searching. The managed attribute name then replaces the standard attribute key as the value for the `Search` operator in the MDEX Engine query. The resulting navigation request contains any record that is tagged with a managed attribute value from the specified managed attribute that matches the search terms.

Performance impact of record search

Because record searching is an indexed feature, each attribute enabled for record searching increases the size of the MDEX Engine process.

The specific size of the increase is related to the size of the unique word list generated by the specific attribute in the data set. Therefore, only attributes that are specifically needed by an application for record searching should be configured as such.

Chapter 14

Working with Search Interfaces

A search interface is a named collection of standard and managed attributes, each of which is enabled for record search.

About search interfaces

A search interface allows you to control record search behavior for groups of one or more attributes.

A search interface may also contain:

- A number or attributes, such as name, cross-field information, and so on.
- An ordered collection of one or more ranking strategies.

Some of the features that can be specified for a search interface include:

- Relevance ranking
- Matching across multiple attributes
- Keyword in context results
- Partial match

You can use a search interface to control the behavior of search against a single standard or managed attribute, or to simultaneously search across multiple attributes.

For example, if a data set contains both an `Actor` standard attribute and `Director` managed attribute, a search interface can provide the user the ability to search for a person's name in both. A search interface's name is used just like a normal attribute when performing record searches. By default, a record search query on a search interface returns results that match any of the attributes in the interface.

Implementing search interfaces

You implement search interfaces with the Latitude Data Integrator.

Before implementing search interfaces, make sure that all the attributes that are going to be included in a search interface have already been enabled for record search. In addition, if the search interface will include a relevance ranking strategy, make sure that the relevance ranking strategy has been configured.

If you are implementing wildcard search in a search interface, search interfaces can contain a mixture of wildcard-enabled and non-wildcard-enabled members (although only the former will return wildcard-expanded results).

You implement a search interface via the RECSEARCH_CONFIG configuration element. The resulting search interface should look similar to this example of a search interface named **AllWine** that uses a relevance ranking strategy named **All**:

```
<RECSEARCH_CONFIG xmlns:ns="http://www.endeca.com/MDEX/XQuery/2009/09"
   xmlns:mdex="http://www.endeca.com/MDEX/XQuery/2009/09"
   xmlns:config-service="http://www.endeca.com/MDEX/config/services/con¬
fig/2010"
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <SEARCH_INTERFACE CROSS_FIELD_BOUNDARY="NEVER" CROSS_FIELD_RELE¬
VANCE_RANK="0"
    DEFAULT_RELRANK_STRATEGY="All" NAME="AllWine">
    <MEMBER_NAME RELEVANCE_RANK="4">WineType</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="3">WineName</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="2">Winery</MEMBER_NAME>
    <MEMBER_NAME RELEVANCE_RANK="1">Description</MEMBER_NAME>
  </SEARCH_INTERFACE>
</RECSEARCH_CONFIG>
```

For information on how to configure a search interface, see the *Latitude Data Integrator Guide*.

Options for allowing cross-field matches

The CROSS_FIELD_BOUNDARY attribute specifies when the MDEX Engine should try to match search queries across attribute boundaries.

The three settings for CROSS_FIELD_BOUNDARY are:

Setting	Description
Always	The MDEX Engine always looks for matches across attribute boundaries, in addition to matches within an attribute. If you choose to use cross-field matching, the **Always** setting is recommended and is the default. For example, in the `Sony camera` user query, if **CROSS_FIELD_BOUNDARY** is set to **Always**, the MDEX Engine returns all matches with `Brand = Sony` and `Product_Type = camera`.
Never	The MDEX Engine does not look across boundaries for matches.
On Failure	The MDEX Engine only tries to match queries across attribute boundaries if it fails to find any matches within a single attribute. Note that in most cases, the **Always** setting provides better results than the **On Failure** setting.

By default, record search queries using a search interface return the union of the results from the same record search query performed against each of the interface members.

For example, assume a search interface named `MoviePeople` that includes `actor` and `director` attributes. Searching for `deniro` against this interface returns the union of records that results from searching for `deniro` against the `actor` attribute and against the `director` attribute.

Less frequently, you may wish to allow a match to span multiple attributes. For example, in the same `MoviePeople` search interface, a query for `clint eastwood` returns records where either an `actor`

standard attribute or a `director` attribute is assigned a value containing the words `clint` and `eastwood`. This behavior is useful for this query, where the search terms all relate to a single concept (the actor/director Clint Eastwood).

However, in some cases returning a union of the results from the same record search query performed against each search interface member is unnecessarily limiting. For example, in a home electronics catalog application, a customer searching for `Sony camera` might be interested in a broad range of products, but this record search would only return the few products that have the terms `Sony` and **camera** in the product name.

In such cases, you can use the `CROSS_FIELD_BOUNDARY` attribute when you create a search interface. This attribute specifies when the MDEX Engine should try to match search queries across attribute boundaries, but within the members of the search interface.

How cross-field matches work in multi-assign cases

When a search interface member (that is, a searchable attribute) is multi-assigned on a record, the multi-assigns are treated by the MDEX Engine as separate matches, just as if they were values from different attributes. A search that matches two or more terms in separate multi-assign values for the same attribute is treated as a cross-field match by the MDEX Engine.

For example, assume a record has the following attribute values:

```
P_Tag: Tom Brady
P_Tag: Jersey
```

A search against P_Tag for "tom brady jersey" is treated as a cross-field match, even though all results were found in the same attribute (P_Tag).

Additional search interface options

You can configure other features for the search interface by specifying other match-related attributes to the `SEARCH_INTERFACE` element.

The following table lists the attributes (other than the `CROSS_FIELD_BOUNDARY` attribute) that you can specify with the `SEARCH_INTERFACE` element.

Attribute	Purpose
`DEFAULT_RELRANK_STRATEGY`	For record search, assigns a default relevance scoring function to a search interface.
`CROSS_FIELD_RELEVANCE_RANK`	Specifies the relevance rank score for cross-field matches. The value should be an unsigned 32-bit integer. The default value for `CROSS_FIELD_RELEVANCE_RANK` is 0.
`STRICT_PHRASE_MATCH`	Specifies that the MDEX Engine should interpret a query strictly when comparing white space in the query with punctuation in the source text. If set to FALSE, partial word tokens connected in the source text by punctuation can be matched to a phrase query where the partial tokens are separated by spaces instead of matching punctuation. The default value of this attribute is TRUE.

You can also use the `PARTIAL_MATCH` element to specify if partial query matches should be supported for the `SEARCH_INTERFACE` that contains this element.

Search interfaces in queries

Use the name of the search interface in the `Search` element of a Conversation Web Service query.

The search interface can be specified in a query by specifying the name of the interface in the `Key` attribute. The following example uses the `AllProps` search interface:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
         <FilterState/>
         <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchOperator" Within="false">
           <Search Mode="AllPartial" Key="AllProps">merlot</Search>
         </Operator>
         <ContentElementConfig xsi:type="RecordListConfig" HandlerFunc¬
tion="RecordListHandler"
            HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
            Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
            <RecordsPerPage>5</RecordsPerPage>
         </ContentElementConfig>
      </Request>
   </soap:Body>
</soap:Envelope>
```

By default, using a search interface in a search performs a logical `OR` on the attributes in the interface. For example, if a data set contains both an `Actor` standard attribute and `Director` managed attribute, a search interface can provide the user the ability to search for a person's name in both.

Tips for troubleshooting search interfaces

All the tips for troubleshooting basic record search are also useful for troubleshooting record search that uses search interfaces. To get the most out of the search interfaces feature, make sure to set your search interfaces to contain the relevant searchable fields.

Chapter 15

Using Value Search

This section discusses how the MDEX Engine performs value search and how to configure it for your application.

About value search

Value search allows users to perform keyword searches across attributes for values with matching names.

End users of applications powered by Endeca can search all types of attribute values, including values for standard and managed attributes. The front-end application can present these values to the end-user, allowing the user to select them and create a new navigation request.

Value search is enabled differently for attributes:

- Standard attributes. You can make standard attributes of type string value searchable. To configure a set of standard attributes of type string whose values will be indexed for search, modify the values of the `IsPropertyValueSearchable` attribute on the PDRs.
- Managed attributes. All managed attributes are indexed for value search by default, and you cannot disable value search for them.

To request value search on any standard attributes that are value searchable, use the Conversation Web Service complex type `ValueSearchConfig`. With this type, you can:

- Limit the scope of searches to those values that belong to particular standard attributes using the `RestrictToProperties` element.
- Limit the number of values to return per attribute with the `MaxPerProperty` attribute.

In the Conversation Web Service response, the result of a value search is returned in a `ValueSearch` type that contains attribute values. The attribute values are organized by attribute and returned as a full path of hierarchical refinements.

How value search works

Value search returns single values that match the user's search terms, organized by attribute.

To be considered a valid result, a value must match all of the search terms that the user provides in the request to the MDEX Engine.

Example of value search

For example, a value search for `red` might return:

Attribute	Values
`WineType`	`Red`
`Wineries`	`Green & Red, Red Hill, Red Rocks`
`Drinkability`	`Drink with red meat`

When to use value and record search

Value search is sometimes confused with record search. This topic provides examples of when to use each type of search.

Understanding the differences between the two basic types of keyword search (record search and value search) is important before creating a solution for a specific business problem. Use the following recommendations:

Type of keyword search	When to use
Value search	In general, data sets with little descriptive text and extensive attribute values of type string that represent the most frequently searched terms (for example, `autos`) are a good fit for value search. Keyword searches are usually suitable for such keywords as `make`, `model`, or `year`. These keywords are also likely candidates for being configured as managed attributes in your application.
Record search	Data sets with descriptive text or names (such as news articles) are better suited for record search. This is because a reasonable set of attribute values for such a data set cannot be expected to cover all the terms required to handle keyword search. In such cases, text search allows an application to search directly against record text (such as the body of an article).

For many applications, a combination of value search and record search is the best solution. In this case, separate value search and text search queries are executed simultaneously for the same keywords:

- If a value matches, the user is given the opportunity to select that value in place of the record search query to produce results.
- If no values match, the user is still left with the matching records for a record search query.

Keep in mind that navigation queries and value search queries are completely independent. In the scenario described above where both queries are executed simultaneously, neither query affects the other. Record search is a variation of a navigation query. Record search could return results even though value search does not, and vice-versa.

Enabling value search

You enable a standard attribute for value search by changing the values in the `mdex-property-IsPropertyValueSearchable` attribute in the PDR, using the Latitude Data Integrator.

Managed attributes are always enabled for value search in the MDEX Engine. In addition, you can also enable standard attributes of type string for value search. In this case, these attributes are indexed and searched by the MDEX Engine. Only the standard attributes of type string can be enabled for value search.

The `mdex-property-IsPropertyValueSearchable` attribute in the PDR specifies whether an attribute in your data set is value searchable. The valid settings for this attribute are:

- `true` means that the attribute is enabled for value search. This is the default.
- `false` means that the attribute is not enabled for value search.

If, in addition to enabling value search for specific attributes of type string, you also would like to enable wildcard search for all value search queries, set the `mdex-config_EnableValueSearchWildcard` attribute in the Global Configuration Record (GCR) to `true`.

For information on how to enable a standard attribute for value search, see the *Latitude Data Integrator Guide*.

Related Links

Performance impact of value search on page 144
This topic discusses value search and its impact on MDEX Engine performance.

Utilizing value search in Latitude Studio

Value search supports the refinement search available in the **Guided Navigation** component, and the ability to utilize typeahead search in the **Search Box** component.

For additional information on configuring Latitude Studio components that utilize value search, see the *Latitude Studio Power User's Guide.*

Interaction of value search and wildcard search

By default, value search allows wildcards at the end of the search term (such as `gua*` for the search term `guarantee`). To enable wildcards elsewhere in a search term, you need to set the `mdex-config_EnableValueSearchWildcard` attribute in the Global Configuration Record (GCR) to `true`, for the standard attribute in your records.

The following examples illustrate how the MDEX Engine treats wildcards in value searches:

- A wildcard search at the end of the search term, such as `gua*` is conducted by the MDEX Engine for all standard attributes for which value search is enabled.
- Wildcard searches of type `*uara` and `g*ara` are conducted by the MDEX Engine only if the GCR attribute `mdex-config_EnableValueSearchWildcard` is set to `true` for the corresponding standard attribute on your records. The default value for this attribute is `false`, meaning that wildcard search is disabled for value search.

Performance impact of value search

This topic discusses value search and its impact on MDEX Engine performance.

Limit value search scope and the number of returned results

If you submit a value search query, the query is generally very fast. The runtime performance of value search directly corresponds to the number of values and the size of the resulting set of matching values. In general, this feature performs at a much higher number of operations per second than navigation requests. The most common performance problem is when the resulting set of values is exceptionally large (greater than 1,000), thus creating a large results page. To avoid it, limit the number of results per request, using value search parameters.

The query will be faster if you limit the scope and the number or results returned:

- To limit the scope of value search, use the `RestrictToProperties` element to specify the specific standard attributes for which you expect matches returned by the MDEX Engine.
- To limit the number of results returned per each value-searchable attribute, use the `MaxPerProperty` attribute in your request.

Decide which attributes to make value searchable

All managed attributes are always value searchable (you cannot toggle the value search setting for them). In addition, standard attributes of type string can be made value searchable. The `IsProperty¬ValueSearchable` attribute on the PDR controls whether the attribute in your record set is enabled for value search.

Before changing a value search setting for an attribute, examine your data to decide which of the attributes in your record set need to be value searchable. Next, turn off value search for attributes you will not be using for navigation, such as those standard attributes that contain long chunks of text.

Implementing value search with the API

This section provides examples of Conversation Web Service requests and responses and describes parameters you can use for value search.

Related Links

Value search query format on page 145
A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

Constructing a value search query on page 146
You create a value search query by issuing a request that uses the `ValueSearchConfig` type.

Restricting value search to specific attributes on page 148
Value search queries could potentially contain many results. You can use `RestictToProperties` attribute to limit the number of returned results to a list of one or more specified attributes.

Limiting the number of results per attribute on page 148
Another way to limit value search results is to specify the number of values to return for each attribute, using the `MaxPerProperty` attribute.

Retrieving the number of matching results on page 149
The standard response to any value search request always includes information about the total number of matched values found, and whether all of them have been returned in this request. This information is returned in the `TotalValuesCount` and `HasMore` attributes on the `PropertyMatches` element.

Ordering results on page 150
Value search results consist of values grouped by attribute. Attributes in the result list are returned in an ascending alphabetical order.

Specifying relevance ranking strategy for results on page 150
To rank the order of results received in response for a value search request, you can use the `RelevanceRankingStrategy` attribute.

Value search query format

A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

The `ValueSearchConfig` type controls the behavior of a single value search configuration. This type has the following parameters (some of which are optional):

Parameter	Description
`Id`	An identifier for this query configuration. Optional.
`Mode`	Specifies a search mode, such as `Any`, or `AllPartial`. Optional. If `Mode` is not used, the query defaults to using the `All` search mode.
`MaxPerProperty`	Limits the number of matches returned per attribute. Optional. If this attribute is omitted, all found matches for the attribute are returned.
`RelevanceRankingStrategy`	Specifies a relevance ranking strategy to use on the results. Optional. If this attribute is omitted and you do not specify a relevance ranking strategy, the MDEX Engine uses the value for the strategy provided in the `DIMSEARCH_CONFIG` XML configuration document. Further, if the document does not specify a strategy, the MDEX Engine ranks the results using the following three strategies in this order (to break ties): `interp`, `exact` and `static("nbins",descending)`.
`SearchTerm`	Contains the search term(s) (also known as keywords) the MDEX Engine uses to conduct value search. Each entry can be plus- or space-delimited. Required.
`RestrictToProperties`	Specifies one or more standard attribute names within which to conduct value search. Optional. If not specified, matches for all attributes are returned which may affect performance. If a managed attribute is searched, using this attribute you can search within a whole attribute and its entire hierarchy of values,

Parameter	Description
	but you cannot restrict value search to a subtree within a particular root value in the hierarchy.

Example

The following example illustrates the format of a typical value search request in the Conversation Web Service. In this request, a search is conducted for terms 1, 2, 3 within the `WineRating` attribute. The number of requested results to return per attribute is set to 5:

```
<ContentElementConfig
 xsi:type="ValueSearchConfig"
 HandlerFunction="ValueSearchHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
 Id="ValueSearch" Mode="Any"
 MaxPerProperty="5"
 RelevanceRankingStrategy="static(nbins,descending)"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SearchTerm>1 2 3</SearchTerm>
  <RestrictToProperties>
    <Property>WineRating</Property>
  <RestrictToProperties>
</ContentElementConfig>
```

Constructing a value search query

You create a value search query by issuing a request that uses the `ValueSearchConfig` type.

Use the parameters for `ValueSearchConfig` specified in its format.

As a rule of thumb, for any attribute that could contain more than 100 possible results, use `<RestrictToProperties>` and `MaxPerProperty` attributes to help control the results returned from the MDEX Engine. Without these controls, the size of the resulting response from the Conversation Web Service could cause slow response times between your front-end application and the MDEX Engine.

To create a value search query:

Create a Conversation Web Service request, similar to the following example:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
  <ContentElementConfig HandlerFunction="ValueSearchHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

    Id="ValueSearch" MaxPerProperty="3" Mode="All"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Value¬
SearchConfig">
    <SearchTerm>Mer*</SearchTerm>
    <RestrictToProperties>
      <Property>WineType</Property>
      <Property>Name</Property>
    </RestrictToProperties>
```

```
  </ContentElementConfig>
</Request>
```

In this query, a search term, `Mer*`, uses wildcards. This implies that the attributes `WineType` and `Name` are enabled for wildcard search during value searches.

This query specifies two attributes that will be searched — `WineType` and `Name`; it also limits to 3 the number of values that are requested to be returned from the MDEX Engine, with the `MaxPer¬Property` attribute.

The response may look like the following example:

```
<ContentElement xsi:type="ValueSearch" Id="ValueSearch"
xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <PropertyMatches Name="WineType" DisplayName="Wine Type" TotalVal¬
uesCount="1" HasMore="false">
    <Match>
      <MatchingValue>/Red/Merlot</MatchingValue>
      <FullPath>
        <DimensionValue>
         <DimensionValue DimensionName="WineType" Spec="/">WineType</Di¬
mensionValue>
          <Operator xsi:type="RefinementOperator" Name="Decimal" Spec="/"/>

        </DimensionValue>
        <DimensionValue>
          <DimensionValue DimensionName="WineType" Spec="/Red">Red</Di¬
mensionValue>
          <Operator xsi:type="RefinementOperator" Name="Decimal"
Spec="/Red"/>
        </DimensionValue>
        <DimensionValue>
         <DimensionValue DimensionName="WineType" Spec="/Red/Merlot">Mer¬
lot</DimensionValue>
          <Operator xsi:type="RefinementOperator" Name="Decimal"
Spec="/Red/Merlot"/>
        </DimensionValue>
      </FullPath>
    </Match>
  </PropertyMatches>
  <PropertyMatches Name="Name" DisplayName="Wine Name" TotalValuesCount="4"
 HasMore="true">
    <Match>
      <MatchingValue>Merlot Napa Valley Limited Reserve</MatchingValue>
    </Match>
    <Match>
      <MatchingValue>Biltmore Reserve Merlot</MatchingValue>
    </Match>
    <Match>
      <MatchingValue>Whitehall Lane Merlot</MatchingValue>
    </Match>
  </PropertyMatches>
</ContentElement>
```

In this response, the following information is returned:

- `FullPath` specifies the full path of refinements, for returned results.
- `TotalValuesCount` specifies the number of values returned for each value-searchable attribute.

- `HasMore` specifies whether there exist more attribute matches, beyond those that are returned. Because the request limited the number of result values to 3, the list of results returned for the `Name` attribute contains three values and also indicate that an additional matching value exists that is not returned: `TotalValuesCount="4"` and `HasMore="true"`

Related Links

Value search query format on page 145
A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

Restricting value search to specific attributes

Value search queries could potentially contain many results. You can use `RestictToProperties` attribute to limit the number of returned results to a list of one or more specified attributes.

If an managed attribute is searched, using the `RestictToProperties` attribute you can search within a whole managed attribute and its entire hierarchy of values, but you cannot restrict value search to a subtree within a particular root value in the hierarchy.

To restrict value search to searching specific attributes:

Use the `RestictToProperties` attribute to specify a list of attributes for which value search is requested.

For example, the following request limits value search to attributes `WineType` and `Name`:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
  <ContentElementConfig HandlerFunction="ValueSearchHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
 Id="ValueSearch"
  MaxPerProperty="3" Mode="All"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Value¬
SearchConfig">
    <SearchTerm>Merlot</SearchTerm>
    <RestrictToProperties>
      <Property>WineType</Property>
      <Property>Name</Property>
    </RestrictToProperties>
  </ContentElementConfig>
</Request>
```

Related Links

Value search query format on page 145
A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

Limiting the number of results per attribute

Another way to limit value search results is to specify the number of values to return for each attribute, using the `MaxPerProperty` attribute.

To set the number of attribute values to return for each attribute:

Use the `MaxPerProperty` attribute with an integer that specifies the number of values to return per attribute.

For example, the following query:

```
<ContentElementConfig xsi:type="ValueSearchConfig"
      HandlerFunction="ValueSearchHandler"
      HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
      Id="ValueSearch" Mode="Any" MaxPerProperty="1"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <SearchTerm>red</SearchTerm>
</ContentElementConfig>
```

returns one result, per each attribute:

Attributes	Values
`WineType`	`Red`
`Wineries`	`Green & Red`
`Drinkability`	`Drink with red meat`

Related Links

Value search query format on page 145

A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

Retrieving the number of matching results

The standard response to any value search request always includes information about the total number of matched values found, and whether all of them have been returned in this request. This information is returned in the `TotalValuesCount` and `HasMore` attributes on the `PropertyMatches` element.

A `PropertyMatches` element appears in the response only for those attributes in which matches were found, and contains attribute values for those matches. It contains two attributes that provide information on the number of values found and returned:

`TotalValuesCount`	Specifies the total number of matched values found per attribute.
`HasMore`	Specifies whether any results were cut off because of a limit specified in the request with `MaxPerProperty`.

The following abbreviated example illustrates how the information about found matches and returned matches is reflected in the response.

In this example, the MDEX Engine found four matching values within an attribute, which is reflected by `TotalValuesCount="4"`. However, because `MaxPerProperty` was set to 3 in the request (not shown in this topic), the response returns only three of the four found matches. This is indicated by `HasMore="true"`.

```
<PropertyMatches Name="Name" DisplayName="Wine Name" TotalValuesCount="4"
HasMore="true">
    <Match>
      <MatchingValue>Merlot Napa Valley Limited Reserve</MatchingValue>
    </Match>
```

```
    <Match>
      <MatchingValue>Biltmore Reserve Merlot</MatchingValue>
    </Match>
    <Match>
      <MatchingValue>Whitehall Lane Merlot</MatchingValue>
    </Match>
  </PropertyMatches>
```

Ordering results

Value search results consist of values grouped by attribute. Attributes in the result list are returned in an ascending alphabetical order.

The ordering of values, within each attribute, uses an order described as follows:

- If you specify a relevance ranking strategy to the MDEX Engine, then the order of results is ranked according to it.
- If you do not specify a relevance ranking strategy, the MDEX Engine uses the value for this strategy provided in the `DIMSEARCH_CONFIG` XML configuration document.
- Further, if the document does not provide a strategy, the MDEX Engine ranks the results using the three strategies in this order to break ties: `interp`, `exact` and `static(nbins,descending)`.

Specifying relevance ranking strategy for results

To rank the order of results received in response for a value search request, you can use the `RelevanceRankingStrategy` attribute.

If you specify a relevance ranking strategy to the MDEX Engine, then the order of results is ranked according to it.

To rank the order of results of the value search request:

Specify the value for the `RelevanceRankingStrategy` attribute in the `ValueSearchConfig` type of your Conversation Web Service request.
For example:

```
<ContentElementConfig xsi:type="ValueSearchConfig"
 HandlerFunction="ValueSearchHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

 Id="ValueSearch" Mode="Any" MaxPerProperty="5"
 RelevanceRankingStrategy="exact,static(nbins,descending)"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SearchTerm>red</SearchTerm>
</ContentElementConfig>
```

Related Links

Value search query format on page 145
A value search query uses a `ValueSearchConfig` complex type with a `SearchTerm` element. This element specifies search term(s) used by the MDEX Engine for a search against value-searchable standard attributes.

About relevance ranking modules on page 191
Relevance ranking modules are the building blocks from which you build the relevance ranking strategies that you actually apply to your search interfaces.

Chapter 16

Using Search Modes

By default, Endeca search operations return results that contain text matching all user search terms. In other words, search is conjunctive by default. However, in some cases a less restrictive matching is desirable, so that results are returned that contain fewer user search terms. This section describes the available search modes for record search and value search operations.

List of valid search modes

The search mode can be specified independently for each record search operation contained in a navigation query, as well as for the value search query.

Valid search modes are the following:

Search mode	Description
`All`	Match all user search terms (that is, perform a conjunctive search). This is the default mode.
`Partial`	Match some user search terms.
`Any`	Match at least one user search term.
`AllAny`	Match all user search terms if possible, otherwise match at least one. The `AllAny` search mode is not recommended in cases where queries can exceed two words. For example, a query on `womens small brown shoes` would return results on each of these four words and thus be essentially useless. In general, `AllPartial` is a better strategy.
`AllPartial`	Match all user search terms if possible, otherwise match some. Because you can configure this mode to match at least two or three words in a multi-word query, `AllPartial` is generally a better choice than `AllAny`.
`PartialMax`	Match a maximal subset of user search terms.

All mode

In `All` mode (the default mode), results must contain text matching each user search query term.

Partial mode

In `Partial` mode, results must contain text matching at least a certain number of user search query terms, according to the rules listed in this topic.

In `Partial` mode, results must contain text matching search query terms, according to the following rules:

- The **MIN_WORDS_INCLUDED** setting specifies the minimum number of user query terms that each result must match. If there are not enough terms in the original query to satisfy this rule, then the entire query must match.
- The **MAX_WORDS_OMITTED** setting specifies the maximum number of user query terms that can be ignored in the user query. If **MAX_WORDS_OMITTED** value is set to zero, any number of words can be ignored.

You can specify both of these settings with the `PARTIAL_MATCH` element in a `SEARCH_INTERFACE` configuration.

In `Partial` mode, result sets always include all of the results that an `All` query have produced, and possibly additional results as well.

Interaction of Partial mode and stop words

The presence of a stop word in a query reduces the minimum term count requirement for a document to match when `Partial` mode is used. The example in this topic explains the interaction between stop words and `Partial` mode.

The MDEX Engine treats stop words in a query as terms that match every document in the entire document set when counting how many terms must match a given query.

Therefore, the presence of a stop word in a query reduces the minimum term count requirement for a document to match by one, the presence of two stop words reduces it by two, and so on.

In practical terms, it means the result set may be both larger and more general than expected.

For example, consider a four-term query (such as `Medical Society of America`) against a search interface configured to allow `Partial` modes to require three terms to match. If one of those four terms (in this case `of`) is a stop word, only two of the other terms have to match, meaning results such as `Botanical Society of America` or `Medical Society Reunion` would be included in the set.

AllPartial mode

In `AllPartial` mode, the MDEX Engine first uses `All` mode to return results matching all search terms, if any are available.

If no such `All` results are available, the MDEX Engine returns the results that `Partial` would have produced. This allows a more conservative matching policy than `Partial`, because high-quality conjunctive results are returned if they exist and `Partial` results are used as a fallback on conjunctive misses.

This behavior, however, can be affected if cross-field matches are applied to the search interface. A search that matches “any” or “partial” inside of the same-field might be returned before a search that matches “all” of the terms but has to cross field boundaries to do so.

In addition, spell correction can also alter the results. A search that matches any or partial spell-corrected in a same field may return before a non-spell-corrected search that matches all terms in different fields.

To the user, this looks like there were no records matching all of the terms, even though there may be many that match cross-field.

Note: `AllPartial` is recommended for record search in a typical catalog application. The default configuration for `Partial`, which works well, can be adjusted to be more inclusive or conservative.

Any mode

In `Any` mode, results need only match a single user search term.

An `Any` result set always includes all of the results that an `All` or `Partial` query have produced, and possibly additional results as well.

Note: The `Any` mode is not recommended for use with record search in typical catalog applications.

AllAny mode

In `AllAny` mode, the MDEX Engine first uses `All` mode to return results matching all search terms, if any are available.

If no such `All` results are available, the MDEX Engine returns the results that `Any` would have produced.

Note: The `AllAny` mode is useful for value search.

PartialMax mode

`PartialMax` mode is a variant of the `AllPartial` mode: `All` results are returned if they exist.

If no such `All` results exist, then results matching all but one terms are returned; otherwise, results matching all but two terms are returned; and so forth.

`PartialMax` mode is subject to the **MIN_WORDS_INCLUDED** and **MAX_WORDS_OMITTED** settings used in the `Partial` mode. Hence, a `PartialMax` result set includes results if (and only if) the corresponding `Partial` result set includes results, and it contains a subset of the `Partial` results (possibly the entire set).

Configuring search modes

This topic summarizes options you can use to implement search modes.

No MDEX Engine configuration flags are necessary to enable search modes.

If you want to configure the minimum number of words for partial match modes and maximum number of words that may be omitted for partial match modes, you can specify these settings with the `PAR¬TIAL_MATCH` element in a `SEARCH_INTERFACE` configuration.

Query parameters for search modes

You can specify a search mode in the `Search` element of a Conversation Web Service query.

The search mode can be specified in a query by specifying the search mode in the `Mode` attribute.

The following example uses the `AllPartial` search mode:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
         <FilterState/>
         <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchOperator" Within="false">
          <Search SpellMode="Default" Mode="AllPartial" Key="AllProps">mer¬
lot</Search>
         </Operator>
         <ContentElementConfig xsi:type="RecordListConfig" HandlerFunc¬
tion="RecordListHandler"
            HandlerNamespace="http://www.endeca.com/MDEX/conversation/han¬
dlers/2010"
            Id="RecordList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
            <RecordsPerPage>5</RecordsPerPage>
         </ContentElementConfig>
      </Request>
   </soap:Body>
</soap:Envelope>
```

Chapter 17

Using Phrase Search

Phrase search allows users to specify a literal string to be searched. This section discusses how to use phrase search.

About phrase search

Phrase search allows users to enter queries for text matching of an ordered sequence of one or more specific words.

By default, an MDEX Engine search query matches any text containing all of the search terms entered by the user. Order and location of the search words in the matching text is not considered. For example, a search for `John Smith` returns matches against text containing the string `John Smith` and also against text containing the string `Jane Smith` and `John Doe`.

In some cases, the user may want location and order to be considered when matching searches. If one were searching for documents written by `John Smith`, one would want hits containing the text `John Smith` in the author field, but not results containing `Jane Smith` and `John Doe`.

Phrase search allows the user to put double-quote characters around the search term, thus specifying a literal string to be searched. Results of a phrase search contain all of the words specified in the user's search (not stemming, spelling, or thesaurus equivalents) in the exact order specified.

For example, if the user enters the phrase query `"run fast"`, the search finds text containing the string `run fast`, but not text containing strings such as `fast run`, `run very fast`, or `running fast`, which might be returned by a normal non-phrase query.

Additionally, phase search queries do not ignore stop words. For example, if the word `the` is configured as a stop word, a phrase search for `"the car"` does not return results containing simply `car` (not preceded by `the`).

Also, phrase search enables stop words to be disabled. For example, if `the` is a stop word, a phrase search for `"the"` can retrieve text containing the word `the`.

Because phrase searches only consider exact matches for contained words, phrase search also provides a means to return only true matches for a particular word, avoiding matches due to features such as stemming, thesaurus, and spelling.

For example, a normal search for the word `corkscrew` might also return results containing the text `corkscrews` or `wine opener`. Performing a phrase search for the word `"corkscrew"` only returns results containing the word `corkscrew` verbatim.

About positional indexing

To enable faster phrase search performance and faster relevance ranking with the Phrase module, your project builds index data out of word positions. This process is called positional indexing.

The MDEX Engine creates a positional index for standard and managed attribute values.

Phrase search is automatically enabled in the MDEX Engine at all times. For phrase search query processing, the MDEX Engine examines potential matching text to verify the presence of the requested phrase query string. This examination process can be slow in the following cases:

- The amount of text data is large (perhaps containing attribute values representing lengthy descriptions)
- The text that is being processed is offline (in the case of document text)

The MDEX Engine uses positional index data to improve performance in these scenarios. Positional indexing improves performance of multi-word phrase search, proximity search, and certain relevance ranking modules. The thesaurus uses phrase search, so positional indexing improves performance of multi-word thesaurus expansions as well. Positional indexing is enabled by default for Endeca attributes.

How punctuation is handled in phrase search

Unless they are included as special characters, all punctuation characters are stripped out, during both indexing and query processing. When punctuation is stripped out during query processing, the previously connected terms have to remain in their original order.

Example of phrase search

You can request phrase matching by enclosing a set of one or more search terms in quotation marks.

You can include phrase search queries in either record search or value search operations. You can combine phrase search with non-phrase search terms or other phrase terms.

Example of phrase search queries

The following example illustrates a phrase search query:

- A record search for a phrase `cd player` is as follows:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
        <Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
         <Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="SearchOperator" Within="false">
            <Search SpellMode="Default" Mode="All" Key="All">"cd play¬
er"</Search>
          </Operator>
        </Request>
      </soap:Body>
</soap:Envelope>
```

A value search for values containing the phrase `Samuel Clemens` is as follows:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
 <State/>
  <ContentElementConfig HandlerFunction="ValueSearchHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

   Id="ValueSearch" MaxPerProperty="3" Mode="All"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Value¬
SearchConfig">
    <SearchTerm>"Samuel Clemens"</SearchTerm>
    <RestrictToProperties>
      <Property>Authors</Property>
   </RestrictToProperties>
  </ContentElementConfig>
</Request>
```

Performance impact of phrase search

Phrase search queries are generally more expensive to process than normal conjunctive search queries.

In addition to the work associated with a conjunctive query, a phrase search operation must verify the presence of the exact requested phrase.

The cost of phrase search operations depends mostly on how frequently the query words appear in the data. Searches for phrases containing relatively infrequent words (such as proper names) are generally very rapid, because the base conjunctive search narrows the results to a small set of candidate hits, and within these hits relatively few possible match positions need to be considered.

On the other hand, searches for phrases containing only very common words are more expensive. For example, consider a search for the phrase `“to be or not to be”` on a large collection of documents. Because all of these words are quite common, the base conjunctive search does not narrow the set of candidate hit documents significantly. Then, within each candidate result document, numerous possible word positions need to be scanned, because these words tend to be frequently reused within a single document.

Even very difficult queries (such as `“to be or not to be”`) are handled by the MDEX Engine within a few seconds (depending on hardware), and possibly faster on moderate sized data sets. If such queries are expected to be very common, adequate hardware must be employed to ensure sufficient throughput. In most applications, phrase searches tend to be used far less frequently than normal searches. Also, most phrase searches performed tend to contain at least one information-rich, low-frequency word, allowing results to be returned rapidly (that is, in less than a second).

Chapter 18

Using Snippeting in Record Searches

This section describes how to use snippeting. Snippeting provides the ability to return an excerpt from a record in context, as a result of a user query.

About snippeting

The snippeting feature provides the ability to return an excerpt from a record — called a snippet — to an application user who performs a record search query.

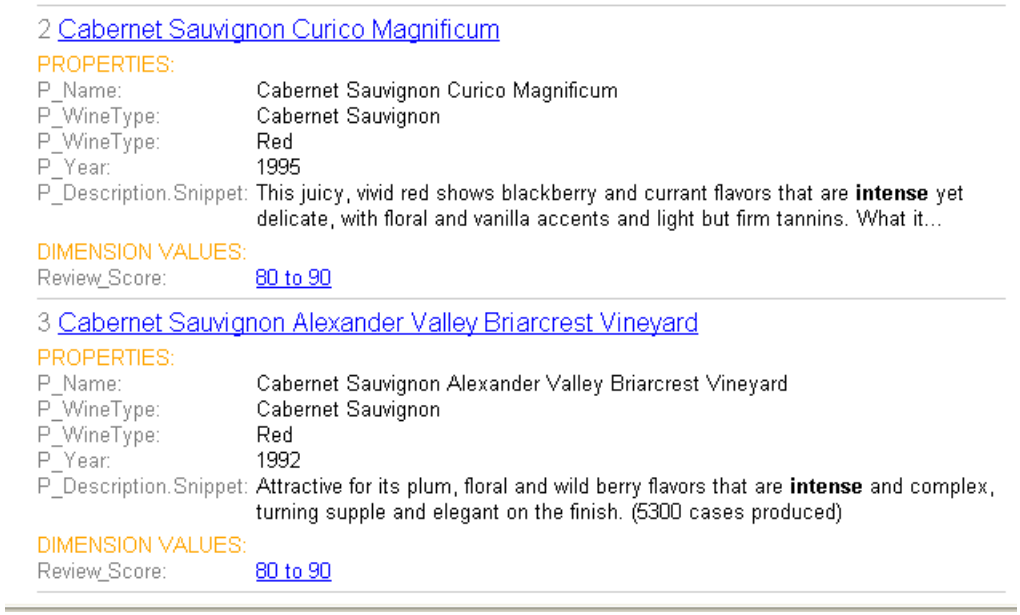
A snippet contains the search terms that the user provided along with a portion of the term's surrounding content to provide context. A front-end application powered by the Latitude Studio displays these snippets on the record list page of a query's results. With the added context, users can more quickly choose the individual records they are interested in.

A snippet can be based on the term itself or on any thesaurus or spell-correction equivalents. At least one instance of a term or equivalent is highlighted per snippet, regardless of the number of times the term or its equivalents appear in the snippet. A thesaurus or spell-corrected alternative may be highlighted instead of the term itself, even if both appear within the snippet.

You enable snippeting on individual members (fields) in a search interface that typically have many lines of content. For example, fields such as Description, Abstract, DocumentBody, and so on are good candidates to provide snippeting results. You can also enable snippeting on a per-query basis.

The result of a query with snippeting enabled contains at least one snippet in which enough terms are highlighted to satisfy the user's query. That is, if it is an AND query, the result contains at least one of each term, and if it is an OR query, it contains at least one of the alternatives.

For example, if a user searches for `intense` in a wine catalog, the record list for this query has many records that match intense. A snippet for each matching record displays on a record list page:


2 Cabernet Sauvignon Curico Magnificum

PROPERTIES:

P_Name: Cabernet Sauvignon Curico Magnificum
P_WineType: Cabernet Sauvignon
P_WineType: Red
P_Year: 1995
P_Description.Snippet: This juicy, vivid red shows blackberry and currant flavors that are **intense** yet delicate, with floral and vanilla accents and light but firm tannins. What it...

DIMENSION VALUES:

Review_Score: 80 to 90

3 Cabernet Sauvignon Alexander Valley Briarcrest Vineyard

PROPERTIES:

P_Name: Cabernet Sauvignon Alexander Valley Briarcrest Vineyard
P_WineType: Cabernet Sauvignon
P_WineType: Red
P_Year: 1992
P_Description.Snippet: Attractive for its plum, floral and wild berry flavors that are **intense** and complex, turning supple and elegant on the finish. (5300 cases produced)

DIMENSION VALUES:

Review_Score: 80 to 90

Snippet formatting and size

A snippet consists of search terms, surrounding context words, and ellipses.

A snippet can contain any number of search terms bracketed by`SnippetTerm` tags. The tags call out search terms and allow you to more easily reformat the terms for display in your front-end application.

The snippet size is the total number of search terms and surrounding context words. Although you can configure the total number of words in a snippet In order to adhere to the size setting for a snippet, it is possible that the MDEX Engine may omit some search terms and context words from a snippet. This situation becomes more likely if an application user provides a large number of search terms and the maximum snippet size is comparatively small.

A snippet consists of one or more segments. If there are multiple segments, they are delimited by ellipses in between them. Ellipses (`...`) indicate that there is text omitted from the snippet occurring before or after the ellipses.

Example of a snippet

For example, here is a snippet made up of two segments with a maximum size set at 20 words. The snippet resulted from a search for the search terms, `Scotland` and `British`, which are enclosed within <`SnippetTerm`> tags.

```
<SearchSnippet>
   <SnippetText>...in Edinburgh </SnippetText>
   <SnippetTerm>Scotland</SnippetTerm>
   <SnippetText>, and has been employed by Ford for 25 years...He first
joined Ford's
   </SnippetText>
   <SnippetTerm>British</SnippetTerm>
   <SnippetText> operation. Mazda motor...</SnippetText>
</SearchSnippet>
```

Enabling snippeting

You enable snippeting globally for the MDEX Engine via the RECSEARCH_CONFIG index configuration document.

The MDEX Engine has several index configuration documents that configure some features. You can edit them using the format specified in the *MDEX Engine Configuration XML Reference* appendix of the *Latitude Data Integrator Guide*. After these documents are edited, you can send them to the MDEX Engine using the Latitude Data Integrator.

The RECSEARCH_CONFIG document allows inclusion of SEARCH_INTERFACE, which in turn lets you specify snippet size for each of its members. The following example shows the syntax:

```
<RECSEARCH_CONFIG>
   <SEARCH_INTERFACE NAME="MySearch">
     <MEMBER_NAME SNIPPET_SIZE="12">Description</MEMBER_NAME>
   </SEARCH_INTERFACE>
</RECSEARCH_CONFIG>
```

The presence of SNIPPET_SIZE attribute enables snippeting for the MEMBER_NAME attribute, and the value of SNIPPET_SIZE specifies the maximum number of words a snippet can contain. Omitting this attribute or setting its value to zero disables snippeting.

Each member of a search interface is enabled and configured separately. In other words, snippeting results are enabled and configured for each member of a search interface and not for all members of a single search interface.

Note: A search interface member is an attribute that has been enabled for search and that has been added to the SEARCH_INTERFACE element.

You can enable and configure any number of individual search interface members. Each member that you enable produces its own snippet. Enabling a member in one search interface does not affect that member if it appears in other search interfaces. For example, enabling the **Description** attribute for Search Interface A does not affect the **Description** attribute in Search Interface B.

To enable snippeting:

1. In any text editor, edit the RECSEARCH_CONFIG document, similar to the following example:

   ```
   <RECSEARCH_CONFIG>
     <SEARCH_INTERFACE NAME="MySearch">
       <MEMBER_NAME SNIPPET_SIZE="10">Description</MEMBER_NAME>
     </SEARCH_INTERFACE>
   </RECSEARCH_CONFIG>
   ```

   In this example, snippet size is set to 10 for an attribute "Description".
2. Use the Latitude Data Integrator to send the RECSEARCH_CONFIG document to the MDEX Engine. For information, see the *Latitude Data Integrator Guide*.

Keep in mind that even if snippeting is enabled globally via the RECSEARCH_CONFIG index configuration document, you can control snippeting on a per-query basis by using the EnableSnippeting query attribute. For example, you can disable snippeting for a query by either setting EnableSnippeting to "false" or omitting it in the query.

Performance impact of snippeting

Enabling snippeting affects query runtime performance.

You can minimize the performance impact on query runtime by limiting the number of words in an attribute that the MDEX Engine evaluates to identify the snippet. This approach is especially useful in cases where a snippet-enabled attribute stores large amounts of text.

Provide the `--snip_cutoff` flag to the Dgraph to restrict the number of words that the MDEX Engine evaluates in an attribute. For example, `--snip_cutoff 300` evaluates the first 300 words of the attribute to identify the snippet.

If the `--snip_cutoff` Dgraph flag is not specified, or is specified without a value, the snippeting feature defaults to a cutoff value of 500 words.

Tips for snippeting

If a snippet is too short and you are not seeing enough context words in it, increase the value for `SNIPPET_SIZE` in the index configuration document. See the topic for enabling snippeting for the detailed format of the index configuration.

Retrieving snippets with the API

To request snippets with the Conversation Web Service, use the `SearchFilter` with the specified search interface. The Conversation Web Service returns snippets as part of the `<RecordListEntry>` element (which also returns records themselves).

Specifying the name of the search interface in the `SearchFilter` retrieves snippeting information:

```
 <SearchFilter Key="MySearch">
</SearchFilter>
```

where `My properties` is the name of the search interface for which snippeting is enabled for its members in the `RECSEARCH_CONFIG`.

Example request

The following request illustrates how to request a snippet with the Conversation Web Service:

```
<Request xmlns="http://www.endeca.com/MDEX/conversation/2010">
  <State>
    <SearchFilter Key="MySearch">red</SearchFilter>
  </State>
  <ContentElementConfig
    xsi:type="RecordListConfig"
    Id="RecordList"
    HandlerFunction="RecordListHandler">
    <Column>WineName</Column>
    <Column>WineDescription</Column>
```

```
  </ContentElementConfig>
</Request>
```

Example response

The following response from the Conversation Web Service returns snippeting information as part of the `RecordListEntry`:

```
<Results xmlns="http://www.endeca.com/MDEX/conversation/2010">
  ...
  <ContentElement Id="RecordList" xsi:type="cs:RecordList">
    ...
    <RecordListEntry>
      <Record>
        <WineName type="mdex:string">ARedWine</WineName>
        <WineDescription type="mdex:string">This wonderfully oaky red is
superb.</WineDescription>
      </Record>
      <ComputedProperties>
        <SearchSnippets Key="WineDescription">
          <SearchSnippet>
            <SnippetText>...wonderfully oaky </SnippetText>
            <SnippetTerm>red<SnippetTerm><SnippetText> is superb...</Snip¬
petText>
          </SearchSnippet>
        </SearchSnippets>
      </ComputedProperties>
    </RecordListEntry>
  </ContentElement>
</Results>
```

Enabling snippets per query with the API

You can enable snippets for a particular attribute on a per-query basis using the `SearchFilter` element in the Conversation Web Service.

Setting the `EnableSnippeting` attribute to `true` in the `SearchFilter` enables snippeting per query, for the specified attribute. The `SnippetLength` attribute sets the length of the snippet; the search term specifies the snippet term:

```
 <SearchFilter Key="WineDescription" EnableSnippeting="true"
SnippetLength="4">red</SearchFilter>
```

Note: Use these settings only if you need to specify snippeting information for a singular attribute for which there is no search interface configured. These settings do not override the settings that globally enable snippeting for members of the search interface in the `RECSEARCH_CONFIG > SEARCH_INTERFACE` index configuration document. In other words, if you enable snippeting at query time, snippets are returned only for Endeca attributes that have snippeting configured in the MDEX Engine.

Chapter 19

Using Wildcard Search

Wildcard search allows users to match query terms to fragments of words in indexed text. This section discusses how to use wildcard search.

About wildcard search

Wildcard search is the ability to match user query terms to fragments of words in indexed text.

Normally, Endeca search operations (such as record search and value search) match user query terms to entire words in the indexed text. For example, searching for the word `run` only returns results containing the specific word `run`. Text containing `run` as a substring of larger words (such as `running` or `overrun`) does not result in matches.

With wildcard search enabled, the user can enter queries containing the special asterisk or star operator (`*`). The asterisk operator matches any string of zero or more characters. Users can enter a search term such as:

```
*run*
```

which will match any text containing the string `run`, even if it occurs in the middle of a larger word such as `brunt`.

Wildcard search is useful for performing text search on data fields such as part numbers, ISBNs, and SKUs. Unlike cases where search is performed against normal linguistic text, in searches against data fields it may be convenient or even necessary for the user to enter partial string values. Details on how data fields that include punctuation characters are processed are provided in this section.

For example, suppose users were searching a database of integrated circuits for Intel `486 CPU` chips. The database might contain records with part numbers such as `80486SX` and `80486DX`, because these are the full part numbers specified by the manufacturer. But to end users, these chips are known by the more generic number `486`. In such cases, wildcard search is a natural feature to bridge the gap between user terminology and the source data.

Note: To optimize performance, the MDEX Engine performs wildcard indexing for words that are shorter than 1024 characters. Words that are longer than 1024 characters are not indexed for wildcard search.

Interaction of wildcard search with other features

The table in this topic describes whether various features are supported for queries that execute a wildcard search.

Feature	Support with wildcard search	Comments
Stemming	No	
Thesaurus matching	No	
Misspelling correction	No	Auto-correct and “Did You Mean?” are not supported.
Relevance ranking	Yes	
Snippeting	No	
Phrase search	No	
Why Did It Match	Yes	
Word interp	Yes	

Ways to configure wildcard search

You use the Latitude Data Integrator to configure wildcard search in your application, using one of the following options.

Related Links

Configuring wildcard search in record search on page 166
You make an attribute wildcard searchable in record searches by changing the value of the `mdex-property_TextSearchAllowsWildcards` attribute in the PDR, using the Latitude Data Integrator.

Configuring wildcard search in value search on page 167
You configure wildcard search during value searches in the Global Configuration Record (GCR), using the Latitude Data Integrator.

Configuring wildcard search for a search Interface on page 167
You can enable wildcard matching for a search interface by adding one or more wildcard-enabled attributes to the search interface.

Configuring wildcard search in record search

You make an attribute wildcard searchable in record searches by changing the value of the `mdex-property_TextSearchAllowsWildcards` attribute in the PDR, using the Latitude Data Integrator.

The `mdex-property_TextSearchAllowsWildcards` attribute of a PDR enables wildcard searches in record search against the attribute. The valid settings for this attribute are:

- If set to `true`, an attribute is wildcard searchable during record searches.
- If set to `false`, an attribute is not wildcard searchable during record searches. The default is `false`.

Note that it is an error if `mdex-property_TextSearchAllowsWildcards` is set to `true` but `mdex-property_IsTextSearchable` is set to `false`. In other words, an attribute must be record searchable in order for it to allow wildcard search in record searches.

Note: Before making this change, examine your data to decide which of the attributes in your record set need to be wildcard searchable. Also, turn off wildcard search in record searches for those attributes on which it won't be used by the users of your front-end application.

For information on how to configure wildcard search in record search for attributes, see the *Latitude Data Integrator Guide*.

Configuring wildcard search in value search

You configure wildcard search during value searches in the Global Configuration Record (GCR), using the Latitude Data Integrator.

Unlike the option for enabling wildcard search in text search which is performed by editing each attribute in its PDR (which affects only a single attribute), the GCR globally affects the enablement of wildcard search in value search for all attributes.

The `mdex-config_EnableValueSearchWildcard` attribute in the GCR specifies whether wildcard search should be enabled or disabled for value search across all attributes in the MDEX Engine. The valid settings for this attribute are:

- If set to `true`, wildcards are supported for value search.
- If set to `false`, wildcards are not supported for value search. The default is `false`.

Wildcard queries at the end of the search term, (for example, `gua*` for the search term `guarantee`), are always enabled even if wildcard search is disabled for value search for the attribute.

For information on how to configure wildcard search for value search, see the *Latitude Data Integrator Guide*.

Related Links

Interaction of value search and wildcard search on page 143

By default, value search allows wildcards at the end of the search term (such as `gua*` for the search term `guarantee`). To enable wildcards elsewhere in a search term, you need to set the `mdex-config_EnableValueSearchWildcard` attribute in the Global Configuration Record (GCR) to `true`, for the standard attribute in your records.

Configuring wildcard search for a search Interface

You can enable wildcard matching for a search interface by adding one or more wildcard-enabled attributes to the search interface.

First, add the desired attributes. Wildcard search can be partially enabled for a search interface. That is, some members of the search interface are wildcard-enabled while the others are not.

Searches against a partially wildcard-enabled search interface follow these rules:

- The search results from a given member follow the rules of its configuration. That is, results from a wildcard-enabled member follow the rules of wildcard search while results from non-wildcard members follow the rules for non-wildcard searches.
- The final result is a union of the results of all the members (whether or not they are wildcard-enabled).

You should keep these rules in mind when analyzing search results. For example, assume that in a partially wildcard-enabled search interface, `Property-W` is wildcard-enabled while `Property-X` is not. In addition, the asterisk (`*`) is not configured as a search character. A record search issued for `woo*` against that search interface may return the following results:

- `Property-W` returns records with `woo`, `wood`, and `wool`.
- `Property-X` only returns records with `woo`, because the query against this attribute treats the asterisk as a word break. However, it does not return records with `wool` and `wood`, even though records with those words exist.

However, because the returned record set is a union, the user will see all the records. A possible source of confusion might be that if snippeting is enabled, the records from `Property-X` will not have `wood` and `wool` highlighted (if they exist), while the records from `Property-W` will have all the search terms highlighted.

To enable wildcard search in a search interface:

1. Enable wildcard search in text search for members of the search interface. (This is controlled by the `mdex-property_TextSearchAllowsWildcards` attribute on the PDR, for each attribute member of the search interface).

   Wildcard search in text search can be partially enabled for a search interface. That is, some members of the search interface can be enabled for wildcard search in text search, while the others are not.
2. Add the desired attributes to the search interface in the `RECSEARCH_CONFIG` document.
3. Use the Latitude Data Integrator to send this document to the MDEX Engine. For information, see the *Latitude Data Integrator Guide*.

MDEX Engine flags for wildcard search

There are no MDEX Engine flags required to enable wildcard search. If wildcard is enabled in record search for an attribute, and is also enabled for value search, the MDEX Engine automatically enables the use of the asterisk operator (`*`) in appropriate search queries.

The following considerations apply to wildcard search queries that contain punctuation, such as `abc*.d*f`:

The MDEX Engine rejects and does not process queries that contain only wildcard characters and punctuation or spaces, such as `*.`, `* *`. Queries with wildcards only are also rejected.

The maximum number of matching terms for a wildcard expression is 100 by default. You can modify this value with the `--wildcard_max` flag for the Dgraph.

In case of wildcard search with punctuation, you may want to increase `--wildcard_max`, if you would like to increase the number of returned matched results.

Latitude Studio development for wildcard search

No specific Latitude Studio development is required to use wildcard search.

If wildcard search is enabled for record search and value search, users can use the **Search Box** component to enter search queries containing asterisk operators to request partial matching. If wildcard search is enabled for value search, type-ahead suggestions can be used in the **Search Box**.

Whereas the simplest use of wildcard search requires users to explicitly include asterisk operators in their search queries, some applications automate the inclusion of asterisk operators as a convenience, or control the use of asterisk operators using higher-level interface elements.

For example, an application might render a radio button next to the search box with options to select Whole-word Match or Substring Match. In Substring Match mode, the application might automatically add asterisk operators onto the ends of all user search terms. Interfaces such as this make wildcard search more easily accessible to less sophisticated user communities to which use of the asterisk operator might be unfamiliar.

Performance impact of wildcard search

To optimize performance of wildcard search, use the following recommendations.

- **Account for increased time needed for indexing**. In general, if wildcard search is enabled in the MDEX Engine (even if it is not used by the users), it increases the time and disk space required for indexing. Therefore, consider first the business requirements for your Endeca application to decide whether you need to use wildcard search.

  Note: To optimize performance, the MDEX Engine performs wildcard indexing for words that are shorter than 1024 characters. Words that are longer than 1024 characters are not indexed for wildcard search.

- **Do not use "low information" queries**. For optimal performance, Endeca recommends using wildcard search queries with at least 2-3 non-wildcarded characters in them, such as `abc*` and `ab*de`, and avoiding wildcard searches with one non-wildcarded character, such as `a*`. Wildcard queries with extremely low information, such as `a*`, require a significant amount of time to process. Queries that contain only wildcards, or only wildcards and punctuation or spaces, such as `*.` (star followed by period), or `* *` (star space star), are rejected by the MDEX Engine.
- **Analyze the format of your typical wildcard query cases**. This lets you be aware of performance implications associated with one specific wildcard search pattern.

  Do you have queries that contain punctuation syntax in between strings of text, such as `ab*c.def*?`

  For strings with punctuation, the MDEX Engine generates lists of words that match each of the punctuation-separated wildcard expressions. Only in this case, the MDEX Engine uses the `--wildcard_max <count>` setting to optimize its performance.

  Increasing the `--wildcard_max <count>` improves the completeness of results returned by wildcard search for strings with punctuation, but negatively affects performance. Thus you may want to find the number that provides a reasonable trade-off.

Chapter 20

Search Characters

This section describes the semantics of matching search queries to result text.

About search characters

The Endeca MDEX Engine supports configurable handling of punctuation and other non-alphanumeric characters in search queries.

This section does the following:

- Describes the semantics of matching search queries to result text (that is, records in record search or attribute values in value search) when either the query or result text contains non-alphanumeric characters.
- Explains how you can control this behavior using the search characters feature of the Endeca MDEX Engine.

Implementing search characters

Search indexing distinguishes between alphanumeric characters and non-alphanumeric characters and supports the ability to mark some non-alphanumeric characters as significant for search operations.

You mark a non-alphanumeric character as a search character in the Global Configuration Record.

Search characters are configured globally for all search operations. For example, adding the plus (+) character marks it as a search character for value search and record search operations.

To mark a non-alphanumeric character as a search character:

1. Edit the contents of the `mdex-config_SearchChars` element of the Global Configuration Record in any text editor, as in the following example.

   This example marks "+" and "_" characters as search characters. You can add more than one character; they are not separated by any delimiters.

   ```
   <mdex-config_SearchChars>+_</mdex-config_SearchChars>
   ```

2. To send the changes to the MDEX Engine, use the Latitude Data Integrator.

   For information, see the *Latitude Data Integrator Guide*.

Query matching semantics

The semantics of matching search queries to text containing special non-alphanumeric characters in the MDEX Engine is based on indexing various forms of source text containing such characters.

Basically, user query terms are required to match exactly against indexed forms of the words in the source text to result in matches. Thus, to understand the behavior of query matching in the presence of non-alphanumeric characters, one must understand the set of forms indexed for source text.

Categories of characters in indexed text

The Endeca system divides characters in indexed text into three categories:

- Alphanumeric characters including ASCII characters as well as non-punctuation characters in ISO-Latin1.
- Non-alphanumeric search characters (configured using the search characters feature, as described below).
- Other non-alphanumeric characters (this category is the default for all non-alphanumeric characters not explicitly configured to be in group 2).

During data processing, each word in the source text (that is, searchable attributes for record search, attribute values for value search) is indexed based on the alternatives for handling characters from the three categories, which is described in subsequent topics.

Indexing alphanumeric characters

Alphanumeric characters are included in all forms.

Because Endeca search operations are not case sensitive, alphabetic characters are always included in lowercase form, a technique commonly referred to as case folding.

Indexing search characters

Search characters are non-alphanumeric characters that are specified as searchable.

Search characters are included as part of the token.

Indexing non-alphanumeric characters

The way non-alphanumeric characters that are not defined as search characters are treated depends on whether they are considered punctuation characters or symbols.

- Non-alphanumeric characters considered to be punctuation are treated as white space. In a multi-word search with the words separated by punctuation characters, word order is preserved as if it were a phrase search. The following characters are considered to be punctuation: ! @ # & ( ) – [{ }] : ; ', ? / *
- Non-alphanumeric characters that are considered to be symbols are also treated as white space. However, unlike punctuation characters, they do not preserve word order in a multi-word search. If a symbol character is adjacent to a punctuation character, the symbol character is ignored. That is to say, the combination of the symbol character and the punctuation character is treated the same as the punctuation character alone. For example, a search on ice-cream would return the same results as a phrase search for “ice cream”, while a search for ice~cream would return the

same results as simply searching for ice cream. A search on ice-~cream would behave the same way as a search on ice-cream. Symbol characters include the following: ` ~ $ ^ + = < > “

Search query processing

The semantics of matching search query terms to result text containing non-alphanumeric characters are described in this topic.

- During query processing, each user query term is transformed to replace all non-alphanumeric characters that are not marked as search characters with delimiters (spaces).
  - Non-alphanumeric characters considered to be punctuation (! @ # & ( ) – [{ }] : ; ', ? / *) are treated as white space and preserve word order. This means that the equivalent of a quoted phrase search is generated. For that reason, all search features that are incompatible with quoted phrase search, such as spelling correction, stemming, and thesaurus expansion, are not activated. (For details, see the chapter "About phrase search.")
  - Non-alphanumeric characters that are considered to be symbols (` ~ $ ^ + = < > “) are also treated as white space. However, unlike punctuation characters, they do not preserve word order in a multi-word search.
- Alphabetic characters in the user query are replaced with lowercase equivalents, to ensure that they match against case-folded indexed strings.
- Each query term in the transformed query must exactly match some indexed string from the given source text for the text to be considered a hit.

As noted above, when parsing user-entered search terms, a query with non-searchable characters is transformed to replace all non-alphanumeric characters (that are not marked as search characters) with white space, but the treatment of word order depends on whether the character in question is considered to be a punctuation character or a symbol. The search behavior preserves the word order and proximity of the search term only in the case of punctuation characters.

For example, a search query for ice-cream will replace the hyphen (a punctuation character) with white space and return only records with this text:

- ice-cream
- ice cream

Records with this text are not returned because the word order and word proximity of text does not match the original query term:

- cream ice
- ice in the cream container

However, assuming the match mode is MatchAll, a search for ice~cream would return non-contiguous results for [ice AND cream].

MDEX Engine flags for search characters

There are no MDEX Engine flags necessary to enable the search characters feature. The MDEX Engine automatically detects the additional search characters.

The MDEX Engine supports an important closely related feature: automatic mapping of ISO-Latin1 international characters to ASCII equivalents in text search queries. You can specify this mapping with the `--latin1` flag for the Dgraph. This option allows search queries containing international characters

such as Spätlese to match against Anglicized result text such as Spatlese. Using the `--latin1` flag causes the Latin1 mappings to be applied to search queries.

Chapter 21

Working with Spelling Correction and Did You Mean

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

About Spelling Correction and Did You Mean

The Endeca MDEX Engine supports two complementary forms of Spelling Correction — automatic spelling correction for record search and value search, and explicit spelling suggestions for record search ("Did you mean?").

The Automatic 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).

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

Automatic 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 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 values and attributes. The MDEX Engine returns spelling-corrected results as normal search results, for both value search and record search operations.

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*.

Logic used for spelling correction

At a high level, the spelling engine in the MDEX Engine performs the following steps related to spelling correction for a given search query.

1. If the search terms generate more than a certain number of hits without any correction, then the spelling engine does not generate any corrections or suggestions.

   For the automatic correction, the threshold for the number of hits is 1. For the Did You Mean feature, the threshold for the number of hits is 20.

2. For each term in the query, the spelling engine finds the 32 corrections with the lowest spelling scores. A low spelling score signifies that the correction is similar to the search term.

   For the Aspell mode that the MDEX Engine uses, the spelling score is based on phonetic distance. The 32 corrections are pruned to corrections with a spelling score below a certain threshold. For the automatic correction, the spelling threshold is 125, for Did You Mean, the spelling threshold is 175.

3. The spelling engine tests each correction in place of the original search term it corrects. Only those corrections which increase the number of hits (relative to the original query) without reducing the number of terms matched are eligible to be returned.
4. The spelling engine selects the best correction based on which of the eligible corrections has the highest number of hits. For record search, this is the number of records matched. For value search, this is the number of records associated with the set of values matched.

Note: For more information about the difference in the treatment of results between record search and value search, see the section “How Value Search Treats Number of Results.”

To change the MDEX Engine configuration for Automatic Spelling Correction and DYM, you can rebuild the spelling dictionary with the `admin?op=updateaspell` command at any time. During the data ingest process, you can periodically run this command to update the spelling dictionary in the MDEX Engine.

Suggestions for automatic correction are not exposed by the MDEX Engine, that is, you cannot update the dictionary manually in the installed MDEX Engine.

In the Global Configuration Record, you can configure the Aspell indexing parameters such as minimum word occurrences, maximum and minimum word length. These parameters let you set the boundaries indicating to the MDEX Engine which words should be included in the spelling dictionary.

How value search treats number of results

Value search results may vary if spelling correction is performed.

An important note applies to the options and behavior associated with value 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, value 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), value search instead uses the number of records associated with the set of value search results computed for a given query.

In other words, value search follows an additional level of indirection to weight the value results computed by spelling suggestion queries according to the number of records than these 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 value and record search operations when given the same query terms.

updateaspell

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

Run this command after you have added data records to the MDEX Engine, to enable spelling correction in the MDEX Engine.

During the data ingest process, you can run the `admin?op=updateaspell`command periodically to update the spelling dictionary used by the MDEX Engine for Automatic Spelling Correction and DYM.

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

- Crawls the text search index for all terms which meet the constraint settings.

  The constraint settings include minimum word occurrences and maximum and minimum number of characters, for records and attribute values. The MDEX Engine uses these constraints to update the spelling dictionary. You can change them in the Global Configuration Record.

- Compiles a temporary text version of the `aspell` word list, `<db_prefix>.worddat`.
- Converts this word list to the binary format required by `aspell`
- Writes the generated binary file into the current index representation in the MDEX Engine.
- Makes the updated `aspell` spelling dictionary available in the MDEX Engine for processing of all queries arriving after this index update. The MDEX Engine uses this updated dictionary when processing all future queries.

Note: Because of the nature of continuous query, once the MDEX Engine processes this administrative request, it will start using the updated spelling dictionary after a certain point in its processing, and all newly incoming queries will be answered against the updated spelling dictionary. However, it is not possible to identify after which particular partial update or after which query the MDEX Engine will start using the newly updated spelling dictionary.

The Dgraph applies the updated settings while continuing to run queries and 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:

```
...
spellengine aspell ran successfully.
```

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.
```

Spelling mode (Aspell)

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 the low-level Aspell spelling module to compute single-word suggestions, that is, words similar to a given user query term and contained within the application-specific dictionary.

Aspell spelling module

The MDEX Engine supports one internal spelling module, Aspell. 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*).

Retrieving spelling suggestions and DYM in query results

You can retrieve spelling suggestion and did you mean (DYM) information in a query using the `SearchAdjustmentHandler` in the `ContentElementConfig` element of your Conversation Web Service request.

If spelling is enabled in the MDEX Engine, and, in addition to breadcrumbs, you want the Conversation Web Service response to contain supplemental information about spelling suggestions and DYM, a second `SearchAdjustmentHandler` `ContentElementConfig` is required. If it is included, spelling correction or DYM suggestions are returned as part of the response for `SearchAdjustmentHandler`. It is important to realize that if spelling is enabled, spelling auto-correction occurs even if the additional `ContentElementConfig` with `SearchAdjustmentHandler` is not included; however, while spelling correction takes place, the spelling correction and DYM suggestions are not returned in the response.

For example, the following abbreviated section of a query request contains `ContentElementConfig` for `SearchAdjustmentHandler`, to ensure that spelling correction and DYM suggestions are returned in the response:

```
<ContentElementConfig
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="SearchAdjustmentConfig"
    HandlerFunction="SearchAdjustmentHandler"
   HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"

    Id="SearchAdjustments"/>
```

The response would then be similar to the following. It contains suggested terms for DYM:

```
<cs:ContentElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="cs:SearchAdjustments" Id="SearchAdjustments">
<cs:SuggestedAdjustment RecordCountIfApplied="15">
  <cs:SearchFilter Key="Essay" Mode="All">jane</cs:SearchFilter>
  <cs:SuggestedTerms>can</cs:SuggestedTerms>
  <cs:Operator xsi:type="cs:ApplySpellingSuggestionOperator">
  <cs:SearchFilter Key="Essay" Mode="All">jane</cs:SearchFilter>
  <cs:Replacement>can</cs:Replacement>
  </cs:Operator>
```

```
  </cs:SuggestedAdjustment>
  </cs:ContentElement>
```

Configuring constraints for spelling dictionaries

The MDEX Engine selects words for the spelling dictionary based on predefined constraints. Modifying these constraints can be useful for improving performance of spell-corrected searches.

The constraint settings are available in the Global Configuration Record.

You can use these configuration settings 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 the settings which the MDEX Engine uses to generate spelling dictionary entries:

1. In the editor of your choice, edit the constraints in the GCR the MDEX Engine should use for adding words to the spelling dictionary.

   You can separately edit settings for entries in the dictionary for record search and value search. In other words, for each attribute assignment on a record, and for each attribute value, you could specify the following settings in the Global Configuration Record:

Attribute	Type	Description
`mdex-config_SpellingRecordMin¬WordOccur`	Int	Specifies the minimum number of times a word must occur in a standard attribute value (record assignment on an attribute) for it to be indexed for spelling correction. The default value is 4.
`mdex-config_SpellingRecordMin¬WordLength`	Int	Specifies the minimum number of characters that a word must contain in a standard attribute value (record assignment on an attribute) for it to be indexed for spelling correction. The default value is 3.
`mdex-config_SpellingRecordMax¬WordLength`	Int	Specifies the maximum number of characters that a word may contain for it to be indexed for spelling correction. The default value is 16.
`mdex-config_SpellingDValMinWor¬dOccur`	Int	Specifies the minimum number of times a word must occur in a managed attribute value for it to be indexed for spelling correction. The default value is 1.
`mdex-config_SpellingDValMin¬WordLength`	Int	Specifies the minimum number of characters that a word must contain in a managed attribute value for it to be indexed for spelling correction. The default value is 3.
`mdex-config_SpellingDValMax¬WordLength`	Int	Specifies the maximum number of characters that a word may contain for it to be indexed for spelling correction. The default value is 16.

2. To send the updated GCR to the MDEX Engine, use the Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.
3. Run the `admin?op=updateaspell` command on the MDEX Engine in order for these changes to take effect.

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`.

The following statements describe how word-break analysis works in the MDEX Engine:

- It is enabled by default.
- As part of the word-break analysis, the MDEX Engine removes breaks from the original term, or adds breaks to the original term if needed.
- The maximum number of word breaks that the MDEX Engine adds to or removes from a query is one.
- The minimum length for a new term created by word-break analysis is two characters. The MDEX Engine does not correct words that are smaller than 2 characters. For example, it does not correct `anear` to `a near`. It could correct to `an ear` if there are actual terms in the data corpus that match both `an` and `ear`.
- 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.

Troubleshooting Spelling Correction and Did You Mean

If spell-corrected results are not returned for words with expected spell-corrected options in the data, use these suggestions for troubleshooting.

- 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 can in some cases correct a word to one on the stop words list.

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.

It is important to carefully analyze the performance of the system together with application requirements prior to production application deployment.

Chapter 22

Using Stemming and Thesaurus

This section describes the tasks involved in implementing the Stemming and Thesaurus features of the Endeca MDEX Engine.

Overview of stemming and thesaurus

The 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 word roots and word derivations.

Stemming is enabled in the MDEX Engine by default and is available only for English. The configuration for stemming becomes known to the MDEX Engine once you run `mkmdex`.

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.

Note: The Endeca stemming implementation does not include decompounding. Decompounding is the ability to decompose a compound word (such as *kindergarten*) into its single word components (*kinder* and *garten*) and then find occurrences based on the smaller words.

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* will always produce a match.
- Stem form matches: Matches will 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) will also match.
- Inflected form matches: Matches will 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 will 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.

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, modifying, or deleting thesaurus entries

Thesaurus entries are added in the THESAURUS XML document.

All XML configuration documents are present in the MDEX Engine. You can edit them using the format specified in the *XML Configuration Reference*, found in the *Latitude Data Integrator Guide*. After these documents are edited, you can send them to the MDEX Engine using the Latitude Data Integrator, thus specifying the configuration your want.

To add a one-way or two-way thesaurus entry, or modify and delete existing thesaurus entries:

1. In any editor, edit the contents of the THESAURUS XML document.
2. Use the Latitude Data Integrator to send the THESAURUS document to the MDEX Engine.

   For information, see the *Latitude Data Integrator Guide*.

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.

Dgraph flags for stemming and thesaurus

Stemming and thesaurus data that has been configured is automatically enabled for use during text indexing and search query processing. In addition, there is no MDEX Engine configuration necessary to configure thesaurus and stemming information.

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

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 chapter in this 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 stop word, 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 stop word and also include a thesaurus equivalence between the words *item* and *items*. This will not automatically mark the word *items* as a stop word; 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 stop word (it is not uncommon to mark all or most single letter words as stop words). 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. In particular, 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.

Chapter 23

Relevance Ranking

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

About the relevance ranking feature

Relevance ranking lets you control the order in which search results are displayed to the end user of an Endeca application.

Typically, the relevance ranking feature is used to ensure that the most important search results are displayed earliest to the user, because users of search-oriented information retrieval systems 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 value search queries. You can establish a system-default relevance ranking for both record search and value search. In addition, you can assign relevance ranking on a per-query basis for both search types.

The importance of a search result is generally an application-specific concept. Thus, the relevance ranking feature provides a flexible, configurable set of result ranking modules. These modules can be used in combinations (called *relevance ranking strategies*) to produce a wide range of relevance ranking effects. Results are scored according to the order of ranking modules within the strategy.

Note: 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 in this chapter.

About 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.

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.

The Exact module is computationally expensive, especially on large text fields. It is intended for use only on small text fields (such as managed attribute values or small managed attribute values like part IDs). This module should not be used with large or offline documents. 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 standard or managed attribute member (or members) of the search interface that caused the document to match the query. Static field ranks are assigned based on the order in which members of a search interface are listed in the search interface configuration. The first 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 the `CROSS_FIELD_RELEVANCE_RANK` attribute of the `SEARCH_INTERFACE` element. This element is used 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 standard attributes, 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_RELEVANCE_RANK` attribute to a low integer such as 0 or 1.

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.

Note: The First module does not work with Boolean searches, cross-field matching, or wildcard search. 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 Frequency module for a result record is the sum of the frequencies of all user search terms in all fields (standard or managed attributes 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 Frequency 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).

Note: Due to performance issues, Endeca does not recommend using the Frequency module with standalone relevance ranking (that is, per-query relevance ranking).

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).

The Glom 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 Number of Terms (Nterms) module.

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 "Using Search Modes" chapter of this 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 the "Using Search Modes" chapter in this guide.
2. Within the above strata, all single-field matches are ranked ahead of all cross-field matches. For more information, see the "Working with Search Interfaces" chapter in this guide.
3. Within the above strata, all non-spelling-corrected matches are ranked above all spelling-corrected matches. See the "Implementing Spelling Correction and Did You Mean" chapter in this guide for more information.
4. Within the above strata, all thesaurus matches are ranked below all non-thesaurus matches. See the "Using Stemming and Thesaurus" chapter in this guide for more information.
5. Within the above strata, all stemming matches are ranked below all non-stemming matches. See the "Using Stemming and Thesaurus" chapter for more information.

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 setting in the search interface.
- 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 "Using Search Modes" chapter in this 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 is configured by editing theRELRANK_PHRASE XML element.

You add a Phrase module with the RELRANK_PHRASE element, which is a sub-element of the REL¬ RANK_STRATEGY element.

The following example shows a relevance ranking strategy named PhraseMatch with a Phrase module:

```
<RELRANK_STRATEGIES>
  <RELRANK_STRATEGY NAME="PhraseMatch">
    <RELRANK_PHRASE APPROXIMATE="TRUE" QUERY_EXPANSION="FALSE" SUB¬
PHRASE="TRUE"/>
  </RELRANK_STRATEGY>
</RELRANK_STRATEGIES>
```

To configure the Phrase module:

1. In any editor, edit the contents of the RELRANK_STRATEGIES configuration document to add or modify the RELRANK_PHRASE element.

   For details on these elements, see the appendix in the *Latitude Data Integrator Guide*. The resulting contents should look similar to the example above.
2. Send the changes to the MDEX Engine using the Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.

Details on the three options are explained in the following topic.

Phrase module options
The Phrase module has a variety of options that you use to customize its behavior.

The Phrase module has three options, which are configured via Boolean attributes:

- The APPROXIMATE attribute sets the use of approximate subphrase/phrase matching.
- The QUERY_EXPANSION attribute determines whether to apply query expansion (spell correction, thesaurus, and stemming).
- The SUBPHRASE attribute enables ranking based on length of subphrases.

These attributes belong to the RELRANK_PHRASE element.

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.

Query expansion

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.

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.

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. Extend subphrases to facilitate matching but rank based on the length of the original subphrase (before extension). Note that this combination can have a negative performance impact on query throughput.

Subphrase	Approximate	Expansion	Description
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.

Phrase module behavior

This topic describes some aspects of the behavior of the Phrase module with other features of the MDEX Engine.

Effect of search modes

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

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

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.

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, Endeca recommends that you enable subphrase if your results contain hyphenated words.

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.

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 an Endeca attribute, 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 Endeca attribute.

For example, using the module `Static(Availability,descending)` would sort result records in descending order with respect to their assignments from the Availability standard attribute. Using the module `Static(Title,ascending)` would sort result records in ascending order by their Title standard attribute assignments.

In a catalog application, setting the static module by Price, descending leads to more expensive products being displayed first.

For value search, the first parameter can be specified as `nbins`, `depth`, or `rank`:

- Specifying `nbins` causes the static module to sort result values by the number of associated records in the full data set.
- Specifying `depth` causes the static module to sort result values by their depth in the managed attributes hierarchy.
- Specifying `rank` causes values to be sorted by the ranks assigned to them for the application.

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.

The Weighted Frequency module ignores matches due to query expansion (that is, such matches are given a rank of 0).

Note: Due to performance issues, Endeca does not recommend using the Weighted Frequency module with standalone relevance ranking (that is, per-query relevance ranking).

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.

You can specify a relevance ranking strategy either in the request issued by the Conversation Web Service, and/or in the configuration XML document.

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 been consulted, they 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:

- You can configure relevance ranking to a search interface in the `RECSEARCH_CONFIG` configuration document, and send this document to the MDEX Engine using the Latitude Data Integrator.
- You can specify a relevance ranking strategy for a particular attribute to override the strategy specified for the selected search interface. This allows relevance ranking behavior to be fully customized on a per-query basis. For details, see the "Using standalone relevance ranking at the query level" topic.

Creating relevance ranking strategies

You create relevance ranking strategies by modifying the`RELRANK_STRATEGIES` index configuration document.

All index configuration documents are present in the MDEX Engine. You can edit them using the format specified in the *XML Configuration Reference* appendix of the *Latitude Data Integrator Guide*. After these documents are edited, you can send them to the MDEX Engine using the Latitude Data Integrator, thus specifying the configuration your want.

You create a relevance ranking strategy by adding one or more `RELRANK_STRATEGY` elements to the root `RELRANK_STRATEGIES` document.

Each `RELRANK_STRATEGY` element, in turn, contains one or more relevance ranking module elements, such as the `RELRANK_INTERP` and `RELRANK_FIELD` module elements in this WineMatch example:

```
<RELRANK_STRATEGIES>
  <RELRANK_STRATEGY NAME="WineMatch">
    <RELRANK_INTERP/>
    <RELRANK_STATIC NAME="Flavors" ORDER="ASCENDING"/>
    <RELRANK_FIELD/>
  </RELRANK_STRATEGY>
</RELRANK_STRATEGIES>
```

Keep in mind that the order of the module sub-elements defines the order in which the strategies are applied to the search results.

To create a relevance ranking strategy:

1. Edit the contents of the `RELRANK_STRATEGIES` document to add or modify the `RELRANK_STRAT¬EGY` elements.

For details on these elements, see the appendix in the *Latitude Data Integrator Guide*. The resulting contents of the edited document should look similar to the example above.

2. Send the RELRANK_STRATEGIES document to the MDEX Engine using the Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.

The new relevance ranking strategy can now be added to a search interface.

Implementing relevance ranking

You can create and control relevance ranking for both record search and value search at a system-default level.

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 attribute. You create search interfaces so you can apply behavior like relevance ranking across a group. You set the search interface for record search by modifying the RECSEARCH_CONFIG index configuration document and sending it to the MDEX Engine with the Latitude Data Integrator. For information about configuring relevance ranking in search interfaces, see the "Working with Search Interfaces" chapter in this guide.

For value search, the "Implementing relevance ranking for value search" topic in this chapter describes the configuration procedure.

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. When you add a Static module, be sure to set the two Static attributes:

- The NAME attribute sets the name of a Endeca attribute that is used for static relevance ranking.
- The ORDER attribute specifies how records should be sorted with respect to the specified Endeca attribute sets. The two values are ASCENDING and DESCENDING.

Ranking order for Field and Maximum Field modules

The Field and Maximum Field modules ranks results based on which Endeca attribute member 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 Endeca attributes ahead of those caused by lower-ranked Endeca attributes.

To change the relevance ranking behavior for these modules, you would move the search interface members to the appropriate position in the search interface (that is, move the MEMBER_NAME attributes up or down within the SEARCH_INTERFACE element).

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 for value search

You can define a system-default relevance ranking strategy for value search operations.

To define a system-default relevance ranking strategy for value search operations, modify the `REL¬RANK_STRATEGY` attribute of the `DIMSEARCH_CONFIG` index configuration document. To do so, create a text file with the configuration document and send it to the MDEX Engine, using the Latitude Data Integrator. For information, see the *Latitude Data Integrator Guide*.

The `RELRANK_STRATEGY` attribute specifies the name of a relevance ranking strategy for value search. The content of this attribute should be a relevance ranking string, as in this example:

```
<DIMSEARCH_CONFIG FILTER_FOR_ANCESTORS="FALSE" RELRANK_STRATEGY="exact,stat¬
ic(rank,descending)"/>

<DIMSEARCH_CONFIG FILTER_FOR_ANCESTORS="FALSE" RELRANK_STRATEGY="interp,ex¬
act"/>
```

For details on the format of the relevance ranking string, see the "Using standalone relevance ranking" topic.

The default ranking strategy for value search operations, which is applied if you do not make any changes to it, is:

```
interp,exact,static
```

Regardless of the `DIMSEARCH_CONFIG` setting for relevance ranking, you can specify a per-query relevance ranking strategy that overrides the `DIMSEARCH_CONFIG` setting. This procedure is documented in the "Using standalone relevance ranking" topic in this chapter.

Using standalone relevance ranking

At the MDEX Engine query level, relevance ranking strategies can be selected to override the default specified for the selected search interface. This is known as standalone relevance ranking.

Standalone relevance ranking allows relevance ranking behavior to be fully customized on a per-query basis. The `RelevanceRankingStrategy` attribute (of the `SearchFilter` element) must specify either the name of an existing relevance ranking strategy or the names of relevance ranking modules.

Relevance ranking module names can be any of these pre-defined modules:

- `exact`
- `field` (useful for record search only)
- `first`
- `freq` (not recommended for use with standalone relevance ranking, due to its performance impact)
- `glom` (useful for record search only)
- `interp`
- `maxfield` (useful for record search only)
- `nterms`
- `numfields` (useful for record search only)
- `phrase`
- `proximity`

- `spell`
- `stem`
- `thesaurus`
- `static`
- `wfreq` (not recommended for use with standalone relevance ranking, due to its performance impact)

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. In addition, module names are case-sensitive, so they must be specified in lower case.

Phrase module parameters

The `phrase` module can take from zero to three parameters to indicate whether subphrase matching, approximate matching, or query expansion should be enabled. 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 an Endeca attribute 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:

```
static(Price,ascending)
```

For value search, the first parameter can be specified as `nbins`, `depth`, or `rank`:

- Specifying `nbins` causes the `static` module to sort result values by the number of associated records in the full data set.
- Specifying `depth` causes the `static` module to sort result values by their depth in the attributes hierarchy.
- Specifying `rank` causes 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 value search results based on the value IDs.

Valid relevance ranking strings

The following are examples of valid relevance ranking strategy strings:

- `exact`
- `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)`

Specifying relevance ranking for queries

You can specify a relevance ranking strategy for both record search queries and value search queries.

Both types of queries let you specify either an existing relevance ranking strategy or the names of the relevance ranking modules.

Record search

For record search, the `RelevanceRankingStrategy` attribute of the `Search` element lets you specify a relevance ranking strategy for the query, as in this example:

```
<Operator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="SearchOperator" Within="false">
   <Search Mode="AllPartial" RelevanceRankingStrategy="exact"
      Key="Description">peach</Search>
</Operator>
```

For more information on the record search operator, see the "Using Record Search" chapter in this guide.

Value search

For value search, the `RelevanceRankingStrategy` attribute of the `ValueSearchConfig` type lets you specify a relevance ranking strategy for the query, as in this example:

```
<ContentElementConfig xsi:type="ValueSearchConfig"
 HandlerFunction="ValueSearchHandler"
 HandlerNamespace="http://www.endeca.com/MDEX/conversation/handlers/2010"
 Id="ValueSearch" Mode="Any" MaxPerProperty="5"
 RelevanceRankingStrategy="exact,static(nbins,descending)"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SearchTerm>red</SearchTerm>
</ContentElementConfig>
```

For more information on the `ValueSearchConfig` type, see the "Using Value Search" chapter in this guide.

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.

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 attribute	Author attribute
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 a text editor, we have defined a search interface named Books that contains both Title and Author standard attributes. The relevance rank is determined by the order in which the Endeca attributes appear in the 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*. Clearly all three of the records are hits, because each record has at least one searchable attribute value containing at least one occurrence of both the words Mark and Twain. But in what order should the results be presented to the user? Without relevance ranking enabled, the results will be 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 or by using query-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 attributes 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 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 Endeca attribute 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 a wine data set. 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, we establish the following two-way thesaurus entries:

```
<THESAURUS>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>cab</THESAURUS_FORM>
    <THESAURUS_FORM>cabernet</THESAURUS_FORM>
  </THESAURUS_ENTRY>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>cinnamon</THESAURUS_FORM>
    <THESAURUS_FORM>spice</THESAURUS_FORM>
    <THESAURUS_FORM>nutmeg</THESAURUS_FORM>
  </THESAURUS_ENTRY>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>tangy</THESAURUS_FORM>
    <THESAURUS_FORM>tart</THESAURUS_FORM>
    <THESAURUS_FORM>sour</THESAURUS_FORM>
    <THESAURUS_FORM>vinegary</THESAURUS_FORM>
  </THESAURUS_ENTRY>
  <THESAURUS_ENTRY>
    <THESAURUS_FORM>dusty</THESAURUS_FORM>
    <THESAURUS_FORM>earthy</THESAURUS_FORM>
  </THESAURUS_ENTRY>
</THESAURUS>
```

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 attribute, 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 attribute
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 attribute
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, Endeca 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 Endeca Professional 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 AllPartial. 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 the "Working with Search Interfaces" chapter in this 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. `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 AllPartial. 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. `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`, `Maxfield`, 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 6

Extending Latitude Studio

Chapter 24

Extending Latitude Studio

Out of the box, Endeca Latitude Studio includes numerous components that you can use to quickly develop an enterprise-quality search application. In addition, Latitude Studio provides a number of extension points for managing query and portlet operations, along with default implementations of the various interfaces that you can modify.

Developer tasks in Latitude Studio

Data source configuration tasks include:

- Modifying data sources.
- Adjusting security.
- Customizing how data sources interact with each other.

Component customization tasks include:

- Adding or modifying portlet components based on the `EndecaPortlet` class, using the Latitude Studio Component SDK.
- Localizing components.

This guide covers all of these developer tasks.

Note: Before modifying data source, make sure to read the data sources chapter of the *Latitude Studio Power User's Guide*. This chapter describes the default interaction model between related data sources.

Licensing requirement for component development

Latitude Studio component development may require the purchase of a third party license.

Latitude Studio uses *Ext JS* in its components and in the default components created by its SDK. An Endeca license does not bundle licensing for ExtJS. Therefore, customers developing components with ExtJS must either purchase their own development licenses from ExtJS, or remove ExtJS and develop components without the use of that Javascript framework.

Obtaining more information

Because Latitude Studio is built upon the Liferay Portal, you can access Liferay's documentation for more information about how to perform administrative and developer tasks.

Specifically, the *Liferay Portal Administrator's Guide* provides extensive information about installing, configuring, and maintaining a portal. To access a free PDF download of this guide, go to *http://www.liferay.com* and navigate to Documentation.

Liferay developer resources

This guide only covers Endeca extensions to the Liferay Portal. For additional developer support, Liferay provides blogs, wikis, and forums. To access this, go to *http://www.liferay.com* and navigate to Community.

The Endeca Developer Network (EDeN)

You can obtain more information about Latitude Studio and other Endeca products at the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Additional Endeca documentation

The complete Latitude documentation set can be accessed from the EDeN knowledge base.

Chapter 25

Security Extensions to Latitude Studio

You may require more than the default data source role-based security discussed in the *Latitude Studio Power User's Guide*. If so, you can customize the automated filtering of data from the MDEX Engine (based on user profile details such as the user's role or group association) by creating a custom `MDEX Security Manager`.

Security Manager class summary

This topic summarizes the `Security Manager` class.

An `MDEX Security Manager` is any concrete class that implements the `com.endeca.portal.data.security.MDEXSecurityManager`.

Abstract base class	`com.endeca.portal.data.security.MDEXSecurityManager`
Default implementation class	`com.endeca.portal.data.DefaultMDEXSecurityManager`
Description	Handles pre-execution query modification based on the user, role, or group-based security configuration of filters.
Default implementation behavior	The default `Security Manager` implementation makes use of the `securityEnabled`, `securityFilters`, `rolePermissions`, `inheritSecurity`, and `parentDataSource` properties. These properties are defined in data source configurations in order to apply role-based security filters to every query issued to the MDEX Engine backing a given data source. Users are assigned to Liferay roles in the Control Panel, and the related associations are made available to every portlet through the user's session. The `Security Manager` is responsible for maintaining an internal map of security filters for each data source that should always be applied to queries issued for that user's session.  **Note:** Record filters are the only supported type of `securityFilter`.  **Note:** `securityEnabled` defaults to `false` if the value is not present.

	Note: `inheritSecurity` defaults to `true` if the data source has a parent, and defaults to `false` if not.

Creating a new MDEX Security Manager

This topic describes the steps required to create an `MDEX Security Manager`.

To create a new `MDEX Security Manager` project:

1. In a terminal, change your directory to `endeca-extensions` within the Component SDK's root directory (normally called `components`).
2. Run one of the following commands:
   - On Windows: `.\create-mdexsecuritymanager.bat <your-security-manager-name>`
   - On Linux: `./create-mdexsecuritymanager.sh <your-security-manager-name>`

   This command creates a `your-security-manager-name` directory under `endeca-extensions`. This directory is an Eclipse project that can be imported directly into Eclipse if you use that as your IDE.

   Note: This directory also contains a sample implementation, which is essentially identical to the default implementation of the `Security Manager` used by Latitude Studio. You can use this sample implementation to help you understand how the `Security Manager` can be used.

Implementing a new MDEX Security Manager

Your `Security Manager` must implement the `applySecurity` method described in this topic.

There are two versions of the `applySecurity` method, one of which your `Security Manager` must implement:

```
public void applySecurity(PortletRequest request, MDEXState mdexState, Query
 query) throws MDEXSecurityException;
```

The Query class in this signature is `com.endeca.portal.data.Query`. This class provides a simple wrapper around an `ENEQuery`.

Using the MDEX Security Manager

In order to use your `MDEX Security Manager`, you must specify a new class for Latitude Studio to pick up and use in place of the default `Security Manager` implementation.

The `your-security-manager-name` directory you created contains an ant build file. The `ant deploy` task places a .jar file containing your `State Manager` into the `portal/tomcat-<version>/lib/ext` directory.

To specify your new class to Latitude Studio:

1. Point the cursor at the Dock in the upper-right corner of the page.
2. In the drop-down menu, choose **Control Panel**.
3. In the **Latitude** section of the **Control Panel** navigation panel, select **Framework Settings**.
4. Change the `df.mdexSecurityManager` property to the full name of your class, similar to following example:

   ```
   df.mdexSecurityManager = com.endeca.portal.extensions.YourSecurityManager¬
   Class
   ```

5. Click **Update Settings**.
6. Restart Latitude Studio so the change can take effect. You may also need to clear any cached user sessions.

Chapter 26

Managing Data Source State in Latitude Studio

Latitude Studio provides an extension point that allows you to define your own interaction model by creating a custom `MDEX State Manager`. In addition, in the *Latitude Studio Power User's Guide*, the data sources chapter describes the default interaction model between related data sources.

State Manager class summary

This topic summarizes the `State Manager` class.

An `MDEX State Manager` is any concrete class that extends from `com.endeca.portal.data.AbstractMDEXStateManager`. This class serves as a data source state manager that can be used to customize how data sources interact with each other during updates and query construction.

Abstract base class	`com.endeca.portal.data.AbstractMDEXStateManager`
Default implementation class	`com.endeca.portal.data.DefaultMDEXStateManager`
Description	Handles data source state updates and pre-execution query modification, based on data source relationships and configuration.
Default implementation behavior	The default state manager implementation makes use of the `ParentDataSource` property defined in data source configurations in order to propagate state changes throughout a hierarchy of data source relationships. When a portlet modifies the query state of its data source, that modification is applied to its parent data source and is also applied to all children of that parent. It is recursive in that it will apply all the way up and back down an ancestor tree. This allows application developers to create more advanced interfaces, such as tabbed result sets where a single **Guided Navigation** component should control the query state for **Results Table** components in individual tabs, by establishing a relationship hierarchy in data source configurations.

Creating a new MDEX State Manager

This topic describes the steps required to create an `MDEX State Manager`.

To create a new `MDEX State Manager` project:

1. In a terminal, change your directory to `endeca-extensions` within the Component SDK's root directory (normally called `components`).
2. Run one of the following commands:
   - On Windows: `.\create-mdexstatemanager.bat <your-state-manager-name>`
   - On Linux: `./create-mdexstatemanager.sh <your-state-manager-name>`

   This command creates a `your-state-manager-name` directory under `endeca-extensions`. This directory is an Eclipse project that can be imported directly into Eclipse if you use that as your IDE.

Note: This directory also contains a sample implementation, which is essentially identical to the default implementation of the `State Manager` used by Latitude Studio. You can use this sample implementation to help you understand how the `State Manager` can be used.

Implementing an MDEX State Manager

Your `State Manager` must implement the two methods described in this topic.

```
public void handleStateUpdate(PortletRequest request, MDEXState mdexState,
 QueryState newQueryState) throws QueryStateException;

public QueryState handleStateMerge(PortletRequest request, MDEXState
mdexState) throws QueryStateException;
```

- `handleStateUpdate()` is called when a portlet calls `DataSource.setQueryState(qs)`. This method should eventually call `mdexState.setQueryState()`. (However, if it determines that, for whatever reason, the `MDEXState's QueryState` should not change, it is not required to make this call.) `handleStateUpdate()` is also responsible for marking any data sources impacted by the update (which could depend upon your implementation of `handleStateMerge()`) so that portlets that listen to them on the page will properly update. For this reason, the `addEventTrigger(PortletRequest request, MDEXState ds)` method is provided for you to call, with the passed in request object and any `MDEXState` objects that are considered changed.
- `handleStateMerge()` is called when a portlet calls `DataSource.getQueryState()`. You are expected to return the `QueryState` that the portlet should get access to for the data source represented by the `mdexState`, taking into account any data source relationships or other aspects of your `State Manager` that might impact query state.

Using the MDEX State Manager

In order to use your `MDEX State Manager`, you must specify a new class for Latitude Studio to pick up and use in place of the default `State Manager` implementation.

The `your-state-manager-name` directory you created contains an ant build file. The `ant deploy` task places a `.jar` file containing your `State Manager` into the `portal/tomcat-<version>/lib/ext` directory.

To specify your new class to Latitude Studio:

1. Point the cursor at the Dock in the upper-right corner of the page.
2. In the drop-down menu, choose **Control Panel**.
3. In the **Latitude** section of the **Control Panel** navigation panel, select **Framework Settings**.
4. Change the `df.mdexStateManager` property to the full name of your class, similar to following example:

   ```
   df.mdexStateManager = com.endeca.portal.extensions.YourStateManagerClass
   ```

5. Click **Update Settings**.
6. Restart Latitude Studio so the change can take effect. You may also need to clear any cached user sessions.

Chapter 27

Installing and Using the Component SDK

You can customize Latitude Studio even further by creating your own components. The Latitude Studio Component SDK is a packaged development environment that you can use to add or modify portlets, themes, and layout templates. The Component SDK is a modified version of the Liferay Plugins SDK. The Endeca version includes enhancements such as the `EndecaPortlet` core class.

Downloading and configuring the Component SDK

You can download the Latitude Studio Component SDK from the Downloads section of the Endeca Developer Network (EDeN).

Before installing the Component SDK, download and unzip `Latitude_<version>_endeca-portal.zip`, as described in the portion of the *Latitude Installation Guide* for installing Latitude Studio. This is the base Latitude Studio code, upon which the Component SDK depends. You do not have to start Latitude Studio.

Note: Do not install the Component SDK in a directory path that contains spaces.

Note: On Windows, for steps b and d below, backslashes in paths must be escaped. That is, use a path similar to the following:

```
portal.base.dir=C:\\my_folder\\Latitude_endeca-portal
```

instead of:

```
portal.base.dir=C:\my_folder\Latitude_endeca-portal
```

To install the Component SDK:

1. Download and unzip `Latitude_<version>_components-sdk.zip` to a separate directory. This is the Component SDK itself.
2. Perform the following steps within the Component SDK:
   a) Create a file `components/build.<user>.properties`
      where `<user>` is the user name with which you logged on to this machine.
   b) Within that `properties` file, add a single property
      `portal.base.dir=<absolute_path_to_portal>`
      where `<absolute_path_to_portal>` is the path to the unzipped `Latitude_<version>_endeca-portal.zip`.

c) Create a `shared.properties` file in the `shared/` directory.
d) Edit `shared/shared.properties` and set the single property `portal.base.dir=<absolute_path_to_portal>`
where `<absolute_path_to_portal>` is the path to the unzipped `Latitude_<version>_endeca-portal.zip`.

Configuring Eclipse for component development

Before developing Latitude Studio components in Eclipse using the Component SDK, two Eclipse classpath variables need to be created.

 **Note:** Depending on your version of Eclipse, the steps below may vary slightly.

To configure the Eclipse classpath variables for Latitude Studio component development:

In Eclipse, go to **Window** > **Preferences** > **Java** > **Build Path** > **Classpath Variables** and create two new variables:

Name	Path	Example
DF_GLOBAL_LIB	Path to the application server global library.	`C:/endeca-portal/tomcat-<version>/¬ /lib`
DF_PORTAL_LIB	Path to the Liferay ROOT Web application library.	`C:/endeca-portal/tomcat-<version>/¬ webapps/ROOT/WEB-INF/lib`

Once these variables have been created, the components generated by the Component SDK are ready to be imported into Eclipse.

Component development overview

This topic provides a high-level overview of the component development process. Subsequent topics explain each step given here in greater detail.

To develop a new Latitude Studio component:

1. Create the component.
2. Import the project in Eclipse.
3. Build and test the new component.

Creating a new component

New Latitude Studio components are extensions of the `EndecaPortlet` class.

To create a new component:

1. At a command prompt, navigate to the Component SDK directory, and from there to `components/portlets`.
2. Run the command `create.bat` *`a-portlet-name-without-spaces`* `"A Friendly Portlet Name"` where:
   - The first argument must not have spaces. The string `-portlet` is automatically appended to the name.
   - The second argument is intended to be a more human-friendly name. Spaces are allowed, but if the name has spaces, it must be enclosed in quotation marks.

   An example command would be `create.bat johns-test "John's Test Portlet"`

Importing the project in Eclipse

Before beginning component development, you have to import the component project you just created into Eclipse.

To import the Latitude Studio Component SDK project you just created into Eclipse:

1. Within Eclipse, choose **File** > **Import** > **General** > **Existing Projects into Workspace**.
2. As the root directory from which to import, select the directory where you installed the Component SDK. You should see multiple projects to import.
3. Import the portlets you need to work with. If your portlets depend on shared library projects located within the `/shared` directory, import those as well.

Note: It takes some time for projects to build after they are imported.

Building and testing your new component

Next, you can build your new component in Eclipse and ensure that it appears in Latitude Studio.

To build your new component in Eclipse:

1. In your new project, open the `build.xml` file at the top level.
2. In the outline view, right-click the deploy task and select **Run as...** > **Ant Build**.

Note: This step is only necessary if you do not have **Build Automatically** checked in the Eclipse **Project** menu.

3. If Latitude Studio is not already running, log on to Latitude Studio and sign in.
4. Look at Latitude Studio logs to confirm that the component was picked up successfully.
5. Test your new component within Latitude Studio by choosing **Add Component** and looking in the **Sample** category. Add the new component to your page by dragging and dropping it.

Modifying Endeca enhancements to the Component SDK

The `build.xml` file in the root directory of each component created by the Component SDK contains three lines that control Endeca's build enhancements.

By default, these three lines are:

```
<property name="shared.libs" value="endeca-common-resources,endeca-discovery-
taglib" />
 <property name="endeca-common-resources.includes" value="**/*" />
 <property name="endeca-common-resources.excludes" value="" />
```

The properties control the behavior described below:

- The `shared.libs` property controls which of the projects in the `shared/` directory are included in your component. These shared projects are compiled and included as `.jar` files where appropriate.
- The `endeca-common-resources` `include` and `exclude` properties control which files in the `shared/endeca-common-resources` project are copied into your component. By default, all `endeca-common-resources` files are included, giving your component the Endeca AJAX enhancements (`preRender.jspf` and `postRender.jspf`) and the ability to switch between data sources in your component's preferences (`dataSourceSelector.jspf`). If your component needs to override any of these files, you must exclude them via these build properties or your code will be overwritten.

  These `include` and `exclude` properties can be specified for any shared library, as shown in the following example:

  ```
  <property name="endeca-discovery-taglib.includes" value="**/*" />
   <property name="endeca-discovery-taglib.excludes" value="" />
  ```

  When unspecified, `includes` default to `"**/*"` and `excludes` default to `""`.

Chapter 28

Working with QueryFunction Classes

Latitude provides a set of QueryFunction classes to allow you to filter and query data. You can also create and implement your own QueryFunction classes.

Provided QueryFunction classes

Latitude provides the following `QueryFunction` classes.

Note that the MDEX Engine for Latitude 2.1 does not support the following filters:

- `EQLFilter`
- `RecordAggregator`

Filters

Filters can be used for component development, as well as to filter the data included in a data source. For details on configuring data sources, see the *Latitude Studio Power User's Guide*.

Function Class	Configuration Properties	Notes
`NegativeRefinement¬ Filter`	`attributeValue`: String `attributeKey`: String `ancestors`: String[]	

`RangeFilter`	`attributeKey`: String
	`rangeOperator`:
	`(LT|LTEQ|GT|GTEQ¬`
	`|BTWN|GCLT|GCGT|GCBTWN)`
	`value1`: numeric
	`value2`: numeric (optional)
	`value3`: numeric (optional)	

`RecordFilter`	`recordFilter`: String	
`RefinementFilter`	`attributeValue`: long	

Function Class	Configuration Properties	Notes
	`attributeKey`: long `multiSelect`: `(AND|OR|NONE)` (optional) `navigable`: `(true|false)` (optional)	

`SearchFilter`	`searchInterface`: String
	`terms`: String
	`matchMode`:
	`(ALL|PARTIAL|ANY|`
	`ALLANY|ALLPARTIAL|`
	`PARTIALMAX|BOOLEAN)`
	`enableSnippeting`: boolean (optional; default is `false`)
	`snippetLength`: int (required if `enableSnippeting` is `true`)
		`searchInterface` can be either a search interface or an attribute enabled for text search.
		`enableSnippeting` is `false` by default.
		To enable snippeting, set `enableSnippeting` to `true`, and provide a value for `snippetLength`.

Configuration functions

Configuration functions are more advanced functions for component development.

Function Class	Configuration Properties	Notes
`AnalyticsQueryConfig`	`analyticsQuery`: String	

`AttributeValueSearchConfig`	`searchTerm`: String
	`maxValuesToReturn`: int (optional)
	`attribute`: String (optional)
	`searchWithIn`: List<String> (optional - list of attributes)
	`matchMode`:
	`(ALL|PARTIAL|ANY|`
	`ALLANY|ALLPARTIAL|`
	`PARTIALMAX|BOOLEAN)` (optional)
	`relevanceRankingStrategy`: String (optional)
		`searchWithIn` is a list of attributes against which to search; `attribute` is automatically included in this list.

`BreadcrumbsConfig`	`returnFullPath:` boolean (optional)	

Function Class	Configuration Properties	Notes
`ExposeRefinement`	`dimValId`: String `dimensionId`: String `ownerId`: String (optional) `dimExposed`: boolean (optional) `exposeAll`: boolean (optional) `maxRefinements`: int (optional; default is 100) `groupKey`: String (required) `groupExposed`: boolean (optional; default is `true`)	At least one of `dimValId` or `dimensionId`is required. `ownerId`can be the ID of a NavConfig instance. `dimExposed` indicates whether the attribute is exposed. `exposeAll` indicates whether the attribute is fully exposed (that is, the "More..." link is selected). `groupExposed` exposes or closes an entire group.
`NavConfig`	`exposeAllRefinements:` boolean `List<RefinementGroupConfigs>`: list of refinement group configs	If no `RefinementGroupConfigs` are specified, no attribute groups or attributes will be returned with the query.
`RecordDetailsConfig`	`recordSpecs`: Object (key-value pair)	Each new `RecordDetailsConfig` is appended to the previous `RecordDetailsConfig`
`ResultsConfig`	`recordsPerPage`: long `offset`: long (optional) `columns`: String[] (optional) `numBulkRecords`: int (optional)	
`ResultsSummaryConfig`		Specifies that results summarization should be included in returned results, but does not preclude other results.
`SearchKeysConfig`		
`SortConfig`	`property`: String `ascending`: boolean	
`SearchAdjustmentConfig`		

Creating a custom QueryFunction class

This topic describes the steps required to create a new `QueryFunction` class.

The steps below create a `QueryFilter` class, but the steps are analogous for creating a `QueryConfig` class.

Note: In order to create `QueryFunction` classes, you must install the Component SDK, which is a separate download.

To create a new `QueryFilter` class:

1. In a terminal, change your directory to `endeca-extensions` within the Component SDK's root directory (normally called `components`).
2. Run one of the following commands:
   - On Windows: `.\create-queryfilter.bat` *`your-query-filter-name`*
   - On Linux: `./create-queryfilter.sh` *`your-query-filter-name`*

   This command creates a `<`*`your-query-filter-name`*`>-filter` directory under `endeca-extensions`. This directory is an Eclipse project that can be imported directly into Eclipse if that is your IDE.

The `endeca-extensions` directory also contains an empty sample implementation of either a `QueryFilter` or a `QueryConfig`, depending on which batch script you ran. This has no effect on `QueryState` in its original form.

The skeleton implementation creates source files that do the following:

- Extends either `QueryFilter` or `QueryConfig`.
- Creates stubs for the abstract methods you need to implement: `applyToDiscoveryServiceQuery`, and `toString`.
- Creates default implementations for `getSetters` and `getGetters`. These use static `setters` and `getters` member properties that use reflection to extract the appropriate methods from the class.
- Creates a no-argument, protected, empty constructor. (The protected access modifier is optional, but recommended.)
- Creates a private member variable for logging.

Implementing a custom QueryFunction class

This topic describes the steps needed to implement your new `QueryFunction` class.

To implement your new `QueryFunction`, you must:

- Add private filter or configuration properties.
- Create getters and setters for any filter properties you add.
- For any property that is not a String, create a setter property that takes a String and does conversion.
- Define a no-argument constructor (protected access modifier optional, but recommended).
- Implement the abstract methods `getSetters`, `getGetters`, `applyToDiscoveryServiceQuery`, and `toString`. You can use the `getSetters` and `getGetters` methods from the sample `QueryFunction`.

Note: Because `.toString()` is used in `.equals()`, you should make sure that two `QueryFunction` objects that are the same return the same value. Specifically, `.toJSON().toString()` does not guarantee ordering of JSON properties, so two `QueryFunction` objects with the same member values may not return the same value if `.toString()` was implemented using `.toJSON().toString`.

Deploying a custom QueryFunction class

In order to use your new `QueryFunction`, you must deploy it to Latitude Studio.

The *`your-query-filter-name`*`-filter|config` directory that you created contains an ant build file.

The `ant deploy` task places a `.jar` file containing the custom `QueryFunction` into the `endeca-portal/tomcat-<version>/lib/ext` directory.

Note: If you are not using the default portal bundle, put the new `QueryFunction.jar` into the container's global classpath.

Restart Latitude Studio so that the portal picks up the new class file.

Once you have deployed your custom `QueryFunction`, you can use it in any component.

Adding the custom QueryFunction .jar file to your Eclipse build path

If you are using Eclipse as your IDE, you need to add the new `.jar` file to your build path of your custom component.

To add the new `.jar` file to your Eclipse build path:

1. Right-click on the project, and select **Build Path** > **Configure Build Path**.
2. Click the **Libraries** tab.
3. Click **Add Variable**, select **DF_GLOBAL_LIB** (which you should have added while setting up the SDK), and then click **Extend**.
4. Open the `ext/` directory and select the `.jar` file containing your custom `QueryFunction`.
5. Click **OK**.

After adding the `.jar` file to the build path, you can import the class, and use your custom `QueryFunction` or `QueryConfig` to modify your `QueryState`.

Obtaining query results

The `Results` class is used to represent results of queries.

Components are always encouraged to add the relevant `QueryConfig` to specify what types of results they need. Calls to `DataSource.execute()`, without any arguments, will continue to work on ENE Presentation API data sources, but are deprecated.

```
QueryState query = getDataSource(request).getQueryState();
query.addFunction(new NavConfig());
QueryResults results = getDataSource(request).execute(query);
```

You can then get the underlying API results and do whatever manipulation is required by your component.

```
Results discoveryResults = results.getDiscoveryServiceResults();
```

Before executing, you can also make other local modifications to your query state by adding filters or configurations to your query:

```
QueryState query = getDataSource(request).getQueryState();
query.addFunction(new ResultsConfig());
query.addFunction(new RecordFilter("Region:Midwest"));
QueryResults results = getDataSource(request).execute(query);
```

When you need to update a data source's state so that all associated components are updated, you must use `QueryState` instances.

```
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
query.addOperation(new RecordFilter("Region:Midwest"));
ds.setQueryState(query);
```

Chapter 29

Localizing Latitude Studio

Latitude Studio is an internationalized application that can be adapted for use in different locales. This section describes how to localize your Latitude Studio components.

Latitude Studio localization scenarios

Latitude Studio localization refers to two sets of tasks.

The first case is translating a component that has already been localized. In this scenario, you are applying the translation to components whose message strings have already been externalized to a resource bundle. Details on modifying and deploying a translated component appear in the next section.

The second, more involved case is developing or updating a component so that it supports localization. For details, see the section beginning with the topic "Setting up a component for localization."

Important: Latitude Studio supports only English data.

About adding a translation to a released component

This section discusses translating a component that has already been localized.

In this scenario, the component's English-language message strings have been externalized into the portlet WAR file's resource bundle. These strings can be translated to the target language and then made available to Latitude Studio.

Note: If you are working with a double-byte, extended character set language, consult the section "Working with non-Unicode characters" that appears later in this chapter before following the procedure below.

Adding a translation to a released component

This procedure can be followed whether you want to translate the content yourself or obtain the translation from a third party.

To add translated message strings to a released component:

1. Unzip the `.war` file of the localized component you want to modify.

2. Edit its `portlet.xml` file to enable the additional locale you want to support. For example, to add French, include `<supported-locale>fr</supported-locale>`.
3. In `WEB-INF/classes/com/endeca/` (or other location, based on your component's class structure), generate a `Resource_[locale].properties` file for the new language. This file should contain target-language values of the properties used in the component. To see the supported properties, refer to the `WEB-INF/classes/com/endeca/Resource_en.properties` file already in the component. Your file should contain a version of each of those messages in your target language.
4. Re-zip the `.war` file of the component and place it in the `endeca-portal/deploy` directory. Liferay hot-deploys the component.
5. Repeat steps 1 through 4 for each component you want to enable for your target language.
6. Start Latitude Studio and add your components, as well as the **Language** component, to the page.
7. In the **Language** component, click the flag associated with your target language.
   Latitude Studio displays the component messages from your resource bundle in your target language. In addition, because the portal itself is also localized, menus and other portal controls also appear in your target language.
8. In the **Language** component, click the United States flag to switch back to English.

Setting up a component for localization

This topic describes the steps needed to develop or update a component so that it supports localization.

To set up a portlet for localization:

1. Update the `portlet.xml` file to specify the locales this portlet will support.

   The following example enables English and German:

   ```
   <supported-locale>en</supported-locale>
   <supported-locale>de</supported-locale>
   ```

2. Update `portlet.xml` to specify the location of the portlet's resource bundle. (The resource bundle is the mechanism the Liferay Portal uses to add localized content to a portlet.)

   Continuing our example, we will include resource files `Resource_en.properties` and `Resource_de.properties` in the sample portlet's `com/endeca/portlet/sample/` directory:

   ```
   <resource-bundle>com.endeca.portlet.sample.Resource</resource-bundle>
   ```

3. Create resource bundles for your supported languages in `WEB-INF/src/[path/to/resource/bundle]_[locale].properties` (for example, the bundle for English for an Endeca component would be `WEB-INF/src/com/endeca/portlet/sample/Resource_en.properties`). For the most part, this is a simple `properties` file with key/value pairs for message IDs and their locale-specific messages.
4. Update your portlet's implementation to use the `LanguageUtils` class to retrieve messages from the resource bundle, rather than hard-coding message strings. This should be done for all messages displayed to the user, including form labels, portlet titles (and other metadata), warning and error messages, preferences pages, help text, and so on. See below for details on how to use the `LanguageUtils` class.

Note: See the sections below for details about portlet-specific messages and messages with tokens.

Note: You may note that the `resource-bundle` attribute is different from the file path you edit messages in. This is because the portlet build process combines common message strings from shared libraries with your portlet-specific messages to create the final `com/endeca/Resource_[locale].properties` file in the compiled portlet WAR. For more information, see the topic below on build process interaction with localization.

Build process interaction with localization

You should edit localization messages in a different resource file from the one you configure the portlet to read messages from.

The build process combines resource files into a single resource file that the component reads messages from. The build combines the component's `com/endeca/PluginResource_[locale].properties` file and any file found in a shared library's directory matching `com/endeca/*Resource_[locale].properties` into a single `com/endeca/Resource_[locale].properties` file. The messages from your component's `PluginResource_[locale].properties` appear at the top of the final `Resource_[locale].properties`, so you can easily override any messages from shared libraries. However, if your component includes more than one shared library, no guarantee can be made about the order in which the resource files from shared libraries will be appended.

Localizing your own shared libraries

If you have included localized messages in your shared libraries, make sure you choose a prefix other than Plugin for the resource file `com/endeca/[prefix]Resource_[locale].properties`. If you do not, this file will override your component's `com/endeca/PluginResource_[locale].properties` file during the build, and your final `com/endeca/Resource_[locale].properties` will be incorrect. Endeca recommends that you choose a prefix for your library's resource file that is distinct and similar to your library's name to avoid file name conflicts with components or other shared libraries.

Switching the locale of a component

Latitude Studio includes resources that you can use to switch a component's locale.

The **Language** component, described in the next topic, can be used to change the locale of a portlet.

There are also controls available in the **Display Settings** section of Liferay's Control Panel (as well as configuration properties in the `portal.properties` file) for setting the default container locale and the available locales.

For full details on using these Liferay features, see the *Liferay Portal documentation*.

Adding the Language component

To change the locale of the server, Endeca recommends using the **Language** component to select an alternate language.

The **Language** component is included in the default **Add Component** menu.

To add the **Language** component:

1. Point the cursor at the **Dock** in the upper-right corner of the page. The **Dock** is labeled "Welcome *<user name>*!"
2. In the drop-down menu, select **Add Component**.

The **Add Component** dialog box opens.

3. In the **Add Component** dialog box, expand the **Tools** category.
   A list of the available **Tools** components appears.
4. Click **Add**, or drag the **Languages** component to your portal page.

5. Click the flag representing the language you want to use. The portal will switch to that language, replacing English with the target language.
   For example, after clicking the Spanish flag, the **Dock** drop-down menu looks like this:

Including common externalized strings

All Latitude Studio components tend to include common messages, like those associated with the data source selector and those associated with saving preferences. The default localizations for these messages are automatically included in your compiled component.

The messages below are the default values. You can change or override these by including the same keys in your `PluginResource_[locale].properties` file.

```
### Common messages

df.portlet-does-not-support-datasource-api=Portlet does not support the API
 used by this data source.

# Data source selector messages
df.select-a-datasource=Select a data source
df.update-datasource=Update data source

df.no-data-source-selected=No data source selected for this portlet. Go to
```

```
 Preferences and select a data source.
df.no-data-source-specified=Error updating data source binding. No data
source was specified in the request.
df.data-source-binding-unchanged=Data source binding was not changed from
\"{0}\".
df.data-source-binding-unsupported-api=Data source binding was not changed
 from \"{0}\". Portlet does not support the API used by the data source
\"{1}\".
df.data-source-binding-changed-successfully=Data source binding successfully
 changed to data source \"{0}\".
df.data-source-binding-error=Error updating data source binding with new
data source name \"{0}\"; please notify your system administrator.

# Save preferences messages
df.save-prefs-success=Preferences updated successfully.
df.save-prefs-error=There was an error saving your preferences.
df.save-analytics-prefs-success=Analytics preferences updated successfully.
df.save-analytics-prefs-error=There was an error saving your analytics
preferences.
```

Note: Latitude Studio retrieves these localized messages with their English defaults. If the messages are not included in a portlet's resource bundle, Latitude Studio uses the hard-coded English defaults without displaying an error.

Including component-specific messages

Resource bundles should include a handful of component-specific messages that allow Latitude Studio to localize the name, description, keywords, and category of the component.

To localize the component's metadata, include the following messages:

```
javax.portlet.title=Sample Endeca Portlet
javax.portlet.short-title=Sample Endeca Portlet
javax.portlet.keywords=Sample, Endeca, Portlet
```

Additionally, if your component is displayed in the **Add Component** menu as part of a custom category (or sub-category), you may need to localize the name of the category. Take the following categories as an example:

```
<display>
  <category name="my.new.category">
    <category name="my.new.sub-category">
      <portlet id="portlet_A" />
    </category>
  </category>
</display>
```

To localize the category names, have your component's resource bundle include the following messages:

```
my.new.category=My Category
my.new.sub-category=My Sub-Category
```

If multiple components declare the same categories, they should all include these messages, since the component container uses the localized messages from the first component that specifies them.

Using tokens in message strings

Message strings can include tokens that are substituted at run-time.

For example, a search breadcrumb may need to display a spelling correction message like *"No matches found for 'bearign'; showing results for 'bearing'"*. This message would appear in a `.properties` file with tokens for the two terms, as in the following example:

```
autocorrect-msg=No matches found for \'{0}\'; showing results for \'{1}\'
```

When including this message in your portlet with the `LanguageUtils` utility, you pass in a list of parameters to substitute for these tokens. This substitution uses the class `java.text.MessageFormat`. Refer to the javadoc for that class for the options available with token substitution. Tokens may also do advanced substitution, such as date substitution formatted appropriately for the locale.

Using the Latitude Studio LanguageUtils class

The core class provided by Latitude Studio to access localized messages is `com.endeca.portlet.util.LanguageUtils`. There are several ways to use this class.

Calling static methods from Java

You can access `LanguageUtils` by calling static methods from your Java class.

The following example shows the static use of the `getMessage` methods to retrieve messages (with token substitution in the third line).

```
LanguageUtils.getMessage(request, "reset");
LanguageUtils.getMessage(request, "num-records");
LanguageUtils.getMessage(request, "search-for", new String[]{ "American"
});
```

A number of convenience method signatures are provided, allowing the user to specify the portlet request and message ID, and optionally to include parameters for token substitution and a default string. The default string may be useful for shared localized messages, allowing portlets to function with a default (un-localized) message if the localized message is not retrieved from the resource bundle.

All method signatures require specifying the `PortletRequest`.

Using the Discovery taglib in JSP

The Discovery `taglib` provides a tag for retrieving localized messages. This is the recommended way to retrieve localized messages in JSPs.

The following is an example using the `taglib`:

```
<%@ taglib uri='http://endeca.com/discovery' prefix="edisc"%>
<edisc:getMessage messageName="no-matching-values"/>

<edisc:getMessage messageName="message-with-params">
    <edisc:param value="test" />
</edisc:getMessage>
```

Using the LanguageUtils class from JSP

You can access `LanguageUtils` to retrieve localized messages in JSP pages.

This is similar to accessing `LanguageUtils` from Java.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" />
<portlet:defineObjects />
<%= LanguageUtils.getMessage(renderRequest, "reset") %>
```

Instantiating the object and call instance methods from Java/JSP

You can instantiate the `LanguageUtils` object and call methods from Java/JSP.

This approach provides the same convenience methods as the static approach, but simplifies the method signatures by removing the need to specify the request on every call. This may be useful for developers who make many calls for localized strings and would prefer to instantiate the object once and simplify the subsequent method calls.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" %>
<%
LanguageUtils lang = new LanguageUtils(renderRequest);
%>
<%= lang.getMessage("reset") %>
<%= lang.getMessage("num-records", "Num records:") %>
<%= lang.getMessage("search-for", "Search for \"{0}\"", new String[]{
"American" }) %>
```

Retrieving all messages from the resource bundle in one call from Java/JSP
You can retrieve all messages at once, in a single call from Java/JSP.

This approach may improve performance in portlets that require frequent access to the resource bundle and want to consolidate the message retrieval to a single call. The rest of the page then makes lookups into the loaded map.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" %>
<%@ page import="java.util.Map" %>
<%
Map<String, String> messages = LanguageUtils.getAllPortletMessages(render¬
Request);
%>
<%= messages.get("reset") %>
<%= messages.get("num-records") %>
<%= LanguageUtils.replaceMessageTokens(messages.get("search-for"), new
String[]{ "American" }) %>
```

Working with non-Unicode characters

This section describes how to work with non-Unicode characters in Latitude Studio.

Because Latitude Studio is Java-based, it can only read Unicode or Latin-1 characters. In the case of other characters, you can work around this limitation by converting the native file to ASCII, using a converter such as *native2ascii*, which is freely available as part of the JDK.

Keep in mind the following guidelines:

1. Use UTF-8 as your encoding. Lesser encodings cannot properly represent Japanese characters.
2. Pick a valid character set, such as Shift-JIS or UTF-8/Unicode, and stick with it. You cannot change character sets midstream—if you change character sets, you must re-enter your values.
3. Make sure the character set in your text editor matches the character set in `native2ascii`.

More information about working with non-Unicode characters can be found on the Liferay Portal Website.

Localizing a component to a non-Unicode language
The following example demonstrates how to localize a component to a double-byte, extended character language.

If you want to use this example as a learning exercise but do not have non-Unicode text of your own to deploy, you can machine-translate your English-language file and use that text in step 5 below.

To localize your portlet to a non-Unicode language (such as Japanese):

1. Within your portlet, create a file `PluginResource_<locale-code>.properties.native` at the appropriate location. For example, if you are working with Japanese, the file name would be `PluginResource_ja.properties.native`.
2. Commit both the `.native` and `.properties` file to your portlet. The `.properties` file is used by the portlet, but because that file uses escaped Unicode notation, it is extremely hard for humans to read. It is easier to make any necessary changes in the `.native` file.
3. Open the `.native` file in an encoding- and character-set-aware text editor such as Notepad++. Make sure the `.native` file uses UTF-8 as its encoding and Shift-JIS as its character set.
4. Copy the contents of the English resource bundle into the `.native` file.
5. Within your text editor, using your translation service, replace the English values with the Japanese values.
6. Save the file.
7. From the command line, run Java's `native2ascii` converter. This tool is typically included in the JDK. In the `encoding` argument, specify Shift_JIS as the character set, your `.native` file as the input, and your final `.properties` file as the output.

   ```
   native2ascii -encoding Shift_JIS PluginResource_ja.properties.native
   PluginResource_ja.properties
   ```

8. Commit both the `.native` and `.properties` file to your portlet. The `.properties` file is used by the portlet, but uses escaped Unicode notation, which is hard to read. The `.native` file is easier to modify.

Obtaining more information about portal localization

This topic provides links to additional information about localization provided by Liferay.

For information about editing Liferay's `Language_<langcode>.properties` file, which Liferay uses to localize the portal's strings, see the section "Languages and Time Zones" in the *Liferay Portal Administrator's Guide*. You can use this information to modify Liferay's translations as necessary.

For extensive documentation on Liferay language display customization, see this *wiki page*.

Appendix A

Suggested Stop Words

Stop words are words that are set to be ignored by the Endeca MDEX Engine.

About stop words

Typically, common words (like "the") are included in the stop word list. In addition, the stop word list can include the extraneous words contained in a typical question, allowing the query to focus on what the user is really searching for.

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.

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

List of suggested stop words

The following table provides a list of words that are commonly added to the stop word list; you may find it useful as a point of departure when you configure a list for your application.

In addition to some or all of the words listed below, you might want to add terms that are prevalent in your data set. For example, if your data consists of lists of books, you might want to add the word book itself to the stop word list, since a search on that word would return an impracticably large set of records.

You can add stop words using the Latitude Data Integrator.

a	do	me	when
about	find	not	where
above	for	or	why
an	from	over	with
and	have	show	you
any	how	the	your

are	I	under	
can	is	what	

Index

E

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W