Verity Query Language and Topic Guide

Version 6.0

August 19, 2005
Part Number DM0682

Verity, Incorporated
894 Ross Drive
Sunnyvale, California 94089
(408) 541-1500

Verity Benelux BV
Coltbaan 31
3439 NG Nieuwegein
The Netherlands

Copyright 2005 Verity, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of the copyright owner, Verity, Inc., 894 Ross Drive, Sunnyvale, California 94089. The copyrighted software that accompanies this manual is licensed to the End User for use only in strict accordance with the End User License Agreement, which the Licensee should read carefully before commencing use of the software.

Verity®, Ultraseek®, TOPIC®, KeyView®, and Knowledge Organizer® are registered trademarks of Verity, Inc. in the United States and other countries. The Verity logo, Verity Portal One™, and Verity® Profiler™ are trademarks of Verity, Inc.

Portions of this product Copyright 2003, Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Sun, Sun Microsystems, the Sun logo, Solaris, Java, the Java Coffee Cup logo, J2SE, and all trademarks and logos based on Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Xerces XML Parser Copyright 1999-2000 The Apache Software Foundation. All rights reserved.

Microsoft is a registered trademark, and MS-DOS, Windows, Windows 95, Windows NT, and other Microsoft products referenced herein are trademarks of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

WordNet 1.7 Copyright © 2001 by Princeton University. All rights reserved

Includes Adobe® PDF. Adobe is a trademark of Adobe Systems Incorporated.

Portions of this product use Teragram Software.

Includes IBM's XML Parser for C++ Edition.

Includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product may incorporate intellectual property owned by Microsoft Corporation. The terms and conditions upon which Microsoft is licensing such intellectual property may be found at

http://msdn.microsoft.com/library/en-us/odcXMLRef/html/odcXMLRefLegalNotice.asp?frame=true

All other trademarks are the property of their respective owners.

Notice to Government End Users

If this product is acquired under the terms of a **DoD contract:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of 252.227-7013. **Civilian agency contract:** Use, reproduction or disclosure is subject to 52.227-19 (a) through (d) and restrictions set forth in the accompanying end user agreement. Unpublished-rights reserved under the copyright laws of the United States. Verity, Inc., 894 Ross Drive Sunnyvale, California 94089.

7/26/05

Contents

Preface 13

Using This Book 14

Version 14

Organization of This Book 14

Stylistic Conventions 15

Related Documentation 17

Verity Technical Support 17

1 Overview 19

Evidence Operators 20

Proximity Operators 21

Relational Operators 22

Concept Operators 24

Modifiers 25

Advanced Operators 27

Topics 28

What is a Topic? 28

Relationship Between Topics and Topic Sets 29

PART I VERITY QUERY LANGUAGE

2 Elements of Query Expressions 33

Overview 34

Simple Queries 34

Operator/Modifier Names 35

Topic Names 35

Automatic Case-Sensitive Searches 35
Auto-Match Phrase to Topic Name 36
Explicit Queries 36
Syntax Options 37
Using Shorthand Notation 37
Specifying Topic Names Explicitly 37
Assigning Importance (Weights) to Search Terms 37
Searching Fields for Null Values 38
Precedence Evaluation 39
Precedence Rules 39
Parentheses in Expressions 42
Prefix and Infix Notation 42
Delimiters in Expressions 43
Angle Brackets for Operators 43
Braces in Expressions 43
Double Quotes for Reserved Words 43
Backslashes for Special Characters 44
Special Characters 44
Characters with Special Meaning 44
Punctuation in Queries 45
Qualify Instance Queries 45

3 Operators 47

Operators for Searching Full Text 47
ACCRUE 48
ALL 49
AND 50
ANY 50
BUTNOT 51
IN 51
NEAR 54
NEAR/n 55
OR 55
PARAGRAPH 56

- PHRASE 57
- SENTENCE 57
- SOUNDEX 58
- STEM 59
- THESAURUS 59
- TYPO/N 60
- WILDCARD 61
- WORD 63

Operators for Searching Text Fields 64
- CONTAINS 64
- ENDS 65
- MATCHES 65
- STARTS 66
- SUBSTRING 67

Operators for Searching Numeric Fields 67
- = (Equals) 67
- != (Not Equals) 67
- > (Greater Than) 68
- >= (Greater Than Or Equal To) 68
- < (Less Than) 68
- <= (Less Than Or Equal To) 69

4 Modifiers 71

CASE 71
LANG/ID 72
MANY 74
NOT 75
ORDER 76
WHEN 77

5 Advanced Query Language 81

Score Operators 82
- COMPLEMENT 82
- LOGSUM and LOGSUM/n 83

MULT/n 84
PRODUCT 85
SUM 85
YESNO 85
Natural Language Operators 86
FREETEXT 86
LIKE 87
Syntax 87
Special Characters in VdkVgwKey Fields 88
VdkVgwKey Fields on Windows Systems 89
Examples of LIKE Expressions 89
Efficiency Considerations 90

PART II TOPICS

6 Elements of Topic Design 93

About Topics and Topic Sets 94
Topic Structure 95
Topic and Subtopic Relationships 96
Storing Topic Sets 96
How Topics Work 96
Using Topics as Stored Queries in Other Verity Applications 97
Making Topics Available 97
Rules About Topics and Topic Sets 98
Operator Precedence Rules 98
Rules About Topics 98
Topic Design Strategies 99
Top-Down Design 99
Bottom-Up Design 100

7 Using Topic Outline Files 101

About Outline (OTL) Files 102
Creating a Topic Outline File 102
Defining Topics in the OTL File 104

Specifying Weights with Subtopics 105
Including and Excluding Documents 105
Specifying Field Evidence Topic Ranges 106
Topic Outline File Elements 107
Topic Definition Modifiers 107
Indentation Characters 109
Topic Structure 109
Defining Topic Structure 110
Defining Top-level Topics 110
Defining Subtopics 110
Subtopic Weight Assignments 111
Assigning the NOT Modifier to Subtopics 111
Evidence Topics 111
Evidence Topic Weight Assignments 113
Assigning Modifiers to Evidence Topics 113
Abbreviated Evidence Topics 114
Defining Subtopics Using the PHRASE Operator 114
Defining Field Evidence Topics 115
How Field Evidence Topics Affect Document Scores 117
Defining Topics for Zone Searching 117
Defining Topics Using Score Operators 118

8 Building Topic Sets from the Command Line 119

Starting mktopics 120
Building a Topic Set 120
Sample mktopics Command 121
mktopics Syntax 122
mktopics Syntax Summary 122
mktopics Syntax Descriptions 122
Checking Topic Precedence Rules 125
Topic Set Indexing 126
Topic Set Encryption 127
Before You Begin 127
Creating an Encryption File 128
Encrypting a Topic Set 128

APPENDIXES

A **Query Parsers** 133

Simple Queries 133

Words and Phrases Separated by Commas 134

Case-Sensitivity 134

How to Search Hyperlink Contents 135

Simple Query Parser 135

Query-By-Example (QBE) Parser 137

Internet-Style Parser 137

Search Terms 138

Including and Excluding Search Terms 138

Search Scope 139

Template Files 140

Query Syntax 142

Zone and Field Searches 142

Pass-Through of Terms 143

Stop Words 143

Testing the Templates 144

BooleanPlus Parser 145

Using Query Parsers Programatically 145

Obtaining a Query Parser Using the VDK API 145

Using VQL with the Internet Query Parser 147

B **Query Limits** 149

Search Time Limits 149

Operator Limits 150

C **Creating a Custom Thesaurus** 151

Creating a Thesaurus Control File 151

Control-File Structure 152

The control Directive. 153

The synonyms Keyword 153

The list Keyword 153

The qparser Keyword 154
Creating a Control File from an Existing Thesaurus 154
Using the LANG/ID Modifier in the Thesaurus Control File 156
Compiling a Thesaurus with mksyd 157
Integrating the Thesaurus with Verity 157
Naming and Installing the Thesaurus 157
Using a Knowledge Base Map to Point to a Thesaurus File 158

Index 159

Figures, Tables, and Listings

Table 1-1 Evidence Operators 20
Table 1-2 Proximity Operators 21
Table 1-3 Relational Operators 22
Table 1-4 Concept Operators 24
Table 1-5 Modifiers 25
Table 1-6 Modifier Preceding Operator Syntax 26
Table 1-7 Operator Preceding Modifier Syntax 26
Table 1-8 Advanced Operators 27
Table 2-1 Field Search Syntax 38
Figure 2-1 Operator Precedence 40
Table 2-2 Operator Precedence Rules 41
Table 2-3 Special Characters 44
Table 3-1 Evidence Operators 47
Table 3-2 Proximity Operators 48
Table 3-3 Concept Operators 48
Table 3-4 Supported XPath Subset 53
Table 3-5 Wildcard Characters 61
Table 6-1 Top-down Design Strategy 99
Table 6-2 Bottom-up Design Strategy 100
Table 7-1 Topic Outline File Elements 102
Table 7-2 Topic Definition Modifiers 108
Table 7-3 Evidence Topic Elements 112
Table 7-4 Field Evidence Topic Elements 115
Table 7-5 Field Evidence Topic Operators 116
Table 8-1 mktopics Syntax Elements 122
Table 8-2 mkenc Syntax Elements 130
Table A-1 Templates 141

Preface

The *Verity Query Language and Topic Guide* describes how to construct simple queries with Verity query language, and how the four parsers that are included with Verity products parse those queries.

This preface contains the following sections:

- Using This Book
- Related Documentation
- Verity Technical Support

Using This Book

This section briefly describes the organization of this book and the stylistic conventions it uses.

Version

The information in this book is current as of K2 Enterprise version 6.0. The content was last modified August 19, 2005. Corrections or updates to this information may be available through the Verity Customer Support site; see "Verity Technical Support" on page 17.

Organization of This Book

This book includes the following chapters and appendixes:

- Chapter 1, "Overview," provides an overview to the Verity query language, topics, and the contents of the entire guide.
- Part I, "Verity Query Language"
  - Chapter 2, "Elements of Query Expressions," provides detail on how to compose simple and complex queries using Verity query language elements, such as operators, modifiers, and syntax.
  - Chapter 3, "Operators," provides a reference to Verity query language operators.
  - Chapter 4, "Modifiers," provides a reference to Verity query language modifiers.
  - Chapter 5, "Advanced Query Language," provides a reference to Verity query language advanced (most sophisticated) operators.
- Part II, "Topics"
  - Chapter 6, "Elements of Topic Design," describes how to define topics and subtopics, create a Topic Outline File, and customize topic information.
  - Chapter 7, "Using Topic Outline Files," describes basic information about building an outline file to create topic sets using `mktopics` and Intelligent Classifier. The `.otl` file is required to make topics with `mktopics`, but is optional for use with Intelligent Classifier. Information includes options and command syntax for a topic outline (`.otl`) file.

- Chapter 8, "Building Topic Sets from the Command Line," describes how to build topic sets using the Verity `mktopics` command-line tool.

- Appendixes
  - Appendix A, "Query Parsers," provides a description of Verity query parsers.
  - Appendix B, "Query Limits," describes the search and operator limits for queries.
  - Appendix C, "Creating a Custom Thesaurus," describes how to create a specialized thesaurus to support synonym search.

Stylistic Conventions

The following stylistic conventions are used in this book.

Convention	Usage
Plain	Narrative text.
Bold	User-interface elements in narrative text: ■ Click **Cancel** to halt the operation.
Italics	Book titles and new terms: ■ For more information, see the *Verity K2 Getting Started Guide*. ■ An *index* is a Verity collection, parametric index, or recommendation index.
`Monospace`	File names, paths, and code: ■ The `name.ext` file is installed in: `C:\Verity\Data\`
`Monospace italic`	Replaceable strings in file paths and code: ■ `user` *`username`*
`Monospace bold`	Data types and required user input: ■ **`SrvConnect`** A connection handle. ■ In the **User Interface** text box, type **`user1`**.

The following command-line syntax conventions are used in this book.

Convention	Usage
`[ optional ]`	Brackets describe optional syntax, as in `[ -create ]` to specify a non-required option.
`|`	Bars indicate "either | or" choices, as in `[ option1 ] | [ option2 ]` In this example, you must choose between `option1` and `option2`.
`{ required }`	Braces describe required syntax in which you have a choice and that at least one choice is required, as in `{ [ option1 ] [ option2 ] }` In this example, you must choose `option1`, `option2`, or both options.
`required`	Absence of braces or brackets indicates required syntax in which there is no choice; you must enter the required syntax element.
`variable`	Italics specify variables to be replaced by actual values, as in `-merge` *`filename1`*
`...`	Ellipses indicate repetition of the same pattern, as in `-merge` *`filename1`*`,` *`filename2`* `[,` *`filename3`* `... ]` where the ellipses specify `,` *`filename4`*, and so on.

Use of punctuation—such as single and double quotes, commas, periods—indicates actual syntax; it is not part of the syntax definition.

Related Documentation

The following books provide more information on creating search applications:

- *Verity K2 Getting Started Guide*
- *Verity Developer Getting Started Guide*
- *Verity K2 Client Programming Guide*
- *Verity Developer's Kit Programming Reference*
- *Verity Command-Line Indexing Reference*
- *Verity Collection Reference*

Verity Technical Support

Verity Technical Support exists to provide you with prompt and accurate resolutions to difficulties relating to using Verity software products. You can contact Technical Support using any of the following methods:

Telephone: (403) 294-1107

Fax: (403) 750-4100

Email: tech-support@verity.com

Web: `http://www.verity.com`

Product documentation, release notes, and document updates are available at the Verity Customer Support Site, at

`https://customers.verity.com`

It is recommended that you periodically check the Customer Support site for the existence of updates to this and other Verity product documents.

Access to the contents of the Customer Support site requires a user name and password. To obtain a user name and password, follow the signup instructions on the Customer Support site home page. You will need to supply your Verity entity ID and Verity license key.

1

Overview

The Verity query language provides a rich language for writing queries that return relevant information. Queries can be composed to search full text only or full text in combination with field information. Simple syntax, such as words and phrases separated by commas, and more complex syntax involving operators and modifiers can be used.

This chapter provides an overview of the operators and modifiers that comprise the Verity query language. Material covered includes:

- Evidence Operators
- Proximity Operators
- Relational Operators
- Concept Operators
- Modifiers
- Advanced Operators
- Topics

Special queries called topics are included as part of the Verity query language. Topics are discussed in "Elements of Query Expressions" on page 33.

Verity toolkit and server products include query parsers. For information about the available query parsers, see "Query Parsers" on page 133.

Evidence Operators

Evidence operators can specify either a basic word search or an intelligent word search. A basic word search finds documents that contain only the word or words specified in the query. An intelligent word search expands the query terms to create an expanded word list so that the search returns documents that contain variations of the query terms. For example, the `THESAURUS` operator selects documents that contain the word specified, as well as its synonyms.

Documents retrieved using evidence operators are not relevance-ranked unless you include the `MANY` modifier. (For examples and more detailed information, see "MANY" on page 74.)

Table 1-1 briefly describes each evidence operator. For examples and more detailed descriptions, see "Operators for Searching Full Text" on page 47.

Table 1-1 Evidence Operators

Operator Name	Description
SOUNDEX	Expands the search to include the word you enter and one or more words that sound like, or whose letter pattern is similar to, the word specified. Collections do not have sound-alike indexes by default; you must build the sound-alike indexes to use this feature.
STEM	Expands the search to include the word you enter and its variations.
THESAURUS	Expands the search to include the word you enter and its synonyms.
TYPO/N	Expands the search to include the word you enter and words similar to the query term. This operator performs "approximate pattern matching" to identify similar words.
WILDCARD	Matches wildcard characters in search strings. Certain characters automatically indicate a wildcard specification.
WORD	Performs a basic word search and selects documents that include one or more instances of the specific word you enter.

Proximity Operators

Proximity operators specify the relative location of specific words in the document; that is, specified words must be in the same phrase, paragraph, or sentence for a document to be retrieved. In the case of the NEAR and NEAR/*n* operators, retrieved documents are relevance-ranked based on the proximity of the specified words. When proximity operators are nested, use the ones with the broadest scope first. Phrases or individual words can appear within SENTENCE or PARAGRAPH operators, and SENTENCE operators can appear within PARAGRAPH operators. Table 1-2 briefly describes each proximity operator. See "Operators" on page 47 for examples and more detailed descriptions.

Table 1-2 Proximity Operators

Operator Name	Description
ALL	Selects documents that contain all of the search elements you specify. A score of 1.00 is assigned to each retrieved document. <ALL> and <AND> are similar and they retrieve the same results. Queries using <ALL> are not relevance-ranked unless you use MANY; all retrieval results are assigned a score of 1.00.
ANY	Selects documents that contain at least one of the search elements you specify. A score of 1.00 is assigned to each retrieved document. <ANY> and <OR> are similar and they retrieve the same results. Queries using <ANY> are not relevance-ranked unless you use MANY; all retrieval results are assigned a score of 1.00.
BUTNOT	Selects documents that qualify your search term (word or phrase) by specifying one or more additional terms that cannot match the search term to count as a hit.
IN	Selects documents that contain specified values in one or more document zones. A document zone represents a region of a document, such as the document's summary, date, or body text. To search for a term only within the one or more zones upon which certain conditions have been placed, qualify an IN query with the WHEN modifier.
NEAR	Selects documents containing specified search terms. The closer the search terms are within a document, the higher the document's score.
NEAR/n	Selects documents containing two or more search terms within *N* number of words of each other, where *N* is an integer between 1 and 1024. The closer the search terms are within a document, the higher the document's score.

Table 1-2 Proximity Operators (continued)

Operator Name	Description
PARAGRAPH	Selects documents that include all of the search elements you specify within the same paragraph.
PHRASE	Selects documents that include the phrase you specify. A phrase is a grouping of two or more words that occur in a specific order.
SENTENCE	Selects documents that include all of the words you specify within the same sentence.

Relational Operators

Relational operators search document fields (such as AUTHOR) that have been defined in the collection. These operators perform a filtering function by selecting documents that contain specified field values. The fields that are used with relational operators can contain alphanumeric characters. Documents retrieved using relational operators are not relevance-ranked, and you cannot use the MANY modifier with relational operators.

A number of relational operators are available for numeric and date comparisons, including the following: = (equals), > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to). See "Operators for Searching Numeric Fields" on page 67 for examples and more detailed descriptions.

Table 1-3 describes the relational operators that are available for text comparisons. See "Operators for Searching Full Text" on page 47 for examples and more detailed descriptions.

Table 1-3 Relational Operators

Operator Name	Description
CONTAINS	Selects documents by matching the character string you specify with the values stored in a specific document field.
ENDS	Selects documents by matching the character string you specify with the ending characters of the values stored in a specific document field.
MATCHES	Selects documents by matching the character string you specify with values stored in a specific document field. Documents are selected only if the search elements specified match the field value exactly. When a partial match is found, the document is not selected.

Table 1-3 Relational Operators (continued)

Operator Name	Description
STARTS	Selects documents by matching the character string you specify with the starting characters of the values stored in a specific document field.
SUBSTRING	Selects documents by matching the character string you specify with a portion of the strings of the values stored in a specific document field.

When using the relational operators in combination with attributes, some operators are interpreted differently than when they are used in a field search.

For example, in the following construct the MATCHES operator is equivalent to the equals sign (=) and no wildcards are allowed.

```
<WHEN> attribute <OPERATOR> value
```

The following table shows the actual operators and values used when matching the attribute value in the query with the stored attributes in a collection's index. The use of wildcards is denoted by an asterisk (*).

Operator in the query	Actual operator used	Interpretation of the attribute's value
`= or MATCHES`	`WORD`	*value*
`STARTS`	`WILDCARD`	*value*`*`
`ENDS`	`WILDCARD`	`*`*value*
`SUBSTRING or CONTAINS`	`WILDCARD`	`*`*value*`*`

Concept Operators

Concept operators combine the meaning of search elements to identify a concept in a document. Documents retrieved using concept operators are relevance ranked. Table 1-4 briefly describes each concept operator. See “Operators for Searching Full Text” on page 47 for examples and more detailed descriptions.

Table 1-4 Concept Operators

Operator Name	Description
ACCRUE	Selects documents that include at least one of the search elements you specify. The more search elements that are present, the higher the score will be.
AND	Selects documents that contain all of the search elements you specify. An score is calculated for each retrieved document. <AND> and <ALL> are similar and they retrieve the same results. Queries using <AND> are relevance-ranked; retrieved documents are assigned a score between 0 and 1.00.
OR	Selects documents that contain at least one of the search elements you specify. A score is calculated for each retrieved document. <OR> and <ANY> are similar and they retrieve the same results. Queries using <OR> are relevance-ranked; retrieval documents are assigned a score between 0 and 1.00.

Modifiers

Modifiers are used in conjunction with operators to change the standard behavior of an operator in some way. When specified, a modifier changes the standard behavior of an operator in some way. For example, you can use the CASE modifier with an operator to specify that the case of the search word you enter be considered a search element as well.

Table 1-5 briefly describes each modifier. For examples and more detailed descriptions, see "Modifiers" on page 71.

Table 1-5 Modifiers

Modifier	Description
CASE	Performs a case-sensitive search.
DATE	Provides support for XML date operators. Used only to extend the WHEN modifier.
LANG/ID	Performs language-specific stemmed searches on collections created with the multilanguage locale.
MANY	Incorporates the density of search words in the calculation of the relevance-ranked score.
NOT	Excludes documents containing the words or phrases.
NUMERIC	Provides support for XML numeric operators. Used only to extend the WHEN modifier.
ORDER	Specifies the order in which search elements must occur in the document.
WHEN	Selects documents that contain specified values in one or more document zones upon which certain conditions have been placed. Used only with the IN operator.
ZONE	Provides support for XML element operands. Used only to extend the WHEN modifier.

Two syntax formats are used to specify modifiers with operators.

The first format specifies the modifier name before the operator name, as shown in Table 1-6. This format is valid for all four types of modifiers. Certain operators are valid only with certain modifiers.

Table 1-6 Modifier Preceding Operator Syntax

Modifier	Valid Operators	Examples
CASE	TYPO/n WORD WILDCARD	<CASE><WORD> iMac
LANG/ID	STEM	<LANG/fr><STEM>un
MANY	WORD WILDCARD STEM SOUNDEX PHRASE SENTENCE PARAGRAPH THESAURUS TYPO/n BUTNOT	<MANY><WORD> virtual
NOT	all operators	cat <AND> dog <AND> <NOT> pet
ORDER	PARAGRAPH SENTENCE NEAR NEAR/N ALL	president <ORDER> <PARAGRAPH> washington <ORDER> <SENTENCE> ("president", "washington")

The second syntax format specifies the operator name before the modifier name, as shown in Table 1-7. This syntax is valid only for the `CASE` and `NOT` modifiers.

Table 1-7 Operator Preceding Modifier Syntax

Modifier	Valid Operators	Examples
CASE	WORD WILDCARD CONTAINS MATCHES STARTS ENDS SUBSTRING	author <CONTAINS/CASE>Don
NOT	all operators	author<CONTAINS/NOT>don author<STARTS/NOT>xxx

Advanced Operators

The following are advanced classes of Verity operators. Advanced operators are not used with modifiers.

- The score operators (YESNO, PRODUCT, SUM, LOGSUM, MULT, and COMPLEMENT) affect how the search engine calculates scores for retrieved documents. When a score operator is used, the search engine first calculates a separate score for each search element found in a document, and then performs a mathematical operation on the individual element scores to arrive at the final score for each document.
- The natural language operators (FREETEXT and LIKE) enable you to specify search criteria using natural language syntax. The search engine uses natural language analysis to translate the query text into Verity query language expression for evaluating and scoring documents.

Table 1-8 briefly describes each advanced operator. For examples and more detailed descriptions, see "Advanced Query Language" on page 81.

Table 1-8 Advanced Operators

Operator Name	Description
COMPLEMENT	Score operator. Calculates scores for documents matching a query by taking the complement (subtracting from 1) of the scores for the query's search elements.
FREETEXT	Natural language operator. Interprets text using the free text query parser, and scores documents using the resulting query expression. All retrieved documents are relevance-ranked. For information about the free text query parser, see "Query-By-Example (QBE) Parser" on page 137
LIKE	Searches for other documents that are *like* the sample one or more documents or text passages you provide. The search engine analyzes the provided text to find the most important terms to use for the search. Retrieved documents are relevance-ranked.
LOGSUM and LOGSUM/n	Score operator. Returns a score that approaches 1 as the sum of the child node's score approaches 1.
MULT/n	Score operator. Multiplies the score returned from its child by the constant. This is the only operator that can return a negative number or a value greater than 1.
PRODUCT	Score operator. Calculates scores for documents matching a query by multiplying the scores for the query's search elements together.

Table 1-8 Advanced Operators (continued)

Operator Name	Description
SUM	Score operator. Calculates scores for documents matching a query by adding together the scores for the query's search elements.
YESNO	Score operator. Enables you to limit a search to only those documents matching a query, without the score of that query affecting the final scores of the documents.

Topics

Topics are the fundamental building blocks in Verity's classification infrastructure, and play a central role in defining categories. The following sections define a topic and explain the relationship between topics and topic sets.

What is a Topic?

A *topic* is a grouping of information related to a concept or a subject area. In Verity terms, a topic is a stored query expression that is written in the Verity Query Language (VQL). A topic models a concept of interest, which is used as the definition for a category. When a topic is evaluated against a set of documents, the Verity search engine identifies the subset of the documents that match the concept that the topic represents.

Consider the following scenario:

- You can use the VQL expression `GM <OR> Ford <OR> Chrysler` to model the concept “North American car manufacturers.” When this expression is evaluated on a set of newspaper articles, the Verity search engine selects all articles that mention GM, Ford, or Chrysler as matching the concept “North American car manufacturers.”
- Topics can be combined using VQL operators to create more complex topic definitions. For example, you might combine the concept “North American car manufacturers” with “European car manufacturers” (another VQL expression). By combining these topics and applying `<NOT>` to the concepts, you could perhaps create a new topic definition corresponding to the concept “Asian car manufacturers.” (This definition assumes no South American or Australian car manufacturers.)
- You can also use sophisticated non-Boolean VQL operators.

Operators and *modifiers* act as the glue that joins related evidence topics. Operators represent logic to be applied to evidence topics and define the criteria for the kinds of documents you want to find. Modifiers apply further logic to evidence topics. For example, a modifier can specify that documents containing an evidence topic not be included in the list of results.

Relationship Between Topics and Topic Sets

A *topic set* is a group of stored queries or topic definitions that have been compiled for use by a Verity application. A topic can be used to define a category, so a topic set contains one or more topics used for classifying documents in a collection. Because a topic set represents many concepts, it is sometimes referred to as a knowledge base. A Verity knowledge base can consist of one or more topic sets. When Verity applications incorporate topics, end users can find information by entering the topic names—instead of entering elaborate queries with complex syntax.

By using individual topics or combining topics, you can create *category definition* rules that are used to decide whether a document belongs to the category. There are several techniques for constructing topics, ranging from domain expertise to the use of automated machine learning techniques. Topics can be combined regardless of how they have been created. One advantage of combining topics is that it allows a gradual buildup so that basic topics can be shared between multiple higher-level topics.

Verity Intelligent Classifier is an application that runs on Microsoft Windows and has a graphical interface for designing, editing, and testing topics. With Intelligent Classifier, users can create topic definitions and build topic sets.

PART I

Verity Query Language

- Chapter 2: Elements of Query Expressions
- Chapter 3: Operators
- Chapter 4: Modifiers
- Chapter 5: Advanced Query Language

2

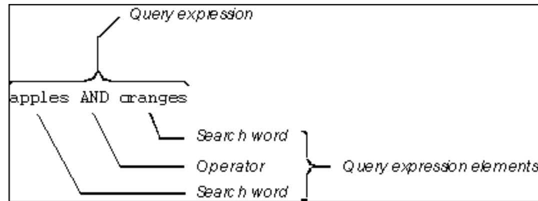
Elements of Query Expressions

This chapter describes the elements of Verity query language used to write simple query expressions. It provides the following information:

- Overview
- Simple Queries
- Explicit Queries
- Syntax Options
- Precedence Evaluation
- Delimiters in Expressions
- Special Characters
- Qualify Instance Queries

Overview

A *query expression* is any statement you enter as criteria for performing a search. The words and operators you use in a query expression comprise its *elements*.



When the simple query parser (the default parser) is used, you can state a query expression using simple or explicit syntax. The syntax you use determines whether the search words you enter will be stemmed, and whether the words that are found will contribute to relevance-ranked scoring.

For further information, see "Query Parsers" on page 133.

Simple Queries

When you use simple syntax, the search engine implicitly interprets single words you enter as if they were preceded by the `MANY` modifier and the `STEM` operator. By implicitly applying the `MANY` modifier, the search engine calculates each document's score based on the *word density* it finds; the denser the occurrence of a word in a document, the higher the document's score.

As a result, the search engine relevance-ranks documents according to word density as it searches for the word you specify, as well as words that have the same stem. For example, "films," "filmed," and filming" are stemmed variations of the word "film." To search for documents containing the word "film" and its stem words, enter the word "film" using simple syntax:

```
film
```

When documents are relevance-ranked, they are listed in an order based on their relevance to your search criteria. Relevance-ranked results are presented with the most relevant documents at the top of the list.

Operator/Modifier Names

Left and right angle brackets (< >) are reserved for designating operators and modifiers. They are optional for AND, OR, and NOT, but required in all other cases.

To include a backslash (\) in a search, insert two backslashes for each backslash character. To search for “C:\bin\print,” enter the following simple syntax:

```
C:\\bin\\print
```

Topic Names

For simple queries, simply enter the topic name as you would a word or phrase.

The search engine also interprets words that are topic names as topics rather than as individual words when you use simple syntax. This means that if the text you enter contains a topic name, the query corresponding to that topic is used instead of the word itself.

Automatic Case-Sensitive Searches

The search engine attempts to match the case-sensitivity provided in the query expression when mixed case is used. For search terms entered completely in lowercase or completely in uppercase, the search engine looks for all mixed-case variations.

Search terms with mixed case automatically become case-sensitive. For example, a query on `Apple` behaves as if you had specified `<case>Apple` (which would find only the precise string `Apple`), while a query on `apple` finds all of the following: `APPLE`, `Apple`, `apple`.

A query all in uppercase does not turn on case-sensitive searching. A query on `APPLE` finds all of the following: `APPLE`, `Apple`, `apple` (as before).

The CASE modifier has the same effect as in previous releases. When used, the case-sensitivity of the query is preserved. For example, if you want to search for the term “OCX” and want to find instances of “OCX” in uppercase only, you could enter the following query:

```
<CASE> <WORD> OCX
```

The search engine would interpret the previous query expression to mean: find all documents containing one or more instances of the word “OCX” spelled in uppercase, not mixed case.

Auto-Match Phrase to Topic Name

If your query expression includes a phrase, the search engine tries to match the phrase with a topic name by substituting the spaces with hyphens. For example, if the phrase "web server" is used in a query expression, the search engine looks for a topic named "web-server." If a match is found, the topic name is used to perform the search.

Partial matches are not valid. For example, the search engine does not match the phrase "web server features" with the topic name "web-server;" it matches the topic name "web-server-features."

Explicit Queries

When you enclose individual words in double-quotation marks ("), the Verity search engine interprets those words literally. For example, by entering the word "film" explicitly in double-quotation marks, the words "films," "filmed," and filming" are not considered in the search. To select documents containing the word "film" without searching for its stemmed words, enter the word "film" using explicit syntax:

```
"film"
```

The following example retrieves documents that contain both the literal phrase "pharmaceutical companies" and the literal word "stock":

```
AND ("pharmaceutical companies", "stock")
```

The `AND` operator does not require angle brackets because it is automatically interpreted as an operator.

The following example retrieves documents containing the phrase "black and white":

```
<PHRASE> (black "and" white)
```

The `PHRASE` operator requires angle brackets, and the "and" is enclosed in double-quotation marks (") because it is to be interpreted as a literal word, not as an operator.

Additionally, when you enter a topic name enclosed in double-quotation marks ("), the search engine will interpret the topic name as a literal word instead of as a topic. This is useful when you want to search for a word that is the same as the name of a topic.

Syntax Options

Following is a summary of Verity query language syntax options, including alternative syntax and topics, that you can use to compose query expressions. These syntax options are available for simple and explicit queries.

Using Shorthand Notation

The Verity query language provides a few alternatives you can use to specify evidence operators. In the following examples, *"word"* represents the word to be located.

Standard Query Expression	Equivalent Format
`<MANY><WORD>`*word*	"word"
`<MANY><STEM>`*word*	'word'

Specifying Topic Names Explicitly

You can specify topics in expressions in a variety of ways. Use any of the following formats to specify a topic explicitly in an expression:

{*topic_name*}

<TOPIC>*topic_name*

<TOPIC>(*topic_name*)

{*KB*:*topic_name*}

In the previous examples, *topic_name* represents the name of the topic used in the expression. *KB* represents the name of the knowledge base used in the expression.

Assigning Importance (Weights) to Search Terms

You can assign a weight to each search term in a query to indicate each search term's relative importance. The weight assignment is expressed as a number between 01 and 100, where 01 represents the very lowest importance rating and 100 represents the very highest importance rating.

To specify a weight with a search term, enter the weight in brackets just before the search term, as shown in the following example:

```
[50]test, [80]help
```

For the previous example, the search engine looks for stemmed variations of the words "test" and "help" and assigns a weight of 50 to the term "test" and a weight of 80 to the term "help." Search results with the highest density of stemmed variations of the term "help" would receive the highest possible scores.

Using explicit syntax, you could enter a query expression with weights as follows:

```
<ACCRUE> ([50]<WORD>(test), [80]<WORD>(help))
```

Searching Fields for Null Values

The search engine supports searching for fields that have a null value. This means that you can perform the basic search and find all of the documents that have a null value for a particular field. You can also search for fields that are populated with a non-null value.

The methods for searching for null or populated field values are indicated in Table 2-1.

Table 2-1 Field Search Syntax

Syntax	Description
fieldname = ""	This syntax is used to search for documents that have a null value for the field named *fieldname*. The value for *fieldname* must be a valid Verity field. If the field name given does not exist for a document, meaning the field is not defined for the document's collection, it does not match the query.
fieldname != ""	Used to search for documents that have some value for the field named *fieldname*. The value for *fieldname* must be a valid Verity field. If the field name given does not exist for a document, meaning the field is not defined for the document's collection, it does not match the query.

Precedence Evaluation

The ways that precedence rules and syntax affect the evaluation of queries are described in the following sections.

Precedence Rules

A Verity query expression is read using explicit precedence rules applying to the operators that are used. Although a query expression is read from left to right, some operators carry more weight than others; this affects the interpretation of the expression. For example, the `AND` operator takes precedence over the `OR` operator. For this reason, the following example is interpreted to mean: Look for documents that contain *b* and *c*, or documents that contain *a*.

```
a OR b AND c
```

To ensure that the `OR` operator is interpreted first, use parentheses as follows:

```
(a OR b) AND c
```

In general, the appropriate use of parentheses in query expressions, especially complex ones, ensures that the query expression is interpreted as intended.

The Verity search engine uses precedence rules to determine how operators are assigned. These rules state that some operators rank higher than others when assigned to topics, and affect how document selections are performed.

Figure 2-1 shows the precedence of the various operators. Higher levels can be parents of lower levels, but the reverse is not true.

Figure 2-1 Operator Precedence

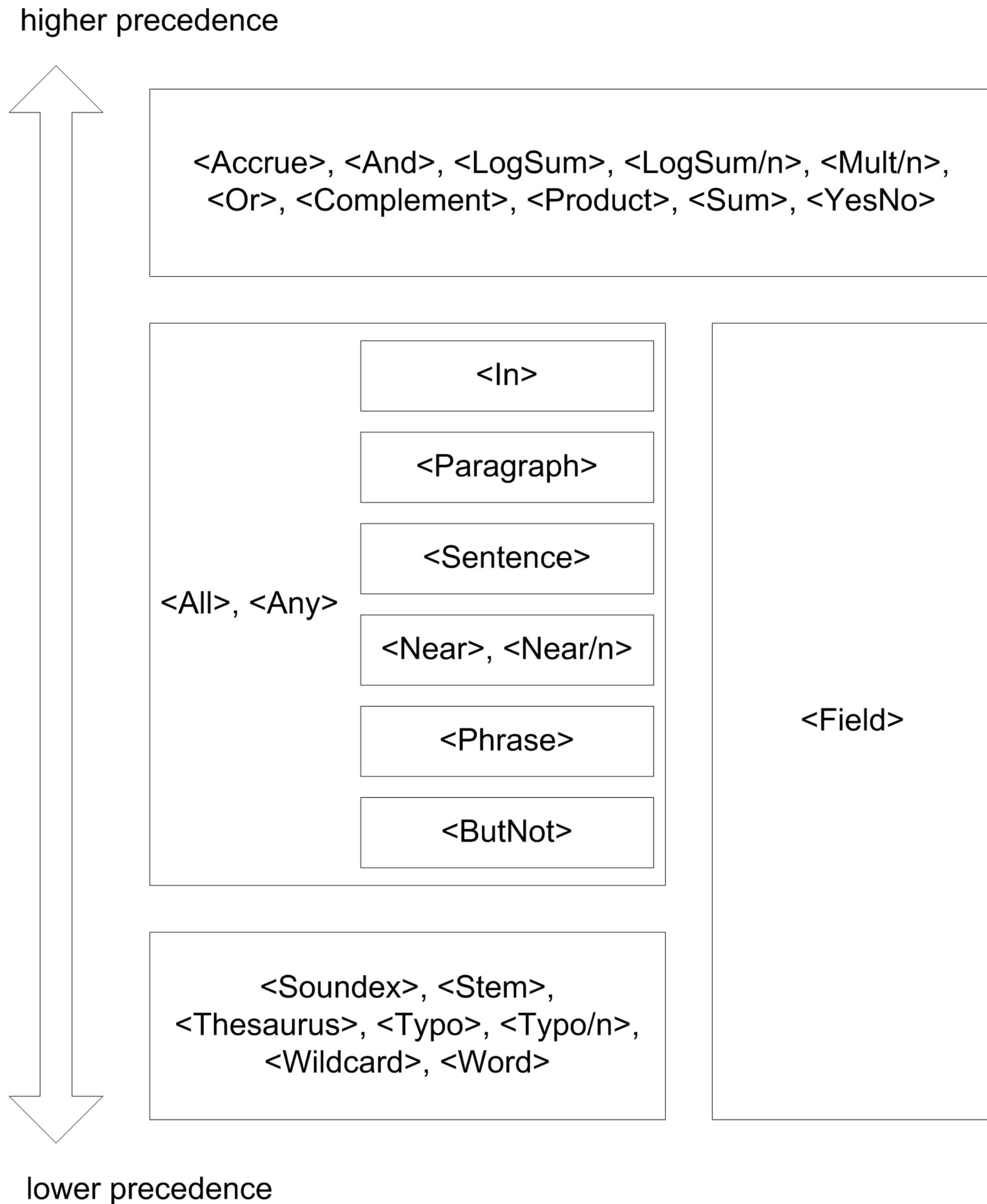


Table 2-2 Operator Precedence Rules

Operator	Precedence	How Precedence is Determined
AND OR ACCRUE	Highest precedence	These concept operators take the highest precedence over the other operators. So, subtopics of topics using these operators can be assigned any of the operators listed below under "incremental precedence" or "lowest precedence."
ALL IN PARAGRAPH SENTENCE NEAR, NEAR/N PHRASE BUTNOT ANY	Incremental precedence (in descending order)	This combination of concept and proximity operators refer to incremental ranges that exist within a document. Subtopics of topics using these operators can be assigned their next lowest operator in the precedence order. So, a phrase takes precedence over a word; a sentence takes precedence over a phrase or a word; and a paragraph takes precedence over a sentence, a phrase, or a word.
WORD STEM SOUNDEX WILDCARD THESAURUS TYPO, TYPO/N	Lowest precedence	These evidence operators reside at the lowest level in a topic structure. Because evidence operators are used with words contained in documents, these operators all have the same precedence.

Note To avoid a precedence violation, do not use ANY or ALL in a parent topic whose child topic includes a concept operator (AND, OR, ACCRUE). Topics that use ANY or ALL cannot have variable weights assigned to them, so you cannot use these operators in a parent topic with any child topic that allows variable weights (such as AND, OR, ACCRUE).

Parentheses in Expressions

Parentheses indicate the order in which the directions are to be performed; information within parentheses is read first, then information outside parentheses is read next. There must be at least one space between operators and words used in the expression. The following example means: Look for documents that contain *a* and *b*, or documents that contain *c*.

```
(a AND b) OR c
```

When there are nested parentheses, start with the innermost level. The following example means: Look for documents that contain *b* or *c* as well as *a*, or that contain *d*.

```
(a AND (b OR c)) OR d
```

Prefix and Infix Notation

Words or topics that use any operator except evidence operators (`SOUNDEX`, `STEM`, `THESAURUS`, `WILDCARD`, and `WORD`) can be defined in prefix notation or in infix notation.

Prefix notation is a format that specifies that the operator is specified before the words or topics used with that operator. The following example means: Look for documents that contain *a* and *b*.

```
AND (a,b)
```

When prefix notation is used, precedence is explicit within the expression. The following example means: Look for documents that contain *b* and *c* first, then documents that contain *a*.

```
OR (a, AND (b,c))
```

Infix notation is a format that specifies that the operator is specified between each element within the expression. The following example means: Look for documents that contain *a* and *b* or documents that contain *c*.

```
a AND b OR c
```

When infix notation is used, precedence is implicit within the expression; for example, the `AND` operator takes precedence over the `OR` operator.

Delimiters in Expressions

Angle brackets (< >), braces ({ }), double-quotation marks ("), and backslashes (\) are used in expressions as described in the following sections.

Angle Brackets for Operators

Left and right angle brackets (< >) are reserved for designating operators and modifiers. They are optional for AND, OR, and NOT but required in all other cases. Examples in this guide appear with and without angle brackets. As the following simple syntax examples show, enter expressions either way:

```
future <AND> trends
future AND trends
```

Both expressions mean: Look for documents that contain the stemmed variations of the words “future” and “trends”.

You can also explicitly specify a topic by using <TOPIC>(*topic_name*), where *topic_name* represents the topic to be used. The following example means: Look for documents that contain elements of the topic named performing-arts and the stemmed variations of the word “acting.”

```
<TOPIC>(performing-arts) AND acting
```

Braces in Expressions

Use left and right braces ({ }) to specify a topic. The following example means: Look for documents that contain elements of the topics named philosophy and history.

```
{philosophy} AND {history}
```

Double Quotes for Reserved Words

To search for a word that is reserved as an operator (AND, OR, and NOT), enclose the word in double quotation marks. For example, to search for the phrase “black and white TV,” enter the following simple syntax:

```
black "and" white TV
```

Enclosing the word “and” in double-quotation marks (“) signifies that “and” should be considered as a word, not an operator.

Backslashes for Special Characters

To include a backslash (\) in a search, insert two backslashes for each backslash character. To search for “C:\bin\print,” enter the following simple syntax:

```
C:\\bin\\print
```

Special Characters

The following information describes how special characters are interpreted.

Characters with Special Meaning

Characters without special meaning in the Verity query language can be entered anywhere in a query. Characters with special meaning are shown in Table 2-3.

Table 2-3 Special Characters

Special Characters	Description
, ( ) [	These characters end a text token.
= > < !	These characters end a text token because they signify the start of a field operator. (! is special: != ends a token.)
' @ ` < { [!	These characters signify the start of a delimited token. These are terminated by the end character associated with the start character.

A backslash removes special meaning from the next character. To enter a literal backslash in a query, use two backslashes. The following examples illustrate the use of the backslash.

```
<FreeText>("\"Hello\", said Emilie.")
'Emilie\'s'
"phrase containing a backslash (\\)"
```

Punctuation in Queries

Punctuation in queries is handled by an automatic expansion mechanism, in which, for example, the string “AT&T” becomes the following:

```
<Any>("AT&T", "AT T", "AT & T")
```

Qualify Instance Queries

The qualify instance feature can be implemented through the `VdkCollectionQualifyCBFnc` call, as described in *Verity Developer’s Kit Programming Reference*. When implemented, users can search for instance data. The qualify instance feature can be implemented using the Verity Developer’s Kit product only.

To search for the word “steve” with an instance value of 52, you enter the following query:

```
steve[52]
```

If a particular search term is to be “qualified,” then the qualification follows the search term in square brackets (`[]`).

When entering a search string, to qualify an individual word or set of words in an expansion list, append the qualification to the leaf in square brackets. For example:

```
 <WORD>apple[67] <AND> <WORD>banana[44]
```

In the previous example, the only documents that would pass this query would be those that had the word “apple” with an instance value of 67 and the word “banana” with an instance value of 44.

Any valid leaf term can be qualified, except a leaf using the `TYPO` or `TYPO/N` operator. Examples:

```
<STEM>orange[45]
<SOUNDEX>kiwi[50]
```

To search for instance data using weights, you must use parentheses surrounding the qualify instance part of the query. For example, the following queries will be processed:

```
[80](orange)[45])
```

The weight of 80 will be applied to the qualify instance leaf: orange[45].

3

Operators

This chapter describes Verity query language operators. These sections are included:

- Operators for Searching Full Text
- Operators for Searching Text Fields
- Operators for Searching Numeric Fields

Operators for Searching Full Text

This section describes operators used for performing full text searches. The following three tables summarize the three "families" of text search operators. The operators and examples of their use are listed in alphabetical order after the tables.

Table 3-1 Evidence Operators

Operator	Modifiers	Automatically Relevance-ranked
SOUNDEX	MANY, NOT	No
STEM	MANY, NOT	No
THESAURUS	MANY, NOT	No
TYPO/N	CASE, MANY, NOT	No
WILDCARD	CASE, MANY, NOT	No
WORD	CASE, MANY, NOT	No

Table 3-2 Proximity Operators

Operator	Modifiers	Automatically Relevance-ranked
ALL	MANY, NOT, ORDER	No
ANY	MANY, NOT	No
BUTNOT	MANY, NOT	No
IN	MANY, NOT, WHEN	Inherits from the subquery.
NEAR	NOT, ORDER	Yes
NEAR/n	NOT, ORDER	Yes
PARAGRAPH	MANY, NOT, ORDER	No
PHRASE	MANY, NOT	No
SENTENCE	MANY, NOT, ORDER	No

Table 3-3 Concept Operators

Operator	Modifiers	Automatically Relevance-ranked
ACCRUE	NOT	Yes
AND	NOT	Yes
OR	NOT	Yes

ACCRUE

`ACCRUE` selects documents that include at least one of the search elements you specify. Valid search elements are two or more words or phrases. Retrieved documents are relevance-ranked.

The `ACCRUE` operator scores retrieved documents according to the presence of each search element in the document using "the more, the better" approach; the more search elements found in the document, the better the document's score.

The following examples illustrate the search syntax. For example, to select documents containing stemmed variations of the words "computers" and "laptops," enter any of the following:

```
computers <ACCRUE> laptops
```

```
computers, laptops
```

```
<ACCRUE> (computers, laptops)
```

The following examples show how you can use the ACCRUE operator in topics.

```
topic_1 <Accrue>
  <Word> computer
  <Word> speed

topic_2 <Accrue>
  p1 <Near>
    <Thesaurus> retrieve
    <Thesaurus> information
  p2 <Phrase>
    <Stem> view
    <Word> and
    <Stem> print
```

Note If you use Intelligent Classifier to create a child node under an ACCRUE operator, the child node is automatically assigned the default weight of 0.50. This allows ACCRUE to differentiate documents with more hits from those with fewer. If all of the children of ACCRUE have weights of 1.00, most documents will have equal scores, regardless of how many of the children's search terms are present within the documents.

For the best selection results, assign weights between 0.80 and 0.20 to the children of ACCRUE.

ALL

Selects documents that contain all of your search elements. Retrieved documents are not relevance-ranked. Scores cannot be assigned to this operator.

For example, to select documents that contain stemmed variations of the phrase "pharmaceutical companies" and stemmed variations of the word "stock," enter the following:

```
pharmaceutical companies <ALL> stock
```

Only those documents that contain both search elements, or stemmed variations of them (for example, "pharmaceutical company," "stocks," and so on), are retrieved. Each retrieved document is assigned a score of 1.00.

The following example retrieves documents that contain “Verity” and “product” and “press release.”

```
example <All>
  <Case><Word> Verity
  <Word> product
  <Phrase>
     <Word> press
     <Word> release
```

AND

Selects documents that contain all of your search elements. Documents retrieved using the `AND` operator are relevance-ranked.

For example, to select documents that contain stemmed variations of the phrase “pharmaceutical companies” and stemmed variations of the word “stock,” enter the following:

```
pharmaceutical companies AND stock
```

Only those documents that contain both search elements, or stemmed variations of them (for example, “pharmaceutical company,” “stocks,” and so on), are retrieved. A calculated score is assigned to each retrieved document.

The unweighted score is the *lowest* score of the child nodes. In particular, if any of the child nodes return a score of 0, `AND` also scores 0. For example, if the lowest score of the children is 0.5739, `AND` returns a score of 0.5739.

ANY

Selects documents that contain at least one of your search elements. Retrieved documents are not relevance-ranked. Scores cannot be assigned to this operator.

For example, to select documents that contain stemmed variations of the word “election” or the phrases “national elections” or “senatorial race”, enter the following:

```
election <ANY> national elections <ANY> senatorial race
```

Only those documents that contain at least one of the search elements, or a stemmed variation of at least one of them, are retrieved. Each retrieved document is assigned a score of 1.00.

BUTNOT

Selects documents that qualify your search term (word or phrase) by specifying one or more additional terms that cannot match the search term to count as a hit. For example, the query:

```
<BUTNOT>(china, china sea, north china, south china)
```

matches the text "fine china is beautiful" and the text "fine china shipped from north china" but does not match the text "the china sea is very cold" or the text "the economic integration of north china and south china".

Wildcard terms are fully supported, so the following query works:

```
<BUTNOT>(chin*, china sea, *th china)
```

The `<BUTNOT>` operator is a proximity operator and has the same operator priority as `<PHRASE>`. A `<BUTNOT>` query term can be used anywhere in a query that a **`PHRASE`** or non-field leaf term (such as `WORD` or `STEM`) can appear. The `<BUTNOT>` operator accepts the `MANY` and `NOT` modifiers, and is Boolean by default.

`<BUTNOT>` terms highlight appropriately in streamed documents. In particular, instances of the first term that match a qualifying term are not highlighted.

IN

Selects documents that contain specified values in one or more document zones. A *document zone* represents a region of a document, such as the document's summary, date, or body text. The `IN` operator works only if document zones have been defined in your collections. If you use the `IN` operator to search collections without defined zones, no documents will be selected. Also, the zone name you specify must match the zone names defined in your collections. Consult your collection administrator to determine which zones have been defined for specific collections.

The `IN` operator can be qualified with the `WHEN` operator to search for a term only within the one or more zones upon which certain conditions have been placed. Use of the `WHEN` operator is described in “WHEN” on page 77.

The following query expression searches document zones named “summary” for the word “safety.”

```
"safety" <IN> summary
```

To search with multiple words, phrases, or topics, enclose them in parentheses. The following query expression searches document zones named “summary” for the word “safety” and stemmed variations of the word “warning.”

```
("safety", warning) <IN> summary
```

To search multiple zones, separate them with commas and enclose them in parentheses. The following query expression searches both the “summary” zone and the “title” zone for the word “safety” and stemmed variations of the word “warning.”

```
("safety", warning) <IN> (summary, title)
```

You must enclose query expressions containing commas in parentheses. The following example searches the “summary” zone for the word “safety” and stemmed variations of the phrase “environmental regulation.”

```
("safety", environmental regulation) <IN> summary
```

The following query expression searches both the “summary” zone and the “title” zone for the word “safety” and stemmed variations of the phrase “environmental regulation.”

```
("safety", environmental regulation) <IN> (summary, title)
```

The following topic example selects documents containing the word “new” in IMG zones whose SRC attribute contains “logo”.

```
<In> IMG <When> SRC <Contains> logo
    <Word> new
```

This matches, for example, an HTML document containing the following line (assuming IMG is a zone defined for the collection).

```
<IMG SRC="new logo.gif">.
```

Note You can enter the node as shown in the previous example, without parentheses if you are using Intelligent Classifier. Parentheses are automatically added around parts of the expression when the node is parsed.

The following topic example selects documents containing the phrase “located here” in A zones with an HREF attribute that contains “verity”.

```
<In> A <When> HREF <Contains> verity
    <Phrase>
        <Word> located
        <Word> here
```

This matches, for example, an HTML document containing the following line (assuming A is a zone defined for the collection).

```
Our site is <A HREF="www.verity.com">located here</A>.
```

The following topic example shows how `<IN>` nodes can be nested.

```
<In> BODY
    <In> A <When> HREF <Contains> verity
        <Word> documents
```

Note In Intelligent Classifier, you can use the Assists window to see what zones have been defined for a given collection. For more information about defining zones in collections, see the *Verity K2 Dashboard Administrator Guide*.

XML Support

You can use the `<IN>` operator for structural searches in a supported XPath subset. Use an XPath to identify zones within an XML document.

That means that the syntax:

```
query <in> (zone1, …, zonen)
```

is now interpreted as

```
query <in> (xpath1, …, xpathn)
```

Note Absolute and relative XPaths, as well as simple zone names, are supported. The relative XPaths are handled from the document root.

For example, to find documents where the first book is about UNIX.

```
VQL: unix <in> //book[1]
```

Table 3-4 Supported XPath Subset

XPath Construct	XML Fragment Example	<IN> operator
Child abbreviation (/)	Books under the bib root element.	/bib/book
Descendant-or-self abbreviation (//)	Books anywhere in the document.	//book
Subscript([n])	1st book under the bib root element.	/bib/book[1]
Attribute (@)	Published books.	//book[@published]

NEAR

Selects documents containing specified search terms within close proximity to each other. Document scores are calculated based on the relative number of words between search terms.

For example, if the search expression includes two words, and those words occur next to each other in a document (so that the region size is two words long), then the score assigned to that document is 1.0. Thus, the document with the smallest possible region containing all search terms always receives the highest score. As search terms appear further apart, the score drops toward zero. A document receives a zero score only if it does not contain all search terms.

The `NEAR` operator is similar to the other proximity operators in the sense that the search words you enter must be found within close proximity of one another. However, unlike other proximity operators, the `NEAR` operator calculates relative proximity and assigns scores based on its calculations.

To retrieve relevance-ranked documents that contain stemmed variations of the words “war” and “peace” within close proximity to each other, enter the following:

```
war <NEAR> peace
```

The following topic examples show how you can use `<NEAR>` with various operators.

```
example <Near>
    <Word> document
    <Word> retrieve

example2 <Near>
    <Phrase>
        <Stem> document
        <Word> retrieval
    <Case><Word> Verity
    <Wildcard> computer*
    <Typo> keyview
    <Any>
        <Word> new
        <Word> announce
```

If the PSW option is used when the collection is built, `<NEAR>` will not cross sentence or paragraph boundaries. That is, it only returns documents where the search terms are within *n* words and are in the same sentence and paragraph.

NEAR/n

Selects documents containing two or more words within *n* number of words of each other, where *n* is an integer. Document scores are calculated based on the relative distance of the specified words when they are separated by *n* words or less.

For example, if the search expression `NEAR/5` is used to find two words within five words of each other, a document that has the specified words within three words of each other is scored higher than a document that has the specified words within five words of each other.

The *n* variable can be an integer between 1 and 1,024, where `NEAR/1` searches for two words that are next to each other. If *n* is 1,000 or above, you must specify its value without commas, as in `NEAR/1000`. You can specify multiple search terms using multiple instances of `NEAR/`*n*, as long as the value of *n* is the same.

Note The `NEAR/`*n* operator default is 1024.

For example, to retrieve relevance-ranked documents that contain stemmed variations of the words “commute,” “bicycle,” “train,” and “bus” within 10 words of each other, enter the following:

```
commute <NEAR/10> bicycle <NEAR/10> train <NEAR/10> bus
```

You can use the `NEAR/`*n* operator with the `ORDER` modifier to perform ordered proximity searches. For more information about the `ORDER` modifier, see “ORDER” on page 76.

If the PSW option is used when the collection is built, `<NEAR/n>` will not cross sentence or paragraph boundaries. That is, it only returns documents where the search terms are within *n* words and are in the same sentence and paragraph.

OR

`OR` selects documents that show evidence of at least one of your search elements. Documents selected using the `OR` operator are relevance-ranked.

For example, to select documents that contain stemmed variations of the word “election” or the phrases “national elections” or “senatorial race,” enter the following:

```
election OR national elections OR senatorial race
```

Only those documents that contain at least one of the search elements, or a stemmed variation of at least one of them, are retrieved. A calculated score is assigned to each retrieved document.

The unweighted score is the highest score of the child nodes. If any of the child nodes return a score of 1, the OR operator's unweighted score is also 1. If the highest scoring subtopic under the OR operator has a score of 0.8442, the OR operator returns 0.8442.

PARAGRAPH

The PARAGRAPH operator selects documents that include all of the search elements you specify within a paragraph. Valid search elements are two or more words or phrases. You can specify search elements in a sequential or a random order. Documents are retrieved as long as search elements appear in the same paragraph.

To retrieve relevance-ranked documents that contain stemmed variations of the word "drug" and the phrase "cancer treating" in the same paragraph, enter the following:

```
drug <PARAGRAPH> cancer treating
```

To search for three or more words or phrases, you must use the PARAGRAPH operator between each word or phrase.

In the following topic example, PARAGRAPH selects documents that contain (1) a sentence containing "computer" and "laptop", and (2) the word "software", both in the same paragraph.

```
example <Paragraph>
    <Sentence>
        <Word> computer
        <Word> laptop
    <Word> software
```

You can use the PARAGRAPH operator with the ORDER modifier to perform ordered proximity searches. For more information about the ORDER modifier, see "ORDER" on page 76.

PARAGRAPH only has the expected behavior if the PSW option is used when the collection is built. If this option is not used, PARAGRAPH matches documents if the search terms occur within a certain distance of each other, whether or not the search terms occur in the same paragraph.

PHRASE

Selects documents that include a phrase you specify. A phrase is a grouping of two or more words that occur next to each other in a specific order.

By default, two or more words separated by a space are considered to be a phrase in simple syntax. Two or more words enclosed in double quotes are also considered to be a phrase. To retrieve relevance-ranked documents that contain the phrase “mission oak,” enter any of the following:

```
mission oak
```

```
"mission oak"
```

```
mission <PHRASE> oak
```

```
<PHRASE> (mission, oak)
```

When entering a new node in the topic tree, you can create a `PHRASE` set of nodes by typing the phrase inside double quotation marks double quotation marks ("). You can also enter a phrase inside single quotation marks as a shortcut. For example:

Entering "`my dog barks`"	produces	`<Many><Phrase>`
		`<Word> my`
		`<Word> dog`
		`<Word> barks`

Entering `'my dog barks'`	produces	`<Many><Phrase>`
		`<Stem> my`
		`<Stem> dog`
		`<Stem> barks`

`PHRASE` has an unweighted score of 1 if the search is successful, and 0 otherwise. Scores of matching documents can be relevance-ranked (range from 0.01 to 1) using the `MANY` modifier.

SENTENCE

Selects documents that include all of the words you specify within a sentence. You can specify search elements in a sequential or a random order. Documents are retrieved as long as search elements appear in the same sentence.

To retrieve relevance-ranked documents that contain stemmed variations of the words "American," and "innovation" within the same sentence, enter the following:

```
american <SENTENCE> innovation
```

```
<SENTENCE> (american, innovation)
```

You can use the SENTENCE operator with the ORDER modifier to perform ordered proximity searches. For more information about the ORDER modifier, see "ORDER" on page 76.

SENTENCE has an unweighted score of 1 if the search is successful, and 0 otherwise. Scores of matching documents can be relevance-ranked (modified to range from 0.01 to 1) by using the MANY modifier.

SENTENCE only has the expected behavior if the PSW option is used when the collection is built. If this option is not used, SENTENCE matches documents if the search terms occur within a certain distance of each other, whether or not the search terms occur in the same sentence. It is also affected by the EOS (end of sentence) setting in the `style.lex` file.

SOUNDEX

Selects documents that include one or more words that sound like, or whose letter pattern is similar to, the word specified. Words must start with the same letter as the word you specify to be selected.

Your collection administrator must have configured collections to support the SOUNDEX operator. See your collection administrator for information.

For example, to retrieve documents containing a word that is close in structure to the word "sale," enter the following:

```
<SOUNDEX> sale
```

The documents retrieved will include words such as "sale," "sell," "seal," "shell," "soul," and "scale." Documents are not relevance-ranked unless the MANY modifier is used, as in:

```
<MANY><SOUNDEX> sale
```

Note In Intelligent Classifier, you can use the Assists window to see what SOUNDEX will return for a given collection.

STEM

STEM selects documents that include one or more variations of the search word you specify.

For example, to retrieve documents containing a variation of the word "film," enter the following:

```
<STEM> film
```

The documents retrieved will include words such as "films," "filmed," and "filming." Documents are not relevance-ranked unless the MANY modifier is used, as in:

```
<MANY><STEM> film
```

When entering a new node in the topic tree, you can create a simple STEM node by enclosing the search term in single quotation marks ('). You can create a phrase of STEM nodes by entering the phrase in single quotation marks.

Note In Intelligent Classifier, you can use the Assists window to see what STEM will return for a given collection.

THESAURUS

Selects documents that contain one or more synonyms of the word you specify.

For example, to retrieve documents containing synonyms of the word "altitude," enter the following:

```
<THESAURUS> altitude
```

The documents retrieved might include words such as "height" or "elevation." Documents are not relevance-ranked unless the MANY modifier is used, as in:

```
<MANY><THESAURUS> altitude
```

Note In Intelligent Classifier, you can use the Assists window to see what THESAURUS will return for a given collection.

The synonyms for a given word are defined in a thesaurus file. Thesaurus files are locale-specific, and Verity provides default thesaurus files for many locales. You can create a customized thesaurus file for a locale or to support an industry-specific terminology; see"Creating a Custom Thesaurus" on page 151 for more information.

TYPO/N

Selects documents that contain the word you specify plus words that are similar to the query term. The `TYPO/`*N* operator performs "approximate pattern matching" to identify similar words. This makes it ideal for use in an environment where documents have been scanned using an Optical Character Reader (OCR).

The optional *N* variable in the operator name expresses the maximum number of errors between the query term and a matched term, a value called the error distance. If *N* is not specified, an error distance of 2 is used.

The error distance between two words is based on the calculation of errors, where an error is defined to be a character insertion, deletion, or transposition. For example, for these sets of words, the second word matches the first within an error distance of 1:

```
mouse, house (m→h)
agreed, greed (a is deleted)
cat, coat (o is inserted)
```

For the following query, documents with the words "sweeping" and "swimming" will match, because there are 3 transpositions in the word (e→i, e→m, p→m).

```
<TYPO/3> sweeping
```

Both of the following queries return the same results. Documents containing the words "swept" and "kept" match, because the "kept" word contains 1 transposition, 1 deletion.

```
<TYPO/2> swept
```

```
<TYPO> swept
```

The `TYPO/`*N* operator must scan the collection's word list to find candidate matching words. This makes it impractical for use in large collections (greater than 100,000 documents unless a current spanning word list is available) or in performance-sensitive environments. Performance can be improved by generating a spanning word list for the collections to be used. For more information on generating spanning word lists, see the *Verity Collection Reference* for information about collection optimization.

Note Please note these limitations. A query term specified with `TYPO/`*N* can have a maximum length of 32 characters. Also, `TYPO/`*N* is not supported with multibyte character sets.

WILDCARD

Selects documents that contain matches to a wildcard character string. The `WILDCARD` operator lets you define a wildcard string, which can be used to locate related word matches in documents. A wildcard string consists of special characters.

For example, to retrieve documents that contain words such as, "pharmaceutical," "pharmacology," and "pharmacodynamics," enter the following:

```
pharmac*
```

Documents are not relevance-ranked unless the `MANY` modifier is used, as in:

```
<MANY> pharmac*
```

The wildcard characters "*" and "?" automatically enable wildcard searching. To use other constructs, use the `WILDCARD` operator explicitly with any of the characters in Table 3-5. By default, searches are case-insensitive. You can change this using the `CASE` modifier.

Table 3-5 Wildcard Characters

Character	Function
`?`	Specifies one of any alphanumeric character, as in `?an`, which locates "ran," "pan," "can," and "ban." It is not necessary to specify the `WILDCARD` operator when you use the question mark. The question mark is ignored in a set (`[ ]`) or in an alternative pattern (`{ }`).
`*`	Specifies zero or more of any alphanumeric character, as in `corp*`, which locates "corporate," "corporation," "corporal," and "corpulent." It is not necessary to specify the `WILDCARD` operator when you use the asterisk. Do not use the asterisk to specify the first character of a wildcard string. The asterisk is ignored in a set (`[ ]`) or in an alternative pattern (`{ }`).
`[ ]`	Specifies one of any character in a set, as in `<WILDCARD> 'c[auo]t'`, which locates "cat," "cut," and "cot." You must enclose the word that includes a set in backquotes (`'`), and a set cannot contain spaces.
`{ }`	Specifies one of each pattern separated by commas, as in `<WILDCARD> 'bank{s,er,ing}'`, which locates "banks," "banker," and "banking." You must enclose the word that includes a pattern in backquotes (`'`), and a set cannot contain spaces.
`^`	Specifies one of any character not in the set, as in `<WILDCARD> 'st[^oa]ck'`, which excludes "stock" and "stack" but locates "stick" and "stuck." The caret (^) must be the first character after the left bracket (`[`) that introduces a set.
`-`	Specifies a range of characters in a set, as in `<WILDCARD> 'c[a-r]t'`, which locates every three-letter word from "cat" to "crt."

Note For Chinese, Japanese, and Korean, a wildcard search can be manually modified to specify how to tokenize the query string. For example, you can search for:
`B CD*` instead of `BCD*`

Note In Intelligent Classifier, you can use the Assists window to see what `WILDCARD` will return for a given collection.

Searching for Nonalphanumeric Characters

Remember that you can search for nonalphanumeric characters only if the `style.lex` file used to create the collections you are searching is configured to recognize the characters you want to locate. Consult your collection administrator for more information.

Searching for Wildcard Characters as Literals

Provided the `style.lex` file is configured for the collections to be searched, you can search for a word containing a wildcard character such as "/" or "*" by preceding the wildcard character with a backslash.

For example, when you enter the following search string:

```
abc\*d
```

the engine finds five-character words matching the "abc*d" string.

When you want to match a literal backslash, you must enter two backslashes.

Searching for Special Characters as Literals

The following nonalphanumeric characters perform special, internal search engine functions, and, by default, are not treated as literals in a wildcard string:

- comma ,
- left and right parentheses ( )
- double quotation mark "

- backslash \
- at sign @
- left curly brace {
- left bracket [
- less than sign <
- backquote `

To interpret special characters as literals, you must surround the whole wildcard string in backquotes (`). For example, to search for the wildcard string “a{b”, you surround the string with backquotes, as follows:

```
<WILDCARD> `a{b`
```

To search for a wildcard string that includes the literal backquote character (`), you must use two backquotes together and surround the whole wildcard string in backquotes (`). For example, to search for the wildcard string “*n‘t”, you can enter the following query:

```
<WILDCARD> `*n``t`
```

You can search on backquotes only if the `style.lex` file used to create the collections you are searching is configured to recognize the backquote character. Consult your collection administrator for information.

WORD

`WORD` selects documents that include one or more instances of only the word you specify without locating stemmed variations of that word.

For example, to search for documents that contain the word “rhetoric,” without also considering the words “rhetorical” and “rhetorician,” enter the following:

```
<WORD> rhetoric
```

Documents are not relevance-ranked unless the `MANY` modifier is used, as in:

```
<MANY><WORD> rhetoric
```

In Intelligent Classifier, if you enter a new node in the topic tree, you can create a simple `<MANY><WORD>` node by enclosing the search term in double quotation marks (“).

Create a phrase of `WORD` nodes by entering the phrase in double quotation marks.

The style.lex file affects how documents are parsed into words when a collection is built. For example, it affects whether “plug-in” is considered to be one word or two. It also affects whether you can search for non-alphanumeric characters, such as in the term “OS/2”.

Operators for Searching Text Fields

This section describes operators that can be used to search document fields.

CONTAINS

Selects documents by matching the word or phrase you specify with values stored in a specific document field. Documents are selected only if the search elements you specify appear in the same sequential and contiguous order in the field value. When you use the `CONTAINS` operator, you specify the field name to search, and the word or phrase to locate.

With the `CONTAINS` operator, the words stored in a document field are interpreted as individual, sequential units. You can specify one or more of these units as search criteria. To specify multiple words, each word must be sequential and contiguous and must be separated by a blank space.

For example, the following title contains eight sequential words:

```
American Version of 'Orient Express' Offers Opulent Ride
```

The following examples demonstrate how you can use the `CONTAINS` operator with sequential, contiguous words to match the previous document title, assuming it is stored in a title field:

```
TITLE <CONTAINS> American Version
```

```
TITLE <CONTAINS> Express Offers
```

The following examples show how you can use a question mark (?) to represent individual variable characters of a word and an asterisk (*) to match multiple variable characters of a word:

```
TITLE <CONTAINS> Amer* Version
```

```
TITLE <CONTAINS> Version of Or????
```

Question marks and asterisks cannot be used to represent white space that appears between words.

The CONTAINS operator does not recognize nonalphanumeric characters. The CONTAINS operator interprets nonalphanumeric characters as spaces and treats the separated values as individual units.

For example, if you have defined a dash (-) as a valid character, and you enter search criteria that include this character, as in on-line, the value is defined as two individual units, as follows:

```
TITLE <CONTAINS> on line
```

ENDS

Selects documents by matching the character string you specify with the ending characters of the values stored in a specific document field.

For example, assume a document field named AUTHOR has been defined. To select documents written by Milner, Wagner, and Faulkner, enter the following:

```
AUTHOR <ENDS> ner
```

MATCHES

Selects documents by matching the character string you specify with values stored in a specific document field. Documents are selected only if the search elements specified match the field value exactly. If a partial match is found, a document is not selected. When you use the MATCHES operator, you specify the field name to search, and the word, phrase, or number to locate.

You can use question marks (?) to represent individual variable characters within a string, and asterisks (*) to match multiple characters within a string.

For example, assume a document field named SOURCE includes the following values:

```
COMPUTER

COMPUTERWORLD

COMPUTER CURRENTS

PC COMPUTING
```

To locate documents whose source is COMPUTER, the MATCHES operator is used as follows:

```
SOURCE <MATCHES> computer
```

Here, the MATCHES operator matches COMPUTER, but not COMPUTERWORLD, COMPUTER CURRENTS, or PC COMPUTING.

To locate documents whose source is COMPUTERWORLD, the MATCHES operator is used as follows:

```
SOURCE <MATCHES> computer?????
```

Now, the MATCHES operator matches COMPUTERWORLD, because each question mark (?) represents specific character positions within the string. COMPUTER and COMPUTER CURRENTS are not matched, because their character strings do not match the length represented by the question marks.

To locate documents whose sources are COMPUTER, COMPUTERWORLD, and COMPUTER CURRENTS, the MATCHES operator is used as follows:

```
SOURCE <MATCHES> computer*
```

Here, the MATCHES operator matches COMPUTER, COMPUTERWORLD, and COMPUTER CURRENTS, because the asterisk (*) represents zero or more variable characters at the end of the string.

To locate documents whose sources include COMPUTER, COMPUTERWORLD, COMPUTER CURRENTS, and PC COMPUTING, the MATCHES operator can be used as follows:

```
SOURCE <MATCHES> *comput*
```

Now, the MATCHES operator matches all four occurrences, because the asterisk (*) represents a string of characters of any length.

STARTS

Selects documents by matching the character string you specify with the starting characters of the values stored in a specific document field.

For example, assume a document field named REPORTER has been defined. To retrieve documents written by Jack, Jackson, and Jacks, enter the following:

```
REPORTER <STARTS> jack
```

SUBSTRING

Selects documents by matching the character string you specify with a portion of the strings of the values stored in a specific document field. The characters that comprise the string can occur at the beginning of a field value, within a field value, or at the end of a field value.

For example, assume a document field named `TITLE` has been defined. To retrieve documents whose titles contain words such as "solution," "resolution," "solve," and "resolve," enter the following:

```
TITLE <SUBSTRING> sol
```

Operators for Searching Numeric Fields

The following sections describe operators used to search numeric and date fields.

= (Equals)

Selects documents whose document field values are exactly the same as the search string you specify.

For example, assume a document field named `ORGNO` has been defined as the number of the organization that wrote the document. To select only those documents written by organization 104, enter the following:

```
ORGNO = 104
```

!= (Not Equals)

Selects documents whose document field values do not match the search string you specify.

For example, to search for documents with ORGNO field values not equal to 104, use this query:

```
ORGNO != 104
```

Another query using the NOT modifier could be used to perform the same search:

```
<NOT>(ORGNO=104)
```

Although using the `NOT` modifier returns the same results as using the `!=` operator, the `!=` operator functions much more efficiently.

> (Greater Than)

Selects documents whose document field values are greater than the search string you specify.

For example, assume a document field named `REVISION` has been defined. To select only those documents that have been revised more than three times, enter the following:

```
REVISION > 3
```

>= (Greater Than Or Equal To)

Selects documents whose document field values are greater than or equal to the search string you specify.

For example, assume a document field named `REVISION` has been defined. To select only those documents that have been revised three times or more, enter the following:

```
REVISION >= 3
```

< (Less Than)

Selects documents whose document field values are less than the search string you specify.

For example, assume a document field named `PAGES` has been defined. To select only those documents less than five pages long, enter the following:

```
PAGES < 5
```

<= (Less Than Or Equal To)

Selects documents whose document field values are less than or equal to the search string you specify.

For example, assume a document field named `DATE` has been defined. To select only those documents dated prior to and including February 14, 1991, enter the following:

```
DATE <= 02-14-91
```

To refine a date search using the time of day, use a 24-hour format. For example,

```
DATE <= 02-14-2003 13:00
```

You can also use the `DATE` field with `today` and `now`. For example,

```
DATE <= today
```

4

Modifiers

This chapter describes Verity query language modifiers. These modifiers can be combined with operators to compose a query expression:

- CASE
- LANG/ID
- MANY
- NOT
- ORDER
- WHEN

CASE

Use the CASE modifier with the WORD or WILDCARD operator to perform a case-sensitive search, based on the case of the word or phrase specified. The CASE modifier is not valid with the SOUNDEX and STEM operators.

To use the CASE modifier, you simply enter the search word or phrase as you wish it to appear in retrieved documents—in all uppercase letters, in mixed uppercase and lowercase letters, or in all lowercase letters.

For example, to retrieve documents that contain the word “iMac” in mixed uppercase and lowercase letters, you would enter:

```
<CASE> <WORD> iMac
```

Only those documents that contain the word "iMac" will be selected. Occurrences of "imac," "Imac," or "IMAC" are not selected.

An example including a CASE modifier with WILDCARD is:

```
<Case><Wildcard> Computer*
```

This example retrieves documents containing "Computer," "Computers," "Computerworld," and so on, but not "COMPUTER," "COMPUTERS," "computer," or "computers."

Other examples of using `CASE` are:

```
<Case><Typo> Verity
```

```
<Many><Case><Word> release
```

LANG/ID

Use the LANG/ID modifier to perform language-specific stemmed search on collections created with the multilanguage locale.

Every collection is created within the context of a locale. For a collection created in any locale other than the multilanguage locale, all text is tokenized and stemmed according to the rules of a single language. For collections created in the multilanguage locale, however, text is tokenized and stemmed according to its document's language. Such a collection, therefore, can have a word index containing words and word stems from many different languages.

To restrict a stemmed search or topic definition over such a collection to consider only the words of a single language, apply the LANG/ID modifier to the query expression, like this:

```
<LANG/fr>un
```

In this example, only documents containing French words whose stem is **un** (such as **un**, **une**, **unes**) would be returned. If the collection also contained Spanish documents containing **uno** or **unas**, those documents would not be returned.

The simple query parser by default assumes a stemmed search. For other parsers, it may be necessary to include the `<STEM>` operator to get a language-specific search:

```
<LANG/fr><STEM>un
```

The language ID used in this modifier must be one of the language codes listed in the *Verity Locale Configuration Guide*.

Note For Chinese, Japanese, and Korean, the LANG/ID modifier also has the effect of forcing language-specific tokenization of the query string itself. For example, during indexing, a particular sequence of characters might be tokenized differently in Chinese than in Japanese. If the same sequence is subsequently used as a search term along with the LANG/ID modifier, the term is tokenized appropriately for the specified language before the collection is searched.

Rules for using the `LANG/ID` modifier are provided in the following list:

- `LANG/ID` is a modifier that can be applied to individual query terms or topic definition files. If you want to apply it to a multiple term query, you must use parentheses.

  Note By default, `LANG/ID` applies only to the next query term. If there is only one `LANG/ID` modifier in a query or topic definition, `LANG/ID` behaves as a global modifier and applies to all query terms.

- If you put more than one `LANG/ID` modifier into a query or topic definition, the last one in the string is the one that is used for the search. This also affects the use of the `LANG/ID` modifier in a thesaurus control file, so that if a `THESAURUS` operator follows a `LANG/ID` modifier in the query, the `LANG/ID` modifier in the `THESAURUS` file is used for the search.
- The `LANG/ID` modifier is ignored if the collection being searched was not indexed using the multilanguage locale.
- The `LANG/ID` modifier is ignored if the language ID it specifies is not valid or is not found in the collection.
- The `LANG/ID` modifier applies only to stemmed search. For a literal search, language-specific stems are not considered.
- The `LANG/ID` modifier applies to the results of the `FREETEXT` and `LIKE` advanced operators.
- If the `LANG/ID` modifier does not appear in a search query that is applied to a multilanguage collection, the search is nevertheless language-specific as long as a default session language has been defined. The search is equivalent to including a `LANG/ID` modifier that specifies the default language.

- To ensure that a search over a multilanguage collection is *not* language-specific, apply the WORD operator to the query (or enclose it in double quotes). That modification will ensure that the search applies to all documents in all languages in the collection, regardless of whether the LANG/ID modifier appears in the query and regardless of whether there is a defined default session locale.

The multilanguage locale and details about language-specific searching and default session language are described in the *Verity Locale Configuration Guide*.

MANY

Counts the density of words, stemmed variations, or phrases in a document, and produces a relevance-ranked score for retrieved documents. The more occurrences of a word, stem, or phrase proportional to the amount of document text, the higher the score of that document when retrieved. Because the MANY modifier considers density in proportion to document text, a longer document that contains more occurrences of a word can score lower than a shorter document that contains fewer occurrences.

For example, to select documents based on the density of stemmed variations of the word "apple," you would enter:

```
<MANY> <STEM> apple
```

To select documents based on the density of the phrase "mission oak," you would enter:

```
<MANY> mission oak
```

The parent node might ignore MANY's relevance ranking if the parent uses an operator, such as ANY, that only cares whether or not the child's score is non-zero.

The behavior of MANY depends on whether "Many" is used with the IDX-CONFIG option when the collection is built. See the *Verity Collection Reference* for more details about PSW encoding.

The MANY modifier cannot be used with AND, OR, ACCRUE, or relational operators.

NOT

Use the `NOT` modifier with a word or phrase to exclude documents that show evidence of that word or phrase.

For example, to select only documents that contain the words “cat” and “mouse” but not the word “dog,” you would enter:

```
cat <AND> mouse <AND> <NOT> dog
```

To search for documents that contain the word “not,” enclose the word “not” in double-quotation marks (‘). For example, to search for the phrase “love not war,” enter any of the following queries.

```
<ORDER> love "not" war
```

```
"love not war"
```

```
<PHRASE> (love, "not", war)
```

In Intelligent Classifier, top-level topics cannot use `NOT`. In the topic tree, `NOT` only has an effect when the node it is on passes information up to its parent node. Assigning `NOT` to a topic has no effect for that topic, only on higher-level nodes. For example, if the topic tree is:

```
topic_1 <Accrue>
    topic_2 <Not><Word> open
topic_3 <Accrue>
    topic_4 <Word> open
```

then `topic_1` retrieves documents that do not contain “open” and `topic_3` retrieves documents that do. However, `topic_2` and `topic_4` both retrieve the same set of documents (those that contain “open”).

ORDER

Use the ORDER modifier to specify that search elements must occur in the same order in which they were specified in the query. If search values do not occur in the specified order in a document, the document is not selected. You can use the ORDER modifier with these operators: PARAGRAPH, SENTENCE, NEAR/*N*, and ALL.

Always place the ORDER modifier just before the operator. The following syntax examples show how you can use either simple syntax or explicit syntax to retrieve documents containing the word “president” followed by the word “washington” in the same paragraph.

Simple syntax

```
president <ORDER><PARAGRAPH> washington
```

Explicit syntax

```
<ORDER><PARAGRAPH> ("president", "washington")
```

To search for documents containing the words “diver,” “kills,” “shark” in that order within 20 words of each other, use one of the following queries:

```
diver <ORDER><NEAR/20> kills <ORDER><NEAR/20> shark
```

```
<ORDER> <NEAR/20> (diver, kills, shark)
```

You can use the NEAR/*N* operator with the ORDER modifier to duplicate the behavior of the PHRASE operator. For example, to search for documents containing the phrase “world wide web,” you can use the following syntax:

```
world <ORDER><NEAR/1> wide <ORDER><NEAR/1> web
```

To search for a word between two other words, you can use the ORDER modifier with the ALL operator, like this:

```
<ORDER><ALL>(dog, cat, squirrel)
```

The previous query searches for “cat” between “dog” and “squirrel”. Stemmed variations of the words will match the query.

The query can be extended to include subquery expressions. For example:

```
<ORDER><ALL>(dog, fat cat, squirrel)
```

The previous query searches for the phrase “fat cat” between the words “dog” and “squirrel”. Again, stemmed variations of the words are considered a match.

In a topic tree, ORDER selects documents that contain the search terms in the specified order (for example, from top to bottom).

For example,

```
example1 <Order><Near>
    <Word> today
    <Word> announced
```

selects documents that contain “today” and “announced” in that order.

The next example shows how you can use ORDER with other operators.

```
<Order><Paragraph>
    <Word> keyview
    <Word> pro
<Order><Phrase>
    <Word> computer
    <Word> file
<Order><Sentence>
    <Word> new
    <Word> release
<Many><Order><Paragraph>
    <Word> computer
    <Word> hardware
<Order><All>
    <Word> press
    <Word> release
```

WHEN

WHEN selects documents that contain specified values in one or more document zones upon which certain conditions have been placed. The following examples illustrate searching for terms within a zone upon which certain conditions have been placed.

To search for the word “here” in a zone named “A,” whose HREF attribute contains the string “verity,” the text might appear as:

```
Our site is <A HREF = "www.verity.com">here</A>.
```

To search for the word “here” in the zone “A” when the HREF contains the string “verity,” you can write the following query:

```
"here" <IN> A <WHEN> (HREF <CONTAINS> "verity")
```

A query condition for the `WHEN` operator must be enclosed in parentheses, as shown in the previous example. A query condition can include one or more Verity operators; it takes the form:

```
attribute_name <attribute_test_operator> "test_value"
```

where *`attribute_test_operator`* is one of the following operators: `<STARTS>`, `<ENDS>`, `<CONTAINS>`, `<=>`, or `<MATCHES>`. Except for =, all operators must be surrounded by angle brackets.

Attribute test operators can be combined with the combination operators `<AND>` or `<OR>`. For example, you can search for the string “IBM” in a zone named “Company,” when the attribute named “reference” is either equal to “major” or “significant” by using the following query:

```
"IBM" <IN> "Company" <WHEN> (reference = "major" <OR>
reference = "significant")
```

XML Support

The XML support for the `<WHEN>` modifier includes:

- ZONE

  Prefix the operand with the `<zone>` modifier. to use element operands.

  The following code example shows a query to find documents containing books published by Addison-Wesley.

  ```
  * <in> book <when> <zone>publisher = "Addison-Wesley"
  ```

- NUMERIC

  Provides support for the numeric operators =, !=, >, >=, <, and <=. Prefix the operand with `<numeric>` to use non-text operator semantics.

  Note All the numeric zones/attributes are stored and used as floats.

  The following code example shows a query to find documents containing books about UNIX that are cheaper than $60.

  ```
  unix <in> book <when> <zone><numeric>price < 60
  ```

- DATE

  Prefix the operand with the `<date>` modifier to use date semantics. The date operators supported are the same as the numeric operators.

  The following code example shows a query to find documents containing books about UNIX that were published in 1999 or later.

```
unix <in> book <when> <date>published > "1/1/1999"
```

  All dates are treated as `vdates` by default. You can change to `xdates` in the `style.prm` file settings.

5

Advanced Query Language

This chapter describes the advanced Verity operators that are not used with modifiers. Four of these operators enable sophisticated combinations of query components for advanced document scoring, and two provide support for natural language analysis of query text.

It includes syntax and usage information for the following operators:

- COMPLEMENT
- FREETEXT
- LIKE
- LOGSUM and LOGSUM/n
- MULT/n
- PRODUCT
- SUM
- YESNO

These operators can be combined together or combined with other Verity query language.

Score Operators

The score operators affect how the search engine calculates scores for retrieved documents. When a score operator is used, the search engine first calculates a separate score for each search element found in a document, and then performs a mathematical operation on the individual element scores to arrive at the final score for each document.

The `YESNO` operator has wide application, whereas the `PRODUCT`, `SUM`, and `COMPLEMENT` operators are intended for use mainly by application developers who want to generate queries programmatically.

COMPLEMENT

Calculates scores for documents matching a query by taking the complement (subtracting from 1) the scores for the query's search elements. To arrive at a document's score, the search engine calculates a score for each search element and takes the complement of these scores.

For example, if the node

```
<Word> computers
```

scores 0.80, then the node

```
<Complement>
  <Word> computers
```

scores 0.20.

The following example displays the search syntax:

```
<COMPLEMENT> ("computers")
```

The `COMPLEMENT` operator is a unary operator. It multiplies search elements as specified. The elements are combined, using the `ACCRUE` operator by default, to generate a single score that is then complemented. The following sample query expression contains two search elements.

```
<COMPLEMENT> ("computers","laptops")
```

In the previous example, the query is evaluated as the word "computers" accrued using the `ACCRUE` operator with the word "laptops." The `COMPLEMENT` operator is applied to the result.

LOGSUM and LOGSUM/n

LOGSUM and LOGSUM/n are score operators. They return a score that approaches 1 as the sum of the child node's score approaches 1.

The following examples assume that *c* is the sum of the scores of the child nodes.

The unweighted score of LOGSUM/*n* is:

$$\frac{1}{1 + e^{-\left(c + \frac{n}{10000}\right)}}$$

If the parameter *n* is not specified, it defaults to zero. In other words, LOGSUM is equivalent to LOGSUM/0.

As the sum of the child nodes increases, the score of LOGSUM/*n* approaches 1. The larger *n* is, the faster is the approach to 1.

Example 1

If a document contains the word "computer" but not "file", then

```
example <LogSum>
    <Word> computer
    <Word> file
```

returns

$$\frac{1}{1 + e^{-1}} = 0.7310$$

Example 2

If a document contains the word "computer" and the word "file", then

```
example <LogSum/5000>
    <Word> computer
    <Word> file
```

returns

$$\frac{1}{1 + e^{-\left(2 + \frac{5000}{10000}\right)}} = 0.9241$$

The weights of the child nodes cannot be negative or greater than 1, but you can use the MULT operator to achieve the same thing. For example:

```
example <LogSum>
    <Mult/20000>
        <Word> computer
    <Mult/-8000>
        <Word> file
```

If a document contains “computer” and “file”, then this topic will return a score of

$$\frac{1}{1+e^{-(2.0-0.8)}} = 0.7685$$

MULT/n

`MULT/`*n* is a score operator that multiplies the score returned from its child by the constant. This is the only operator that can return a negative number or a value greater than 1.

The `MULT/`*n* operator accepts one child node. If *c* is the score of the child node, then the `MULT/`*n* operator’s unweighted score is

$$\frac{n}{10000} \times c$$

If a document contains the word “computer” then the child node returns a score of 1.0, so the topic

```
example <Mult/5000>
    <Word> computer
```

returns a score of

$$\frac{5000}{10000} \times 1.0 = 0.5$$

The parameter *n* can be left out, but *n* defaults to zero, so `MULT` always results in a score of zero.

The parameter *n* can range from -100,000,000 to +100,000,000. So the score of `MULT/`*n* can range from -10,000 times *c* to +10,000 times *c*. The score for this operator can be negative or greater than 1. For an example of how you can use `MULT`, see “LOGSUM and LOGSUM/n” on page 83.

PRODUCT

Calculates scores for documents matching a query by multiplying the scores for the query's search elements together. To arrive at a document's score, the search engine calculates a score for each search element and multiplies these scores together.

Following is an example of search syntax:

```
<PRODUCT> ("computers","laptops")
```

If a search on "computers" generated a score of .5 and a search on "laptops" generated a score of .75, the preceding search would produce a score of .375.

SUM

Calculates scores for documents matching a query by adding together, to a maximum of 1, the scores for the query's search elements. To arrive at a document's score, the search engine calculates a score for each search element and adds these scores together.

Following is an example query expression:

```
<SUM> ("computers","laptops")
```

If a search on "computers" generated a score of .5 and a search on "laptops" generated a score of .2, the preceding search would produce a score of .7 If a search on "computers" generated a score of .5 and a search on "laptops" generated a score of .75, the preceding search would produce a score of 1.00 (the maximum).

YESNO

Forces the score of an element to 1, if the element's score is nonzero. Examples help clarify this.

```
<YesNo> ("Chloe")
```

If the retrieval result of the search on "Chloe" was .75, with the `YesNo` operator, the result would be 1; if the retrieval result is 0, it remains 0.

This operator allows you to limit a search to only those documents matching a query, without the score of that query affecting the final scores of the documents. For example, to search among documents that contain "Chloe," with "Mead" as the determinant for ranking, you cannot simply specify the following:

```
"Chloe" <AND> "Mead"
```

The previous query would produce documents ranked with scores combined from both elements. The following query retrieves the results you want:

```
<YesNo> ("Chloe") <AND> "Mead"
```

If the retrieval result of the search on “Chloe” was .5 and that on “Mead” was .75, without the `YesNo` operator, the combined result would be .5; with the operator, however, it is .75, because the score of `AND` is calculated to be the minimum score of all its search elements.

Natural Language Operators

The natural language operators enable you to specify search criteria using natural language syntax. The search engine uses natural language analysis to translate the query text into Verity query language expression for evaluating and scoring documents. The `FREETEXT` and `LIKE` natural language operators are intended mainly for use by application developers.

FREETEXT

Interprets text using the free text query parser and scores documents using the resulting query expression. All retrieved documents are relevance-ranked. For information about the free text query parser, see “Query Parsers” on page 133.

This operator provides the functionality of the free text query parser, but allows you to combine free text queries with other search criteria using the full Verity query language. For example:

```
<FREETEXT> ( "peace negotiations in the Middle East" ) <AND>
(DATE > 01-01-96)
```

The quotation marks are required. If you want to include embedded quotes, they must be preceded with backslashes, as:

```
<FREETEXT> ( "\"Independence Day\""), ("\"The Arrival\""), science
fiction" )
```

Note In the case where a query or document contains only words defined as stop words in the collection `style.stp` file(s), the free text query parser uses the stop words for the query, ignoring the stop words list.

The `FREETEXT` operator can be combined with other operators in the same way as the `ACCRUE` operator.

LIKE

Searches for other documents that are *like* the sample one or more documents or text passages you provide. The search engine analyzes the provided text to find the most important terms to use for the search. If multiple samples are provided, the search engine assumes all of the samples are about a single theme and selects important terms common across the samples. Retrieved documents are relevance-ranked.

The `LIKE` operator accepts a single operand, called the QBE (query-by-example) specification. The QBE specification can be either the literal text of the example to query on, or it can be a specification of one or more full documents and text passages to use as positive and negative examples.

Note In the case where a query or document contains only words defined as stop words in the collections `style.stp` file(s), a QBE query with the `LIKE` operator returns no results.

Syntax

Document specification is made with a series of text references enclosed in braces. The syntax for specifying references is:

`{[name=]type:value [name=]type:value ...}`

where:

- *`name`* is either `posex` (*positive example*), or `negex` (*negative example*).

  A *negative example* reduces the weights of terms when they occur in a positive example. If terms from a negative example do not exist within the positive example, the negative example has no effect. (Hence a `negex` by itself makes no sense.)

- *`type`* can be one of the following:
  - `VdkVgwKey`, to specify a document by external key
  - `VdkDocId`, to specify a document by internal (session-specific) key,
  - `File`, to specify a file containing the document text (plain text only; no HTML or other formatting)
  - `Text`, to specify the text directly

- `value` is a reference to a piece of text to use as the positive or negative example.

If *name* is not specified, *value* is assumed to be a reference to a positive example (that is, `posex` is the implied name).

The value of *value* depends on *type*:

- `VdkVgwKey` and `VdkDocId`: the document key
- `File`:

filename[:*offset*:*range*]

where a byte offset into the file and a byte range from that offset can be optionally specified

- `Text`: literal text.

If there is no explicit type specifier, *value* is interpreted in the following ways:

- `VdkDocId` if it starts with a # character
- Literal text if it starts with a quotation mark
- `VdkVgwKey` for all other cases

The `Like` operator can be combined with other operators using the same rules as for the `ACCRUE` operator.

Special Characters in VdkVgwKey Fields

The syntax for the `LIKE` operator allows VdkVgwKeys to be enclosed in quotes (either single or double) to avoid parsing confusion. This means VdkVgwKeys containing things like whitespace, curly braces, and quotes can be handled. Backslash must be used to escape quote characters and backslashes embedded in the key, as is standard for string handling.

The syntax supports the use of single quotes for enclosing literal text examples, as in {text:'sample text'}.

The syntax for text: and vdkvgwkey: references has been enhanced to allow the reference value to be enclosed in either single or double quotes, with the usual backslash escaping mechanisms for embedded backslashes and quotes.

Concerning the backslash character in document keys, follow these guidelines. When a backslash appears in a document key, you must enter two backslashes in the <LIKE> syntax. See "VdkVgwKey Fields on Windows Systems" for important information about specifying paths on Windows systems.

Syntax examples are:

```
<LIKE> ( "{text:'sample text'}" )
<LIKE> ( "{text:"sample text"}" )
<LIKE> ( "{text:"sample ‘quote'"}" )
<LIKE> ( "{text:"sample \"quote\""}" )
<LIKE> ( "{vdkvgwkey:keyname}" )
<LIKE> ( "{vdkvgwkey:'{keyname}'}" )
<LIKE> ( "{vdkvgwkey:"{keyname}"}" )
<LIKE> ( "{vdkvgwkey:"c:\\my\\data"}" )
```

VdkVgwKey Fields on Windows Systems

To specify a VdkVgwKey including backslashes on Windows systems, you must double escape the two required backslashes. This means you must enter four backslashes, as shown in the following example.

```
<LIKE> ( "{vdkvgwkey:"c:\\\\my\\\\data"}" )
```

Examples of LIKE Expressions

The following examples illustrate uses of the `LIKE` operator.

Just literal text:

```
<LIKE> ("The dog ate the shoe.")
```

Explicit specification of a single positive example:

```
<LIKE> ( "{posex=vdkvgwkey:doc1}" )
```

Explicit specification of multiple positive and negative examples:

```
<LIKE> ( "{posex=vdkdocid:1234 posex=vdkvgwkey:doc1
negex=text:"stock market"}" )
```

Same as the preceding but with implied reference types:

```
<LIKE> ( "{posex=#1234 posex=doc1 negex=\"stock market\"}" )
```

Similar to the preceding but with implied `posex` names:

```
<LIKE> ( "{vdkdocid:1234 vdkvgwkey:doc1}" )
```

Same as the preceding, but using the most implicit syntax:

```
<LIKE> ( "{#1234 doc1}" )
```

You can combine a text reference list with literal text:

```
<LIKE> ( "{#1234 doc1} And more text" )
```

The preceding QBE specification is equivalent to this:

```
<LIKE> ( "{#1234 doc1 text: \"And more text\"}" )
```

The simplest way of specifying a single positive example by `VgwKey`:

```
<LIKE> ( "{doc1}" )
```

The example is in the file `doc.txt`, starting at the 100th byte:

```
<LIKE> ( "{posex=file:doc.txt:100:200}" )
```

To use a file reference with spaces in the file name, use single quotes, as follows:

```
<LIKE> ( "{file:‘my file.txt’}" )
```

To specify offset and length with single-quoted-filenames, use the following syntax:

```
<LIKE> ( "{file:‘my file.txt:0,5’}" )
```

Quotation marks embedded in `LIKE` expressions must be preceded by backslashes. The backslash indicates to the engine that the following character is supposed to be treated as a literal character.

Efficiency Considerations

In order to process a `LIKE` expression, the search engine must analyze the full text of the examples in the QBE specification. This has the potential to be time consuming, especially if the example documents are large or require expensive filtering.

The processing of `LIKE` queries can be accelerated by extracting feature vectors for documents at indexing time. Feature vectors are extracted during indexing when an appropriate entry is made in the `style.prm` file, as described in the *Verity Collection Reference*. With feature vectors available in the collection, the search engine does not need to touch the original text of the example documents and `LIKE` queries are processed very efficiently.

PART II

Topics

- Chapter 6: Elements of Topic Design
- Chapter 7: Using Topic Outline Files
- Chapter 8: Building Topic Sets from the Command Line

6

Elements of Topic Design

This chapter describes the features of topics, including:

- About Topics and Topic Sets
- How Topics Work
- Rules About Topics and Topic Sets
- Topic Design Strategies

About Topics and Topic Sets

A *topic* is a *grouping of information that comprises a topic definition* related to a concept or a subject area. In terms of the implementation, a topic is a stored query in Verity Query Language (VQL).

A *topic set* is a *grouping of topic definitions* that have been compiled for use by a Verity application.

Because a topic set represents many concepts, or search terms, it is sometimes referred to as a *knowledge base*. It is a catalogue of predefined queries that can be referenced at search time to expand user queries. A Verity knowledge base can consist of one or more topic sets. See "Building Topic Sets from the Command Line" for information about building topic sets.

The subject area of a topic is typically identified by the topic name. For example, the subject of a topic could be financial documents. This topic could be composed of two structural elements: its *name*, for example `finance`, and its *evidence topics* (important terms, acronyms, or jargon used to define the subject) that could contain `inc`, and `company`.

```
finance <Accrue>
  inc <Accrue>
  company <Accrue>
```

Operators and *modifiers* are the glue joining related evidence topics. Operators represent logic to be applied to evidence topics. Modifiers apply further logic to evidence topics. For example, a modifier can specify that documents containing an evidence topic not be included in a list of results.

A topic's structure becomes more sophisticated as topics are added to it. In the following example, the topic `bond` has been added to the structure.

```
finance <Accrue>
  inc <Accrue>
  company <Accrue>
bond <Accrue>
```

Next, another topic, `corporate`, is added at the top level, giving a structure similar to:

```
finance <Accrue>
  inc <Accrue>
  company <Accrue>
bond <Accrue>
corporate <Accrue>
```

Finally, `finance` and `bond` are dragged under `corporate` to form what is now a *top-level topic*, `corporate`. In this new structure, `finance` and `bond` are *subtopics* of the topic `corporate`. `inc` and `company` become *evidence topics* beneath the `finance` subtopic.

```
corporate <Accrue>
  finance <Accrue>
  bond <Accrue>
```

Sophisticated topics are composed of:

- top-level topics
- subtopics
- evidence topics

These elements determine the related subject areas of a topic. Typically, a knowledge base consists of several top-level topics.

Note Subtopics and evidence topics can be used by multiple top-level topics. See "Topic Structure" for more information about topic levels.

In the previous illustration, you might notice the word `ACCRUE`. The Verity search engine is built on the notion that topics represent search concepts. Queries that go beyond a single word or phrase typically involve the `ACCRUE`-class operators (`ACCRUE`, `AND`, `OR`) to combine several branches of evidence in a topic tree. At search time, the combined evidence is evaluated.

Topic Structure

Top-level topics are the highest topics defined in a topic structure. Top-level topics represent the subject areas you want a Verity search agent to find.

Subtopics form the levels between top-level topics and evidence topics.

The name of a subtopic should reflect the subject area that its subtopics or evidence topics combine to describe.

Evidence topics are the lowest units of a topic structure. Evidence topics are strings, made up of combinations of alphanumeric characters. An evidence topic can contain up to 128 alphanumeric characters.

Topic and Subtopic Relationships

Each topic and its associated subtopics form a hierarchical parent and child relationship. When you use a topic to perform a search, the subject area defined by the topic includes its subtopics, their subtopics, and so on, down to the evidence topics of the structure. Topics that are not direct descendants of the topic you use will not be included in the search. A subtopic can be shared by multiple topics.

Storing Topic Sets

Topic sets can be stored in one of the following ways:

- The most common form is in binary files inside a particular directory structure. (The top level directory has the same name as the topic. Inside are a `sysind` directory, and files with names with a format such as `00000000.std`.)
- For backward compatibility, topic sets can also be stored in ASCII representations called *topic outline files*, which have an `.otl` filename extension. For more information, see "Using Topic Outline Files" on page 101

Intelligent Classifier can open and save to either format. The directory format is the native format used by most Verity applications. For more information about using the OTL files, see "Using Topic Outline Files" on page 101 and the *Verity Intelligent Classification Guide*.

How Topics Work

Topics have two main functions:

- You can use them as stored queries in other Verity applications. End users can quickly find information without composing sophisticated queries using complex syntax.

  Optionally, topics can also be preindexed to speed their execution.

  To use topics this way, the topic set must be imported into the application. See "Using Topics as Stored Queries in Other Verity Applications."
- In Intelligent Classifier, topics can also be used to control which documents are assigned to a given category in a taxonomy, and to control the documents assigned in a knowledge tree or parametric index. In this case, the topics are not visible to the end user.

If topics are used this way, the topic set does not need to be imported into another Verity application. The taxonomy or Knowledge Tree *does* need to be imported.

Using Topics as Stored Queries in Other Verity Applications

Four main steps are involved in using topic sets with other Verity applications:

1. Create the topic set(s).

   You can use the `mktopics` command line tool (see "Building Topic Sets from the Command Line") to convert topic sets in OTL format to binary format.

   Intelligent Classifier provides a powerful graphical editor to create topic sets in either binary format or OTL format. Intelligent Classifier can also open an OTL file and export it to a binary format topic set.

2. If you create more than one topic set to be used in the same Verity application, you must also supply a knowledge base map (KBM) file.

   If you are using Intelligent Classifier, a KBM file is automatically created for you. For more information, see the *Verity Intelligent Classification Guide*.

3. Optionally, you can preindex a topic set against a collection. Preindexing speeds up the time taken to run queries that use topics.

   You can use command line tools such as `mktopics` to preindex. See "Building Topic Sets from the Command Line" on page 119 for information about `mktopics`.

   Intelligent Classifier enables you to specify which parts of a topic set will be preindexed.

4. You must tell the Verity application to use the topic set or KBM file. For information about how to configure topic sets, see the *Verity Intelligent Classification Guide*.

Note Topic sets must be in binary format so that other Verity applications can use them. You can use either Intelligent Classifier or the `mktopics` command-line tool to convert an OTL formatted topic set into binary format.

Making Topics Available

You need to make a topic set available to a K2 Server. For complete information about using K2 brokers and servers, see the *Verity K2 Dashboard Administrator Guide*.

Rules About Topics and Topic Sets

Although topics are a powerful search tool, there are certain rules that apply to both topics and topic sets.

Operator Precedence Rules

The Verity search engine uses precedence rules to determine how operators are assigned. These rules state that some operators rank higher than others when assigned to topics, and affect how document selections are performed. See "Precedence Rules" on page 39 for the operator precedence rules.

Rules About Topics

The following list describes a few rules that you should be aware of when using Verity topics.

- Topic Names—All topic names can contain up to 128 alphanumeric characters, including hyphens (-), underscores (_), plus signs (+), dollar signs ($), percent signs (%), periods (.) and circumflexes (^). A topic name cannot start with a period. A topic name must be unique in a topic set.
- Case Sensitivity—Topic names and evidence topics are normally case-insensitive. You can name an evidence topic using all caps, as in `APPLE`; initial caps, as in `Apple`; or all lowercase, as in `apple`. Case is not considered when a search is performed. Thus, if your evidence topic is entered as `APPLE`, the Verity search engine selects documents containing "APPLE", "Apple", or "apple".
- Topic Limits—See "Query Limits" on page 149 for information about topic limits.

Topic Design Strategies

Topic design is more of a task of strategy than a task of organization. As a topic designer, you are encouraging users to access knowledge according to a particular strategy. The topics you define are the tactics you will use to implement that strategy.

Consider the topics you define as questions to be asked, just as you might ask a reference librarian at your local library for information relating to a subject area.

Once you have an understanding of your documents, you are ready to choose a topic design strategy. You can either start with major subject classifications and become increasingly more specific (top-down design), or start with specific details and then create more general "umbrellas" to encapsulate them (bottom-up design).

Table 6-1 and Table 6-2 summarize the two design strategies.

Top-Down Design

Table 6-1 describes the top-down design strategy.

Table 6-1 Top-down Design Strategy

Optimal Conditions	Implementation	Maintenance
Works best with clearly-defined requirements. Ideal if your searchable document set is constantly growing or changing. With this strategy, you are likely to define subjects that might not yet be evident in information sources. **Advantage:** If you find that many new documents contain information not identified in your topic design, you can add new topics.	**Top-level topics:** Use general headings to identify the subject area. **Subtopics:** Use more specific headings to identify primary groupings. **Evidence topics:** Use important terms, acronyms or jargon to define the subject.	If your information sources (your set of indexed documents) change constantly, subjects within documents can be missed, especially at the lowest levels. Periodically re-analyze the information being selected by your topics to ensure that topics critical to your application are current, and the appropriate information is being found.

Bottom-Up Design

Table 6-2 describes the bottom-up design strategy.

Table 6-2 Bottom-up Design Strategy

Optimal Conditions	Implementation	Maintenance
Works best when you have documents that are representative of many other documents that contain similar information. This approach is also useful when your information sources are not subject to many changes or additions.	**Evidence topics:** Start with a document that contains a good representative sample of words or phrases you want to search for. Then group these words by successively higher classifications. **Subtopics and Top-level topics:** Design the topic from individual evidence topics, up through the top level to be defined. With this strategy, your topic design objective is to select documents containing information similar to your lower-level topics.	Keep in mind that topic designs based on the contents of specific documents can miss related subject areas in other documents. For example, if a name is used in the sample document and that name changes in other documents, the new name can be missed in searches. In addition, the specific document set being used to develop your topics might not be representative of all documents contained in your information sources. Periodically review the effectiveness of your searches.

7

Using Topic Outline Files

A topic represents a concept or a subject area. Each topic is assigned a unique name. When Verity applications incorporate topics, users can find information by typing topic names instead of elaborate queries using complex syntax.

Topics offer many benefits:

- Reusability—they can be shared among users
- Simplicity—they can eliminate the need for complex queries
- Speed—they can improve search time

This chapter contains the following information:

- About Outline (OTL) Files
- Creating a Topic Outline File
- Defining Topics in the OTL File
- Topic Outline File Elements
- Topic Structure

About Outline (OTL) Files

A topic outline file (an OTL file) is an ASCII text file named with the suffix `.otl`. You can create a topic outline file from scratch in any directory using your favorite ASCII text editor.

An outline file is required for use with the `mktopics` tool. When you use an OTL file with `mktopics`, you include the file on the command line.

You can use Verity Intelligent Classifier to create `.otl` files. Check the `.otl` file format when you create a new topic set from the Intelligent Classifier window or when you export the current topic set as an `.otl` file. Optionally, you can create a text-based outline file and import the file for use with Intelligent Classifier.

Topic outline files define topics and include the elements described in Table 7-1.

Table 7-1 Topic Outline File Elements

Element	Consists Of	Used For
`$control:`	`1` keyword and comment lines	A control file to be used by Verity search engine tools
Topic definition modifiers	Modifiers to be assigned to the top-level topic, its subtopics, and its evidence topics	Define evidence topics, track who has updated the topic outline file, when additions or edits have been made, and add annotations to topics
Indentation characters	Asterisks (`*`) that denote indentation of subtopics and evidence topics within a topic structure	Indicate the level of a topic with respect to its parent topic and its children

Creating a Topic Outline File

A topic outline file is an ASCII text file named with the suffix `.otl` and used to define topic structure. You can create a topic outline file in any directory using your favorite ASCII text editor. If you are using the Verity Intelligent Classifier, you can export your taxonomy as an OTL file.

This file is optional, but if you want to set up a tailored taxonomy, you need to create this file.

The file includes the following topic definition elements:

- `$control: 1` keyword and comment lines
- The topic definition, which includes the weight, operator, and modifiers to be assigned to the top-level topic, its subtopics, and its evidence topics
- Topic definition modifiers, as needed by the topic
- Indentation characters, which indicate the level of that topic with respect to its parent topic and its children

These elements are shown in the following example, which defines a topic named `art`.

```
Topic Outline
File Elements              Topic Outline File
keyword                    $control: 1

comment line               # Beginning of art topic definition
top-level topic            art ACCRUE
                               /author = "fsmith"
topic definition modifiers     /date   = "30-Dec-01"
                               /annotation = "Topic created
                                              by fsmith"
subtopic topic             * 0.70 performing-arts ACCRUE
  evidence topic           ** 0.50 WORD
  topic definition modifier    /wordtext = ballet
  evidence topic           ** 0.50 STEM
  topic definition modifier    /wordtext = dance
  evidence topic           ** 0.50 WORD
  topic definition modifier    /wordtext = opera
  evidence topic           ** 0.30 WORD
  topic definition modifier    /wordtext = symphony
subtopic                   * 0.70 visual-arts ACCRUE
                           ** 0.50 WORD
                               /wordtext = painting
                           ** 0.50 WORD
                               /wordtext = sculpture
subtopic                   * 0.70 film ACCRUE
                           ** 0.50 STEM
                               /wordtext = film
subtopic                   ** 0.50 motion-picture PHRASE
                           *** 1.00 WORD
                               /wordtext = motion
                           *** 1.00 WORD
                               /wordtext = picture
                           ** 0.50 STEM
                               /wordtext = movie
subtopic                   * 0.50 video ACCRUE
                           ** 0.50 STEM
                               /wordtext = video
                           ** 0.50 STEM
                               /wordtext = vcr
                           # End of art topic
```

Defining Topics in the OTL File

As you begin defining topics in an outline file, it is a good idea to define simple topics and then add to them. This section discusses some aspects of creating a topic outline file.

If you are receiving errors from `mktopics` about your topic outline file, check for precedence rule violations.

In the following example, new `STEM` evidence topics, `bond` and `corporate` have been added to the `finance` topic.

```
finance <Accrue>
  inc <Accrue>
  company <Accrue>
bond <Accrue>
```

The `finance` and `bond` topics are moved under the `corporate` topic to make `corporate` the new stem topic.

```
corporate <Accrue>
  finance <Accrue>
  bond <Accrue>
```

The edited outline file for this topic structure appears as follows.

```
$control: 1
# Beginning of corporate topic
corporate ACCRUE
  /date = "30-Dec-01"
  /annotation = "Generic corporate information."
* 0.50 finance ACCRUE
** 0.50 WORD
  /wordtext = inc
** 0.50 STEM
  /wordtext = company
** 0.30 WORD
  /wordtext = corporate
* 0.50 bond ACCRUE
  /date = "31-Dec-01"
# End of corporate topic
```

After you start using a topic to perform searches, you might find you need to make additions to enhance the topic's document selection.

Specifying Weights with Subtopics

When you define a subtopic as a top-level topic, include the subtopic weight where you name it under its parent topic, because a child's weight is significant in relation to its parent. Because of this, you can use the subtopic in several locations in a topic outline file, depending on your desired selection results.

The weight for a subtopic can be a value from 0.01 to 1.00, where `1.00` indicates that the topic is of very high importance. Alternatively, the weight can be a value from 1 to 100, where `100` indicates that the topic is of very high importance.

In the following topic outline file, the subtopic `finance` appears as a top-level topic, but is used in two different locations. Each location uses a different weight for this subtopic, depending on its relevance to the parent topic. The abbreviated style is used to define phrase evidence topics and word evidence topics.

```
$control: 1
# Beginning of finance topic
finance ACCRUE
* 0.70 "corporate finance"
* 0.40 "financial information"
* 0.60 'money'
# End of finance topic

# Beginning of bond definition
bond ACCRUE
* 0.50 "corporate bonds"
* 0.50 "stocks and bonds"
* 0.50 'finance'
# End of bondtopic
```

The weights specified with the children of the `finance` subtopic remain the same, even though the weight of the `finance` subtopic differs where it has been specified under the `bond` topic.

Including and Excluding Documents

You can include or exclude documents by defining a field evidence topic at a higher level than the evidence topics that will be used to select documents. You can limit selected documents to those that match the field evidence topic.

For example, the following topic limits the search performed by the `bond` topic to documents dated October 10, 2000 and later. This example assumes that the `bond` topic has been previously defined in the topic outline file.

```
$control: 1
# Beginning of bond-limit topic
bond-limit AND
* 1.00 bond
* 1.00 bond-date FILTER
  /definition = "DATE >= 10-Oct-01"
# End of bond-limit topic
```

In the previous example, the `bond-limit` topic uses the `AND` operator. The `bond` and `bond-date` subtopics each have a weight of 1.00. Use of the `AND` operator along with weight assignment ensures that both subtopics must be present in a document when the `bond-limit` topic is used.

Specifying Field Evidence Topic Ranges

To specify a range, define a parent topic that uses the `AND` operator and has two field evidence topics as children. Use the operators `GREATER THAN OR EQUAL TO (>=)` and `LESS THAN OR EQUAL TO (<=)` to specify the beginning and ending values of the range.

In the following example, the field evidence topic named `date-range` selects documents dated from February 1, 2001 through February 28, 2003.

```
$control: 1
date-range AND
* from-date FILTER
  /definition = "Date >= 01-Feb-01"
* to-date FILTER
/definition = "Date <= 28-Feb-03"
```

Topic Outline File Elements

The elements that make up a topic outline file are described in the following sections.

$control: 1 Keyword

The `$control: 1` keyword identifies a file as a control file to be used by Verity search engine tools. This keyword always appears as the first non-comment line in a topic outline file, and is entered as `$control: 1`, as shown in the examples in "Including and Excluding Documents" on page 105. The number following the keyword denotes the file version number, and is used internally by the Verity search engine.

Comment Lines

You can include comment lines in a topic outline file by beginning the lines with a pound sign (#) character. Comment lines can appear on separate lines, or can be added to the end of a statement. Blank lines can also be used to separate topics and improve readability.

Topic Definition Modifiers

Topic definition modifiers define evidence topics, track who has updated the topic outline file, track when additions or edits have been made, and add annotations to topics. You can use the topic definition modifiers in a topic outline file.

The topic definition modifiers are optional; you can create topics without the modifiers. The topic definition modifiers are described in Table 7-2.

Table 7-2 Topic Definition Modifiers

Modifier	Description
/annotation	Use the /annotation topic definition modifier to add a description to the topic. You can enter up to 2,000 character of text. Enclose text in double quotes (""), as shown in the following example: `/annotation = "This topic is used by fsmith."`
/author	Use the /author topic definition modifier identify the author of a top-level topic, a subtopic, or an evidence topic. Enter the name of the author in double quotes ("") following the /author topic definition modifier, as shown in the following example: `/author = "fsmith"`
/date	Use the /date topic definition modifier to identify the date a topic has been added or changed. You can enter the date in any manner you wish. Enclose the date in double quotes ("") following the /date topic definition modifier, as shown in the following example: `/date = "December 18, 2001"`
/definition	Use the /definition topic definition modifier when you want to use a relational operator with a field topic. Field topics contain fields that have been defined in the style.ddd file. Relational operators perform a filtering function by selecting documents that contain specified values in their associated fields (such as AUTHOR). The fields that are used with relational operators can contain numbers or alphanumeric characters.

/wordclass	Use the /wordclass modifier when you want to reference qualify instance data. Qualify instance data can be referenced only by an application built by the Verity Developer's Kit. For example, to define a topic for instance 39 of the word orange, use this syntax:
	`* t27 word`
	`/wordtext  = "orange"`
	`/wordclass = 39`
	Alternatively, you can use this syntax:
	`* "orange"`
	`/wordclass = 39`

/wordtext	Use the /wordtext topic definition modifier to define a evidence topic (the lowest topic level).
/zonespec	Use the /zonespec topic definition modifier when you want to reference a zone in a topic outline file.

Indentation Characters

Use asterisks (*) to denote indentation of subtopics and evidence topics within a topic structure. For example, a top-level parent topic uses no indentation. Subtopics of a top-level topic are preceded by one asterisk; subtopics of a subtopic are preceded by two asterisks; children of subtopics are preceded by three asterisks, and so on.

Topic Structure

A topic can be as simple or as complex as needed, and can contain as many levels of indentation as are required to accurately describe the characteristics of the parent topic.

As you prepare to create topics, consider the naming conventions you will use. Topic names should help identify the subject matter of the kinds of documents you want to select.

To ensure the best search performance, use alphanumeric characters (A through Z, and 0 through 9) for topic names. You can also use foreign characters with ASCII values greater than or equal to 128, as well as the following symbols.

$	dollar sign
%	percentage sign
^	circumflex
+	plus sign
-	dash
_	underscore

Using other non-alphanumeric characters could cause misinterpretation of the topic name and could affect results.

A topic name can be up to 128 characters long. Topic names can be entered in any combination of uppercase and lowercase letters.

WARNING! If two topics with the same name exist but have different topic outline files, the second topic outline file replaces the first topic outline file when the topic building tool is used. There is no warning message.

Defining Topic Structure

You can name and structure topics to guide a user's search. Topic names should help identify the subject matter of the kinds of documents you want to select.

The following sections contain examples of structuring topics to tailor information retrieval.

Defining Top-level Topics

To define a top-level topic, in the OTL file, type the topic name first, followed by the operator to be assigned to the topic. In the following example, the top-level topic named `merger-activity` includes the `/author`, `/date`, and `/annotation` topic definition modifiers:

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
```

The `/author`, `/date`, and `/annotation` topic definition modifiers are optional; you do not need to include these modifiers in your topic.

Defining Subtopics

To define a subtopic, enter the indentation character(s) first, followed by the weight to be assigned to the subtopic, the subtopic name, and the operator to be assigned to the subtopic. You can also include the `/author`, `/date`, and `/annotation` modifiers.

In the following example, the subtopics named `trade-action`, `speculation`, and `offering-postponed` have been added to the top-level topic named `merger-activity`. In addition, the subtopic named `offering-postponed` includes the subtopic named `standstill-agreement`.

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
* 0.50 offering-postponed AND
** 0.50 standstill-agreement PHRASE
```

When creating a subtopic that uses the PHRASE operator, you can define the words that comprise the phrase as evidence topics, or you can use an abbreviated style to define the phrase. For more information, refer to "Defining Subtopics Using the PHRASE Operator" on page 114.

Subtopic Weight Assignments

If you do not assign a weight to a subtopic, the Verity search engine automatically assigns a weight based on the operator used by the subtopic's parent. So, in the previous example if the parent (*) uses the ACCRUE operator, the subtopic (**) will have an automatic weight assignment of 0.50.

The Verity search engine ignores a weight assigned to a subtopic with a parent that does not accept a variable weight.

Assigning the NOT Modifier to Subtopics

To assign the NOT modifier to a subtopic, enter the modifier following the indentation character(s) and preceding the weight. In the following example, the subtopic named standstill-agreement has been assigned the NOT modifier.

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
* 0.50 offering-postponed AND
** NOT 0.50 standstill-agreement PHRASE
```

To specify a NOT modifier, you can use a tilde character (~) instead of the word NOT.

Evidence Topics

Enter information for evidence topics in the following format:

```
weight evidence_topic_OPERATOR
     /wordtext = evidence_topic_name
```

Table 7-3 describes the elements used in evidence topic syntax.

Table 7-3 Evidence Topic Elements

Element	Description
`weight`	Assigns a weight to the definition topic, if a weight is accepted by the parent. If a weight is not assigned, the engine automatically assigns a weight of 1.00. The weight can be a value from 0.01 through 1.00, where an assignment of 1.00 indicates that the topic is of very high importance. Alternatively, the weight can be a value from `1` through `100`, where 100 indicates that the topic is of very high importance.
`evidence_topic_OPERATOR`	Specifies the evidence topic operator to be used.
`evidence_topic_name`	Specifies the name to be assigned to the evidence topic. **Note:** The evidence topic name is not assigned in double quotes.

In the following example, the evidence topics `rise`, `rose`, and `speculation` have been assigned to the subtopic named `speculation`.

Note Asterisks (`*`) are used as indentation characters to indicate the relationship between these evidence topics and their parent topic.

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
** 0.50 STEM
   /wordtext = rise
** 0.50 STEM
   /wordtext = rose
** 0.50 STEM
   /wordtext = speculation
* 0.50 offering-postponed AND
** NOT 0.50 standstill-agreement PHRASE
```

Evidence topics can also be defined in an abbreviated style that does not use the `/wordtext` topic definition modifier. For more information, refer to "Topic Structure" on page 109.

Evidence Topic Weight Assignments

If you do not assign a weight to an evidence topic, the engine automatically assigns a weight based on the operator used by the evidence topic's parent. Thus, if the parent uses the `AND` operator, the evidence topic will have an automatic weight assignment of 1.00.

If you assign a weight to an evidence topic with a parent that does not accept a variable weight, the Verity engine ignores the weight.

Assigning Modifiers to Evidence Topics

To assign the `MANY` or `CASE` modifier to an evidence topic, enter the modifier before the evidence topic operator, using this syntax:

```
<MODIFIER><OPERATOR>
<MODIFIER><MODIFIER><OPERATOR>
```

or this syntax:

```
<OPERATOR/MODIFIER>
<OPERATOR/MODIFIER/MODIFIER>
```

To assign the `NOT` modifier to a evidence topic, enter the modifier preceding the weight.

You can use one, two, or three modifiers, depending on the function of the evidence topic and the operator assigned to its parent. The following example illustrates how modifiers can be assigned to evidence topics.

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation AND
** 0.50 STEM
   /wordtext = rise
** 0.50 <WORD/CASE/MANY>
   /wordtext = Rose
** 0.50 <MANY><STEM>
   /wordtext = speculation
```

Abbreviated Evidence Topics

When defining evidence topics that use the `WORD`, `STEM`, or `SOUNDEX` operators, you can abbreviate the entries in your topic outline file. Abbreviated evidence topics do not use the `/wordtext` topic definition modifier. Weights can be assigned to an abbreviated evidence topic, depending on the operator used by the parent topic. If no weight is assigned, the Verity engine automatically assigns the evidence topic a weight of 1.00.

The following examples illustrate defining abbreviated evidence topics.

Abbreviated WORD Evidence Topics

Evidence topics that define words can be abbreviated by enclosing the word in double quotes (""), as shown in the following:

```
* 0.50 "acquisition"
```

Abbreviated STEM Evidence Topics

Evidence topics that define stems can be abbreviated by enclosing the stem in single quotes ('), as shown in the following:

```
* 0.50 'merger'
```

Abbreviated SOUNDEX Evidence Topics

Evidence topics that define sound-alike words can be abbreviated by enclosing the word to be matched with at symbols (@), as shown in the following example:

```
* 1.00 @airplane@
```

Defining Subtopics Using the PHRASE Operator

When you use the `PHRASE` operator to define a subtopic, enter the words that comprise the phrase as children of the subtopic. The order in which you list the words of the phrase specifies how they must appear in documents.

For example, the phrase “standstill-agreement” is defined by the subtopic `standstill-agreement`, as follows:

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
* 0.50 offering-postponed AND
** NOT 0.50 standstill-agreement PHRASE
```

```
*** 1.00 WORD
   /wordtext = standstill
*** 1.00 WORD
   /wordtext = agreement
```

You can also define phrases by enclosing the words that comprise the phrase in double quotes (""), as follows:

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
* 0.50 offering-postponed AND
** "standstill agreement"
```

When this abbreviated style is used, the phrase does not have a title.

When the abbreviated style is used to define a phrase, the weight does not have to be included, because children of `PHRASE` topics do not accept weights other than 1.00.

Defining Field Evidence Topics

Type information for field evidence topics using the following format:

```
topic_name FILTER
   /definition = "FIELD OPERATOR value"
```

Table 7-4 describes the elements for field evidence topic syntax.

Table 7-4 Field Evidence Topic Elements

Expression Element	Description
topic_name	Specifies the name assigned to the topic.
FILTER	Specifies that a field search is to be performed using the information enclosed in quotes.
/definition =	Specifier preceding the field evidence topic definition.
FIELD	Specifies the name of the field to be searched. The field name given must be a valid field name, as defined in the collection policy files.
OPERATOR	Specifies the name of the relational operator to be used.
value	Specifies the variable to be used to perform the field search.

The argument used with the `/definition` topic definition modifier is enclosed in double quotes (""). The relational operators that can be assigned to field evidence topics are listed in Table 7-5.

Table 7-5 Field Evidence Topic Operators

Operator Name	Alternate Symbol	Notes
EQUALS	=	You can use symbols rather than names when specifying the first five operators
GREATER THAN	>	
GREATER THAN OR EQUAL TO	>=	
LESS THAN	<	
LESS THAN OR EQUAL TO	<=	
CONTAINS		
MATCHES		
STARTS		
ENDS		
SUBSTRING		

Note When a character string is used to represent a value for a field containing text, case is ignored. So if you enter `TITLE contains computer` as an argument, documents containing the word Computer, COMPUTER, or computer in their titles will be located.

The following example illustrates a field evidence topic that performs a filtering function for documents containing the word `Economy` in the `TITLE` field.

```
merger-activity ACCRUE
   /author = "fsmith"
   /date = "30-Dec-01"
   /annotation = "This topic used by Marketing."
* 0.50 merger-title FILTER
   /definition = "TITLE Contains Economy"
* 0.50 trade-action ACCRUE
* 0.50 speculation ACCRUE
* 0.50 offering-postponed AND
** "standstill agreement"
```

When the topic `merger-activity` is used in a search, documents containing the word `Economy` in their titles and containing the maximum occurrences of the other search criteria are scored higher than those that do not contain the word `Economy` in their titles, or have fewer occurrences of the other search criteria.

How Field Evidence Topics Affect Document Scores

When defining field evidence topics, keep in mind that the filtering function performed depends on where the field evidence topic exists within the topic set.

In the topic set listed above, documents that contain the word `Economy` in their titles, as well as the information specified by the other field evidence topics, are scored higher than documents that do not contain `Economy` in their titles, but contain the information as specified by the other field evidence topics.

Place field evidence topics at a higher level than other subtopics if you want to:

- Filter documents according to information specified by the field evidence topic
- Score selected documents based on the topic's remaining subtopics

Defining Topics for Zone Searching

To search over a zone named `title` for the stemmed variations of the words “health” and/or “insurance,” specify the following in the `topic outline` file:

```
health-care IN
    /zonespec = "title"
* <ANY>
** 'health'
** 'insurance'
```

You can also use multiple zones. To search over zones named `title` and `subject` for stemmed variations of the words “health” and “insurance,” specify the following:

```
health-care IN
    /zonespec = "title,subject"
*<ANY>
** 'health'
** 'insurance'
```

You can impose conditions. You can specify a search over a zone named `<A>` (anchor) in an HTML file. The corresponding `HREF` contains the term "verity.com". To look for the precise string "Verity" specify the following:

```
verity-link <In>
   /zonespec = "A <When> HREF <Contains> verity.com"
* "Verity"
```

Currently, the `IN` operator takes only one subquery node to find within the constraint of the zone. Similarly, the simple query construct:

```
cat <AND>dog<IN>title
```

is interpreted as:

```
cat<AND>(dog <IN>title)
```

instead of: (

```
cat<AND>dog)<IN>title
```

Defining Topics Using Score Operators

The use of the score operators `PRODUCT`, `SUM`, and `COMPLEMENT` in the `topic outline` file is shown in the following example:

```
Sample-Product <Product>
* 50 "steve"
* 25 "bill"

Sample-Sum <Sum>
* 75 "verity"
* 25 "search"

Sample-Complement <Complement>
* 25 "verity"

Sample-Complement-2 <Complement>
* SubTopic <Accrue>
** "steve"
** "bill"

Sample-YesNo <YesNo>
* "Steve"
```

8

Building Topic Sets from the Command Line

OTL format topic sets can be created using a text editor. However, this format cannot be deployed directly in other Verity applications. You can use the `mktopics` command-line tool to build a binary format topic set from an OTL file if you do not have Intelligent Classifier. There are two features unique to `mktopics`: generating encrypted topic sets to protect intellectual property, and indexing topic sets against collections to improve search efficiency.

Building a topic set from an OTL file using `mktopics` involves two major activities:

- Creating topic definitions in a topic outline (OTL) file
- Building topic sets using `mktopics`

This chapter discusses how to build a topic set using the `mktopics` command-line tool, and contains the following information:

- Starting mktopics
- mktopics Syntax
- Checking Topic Precedence Rules
- Topic Set Indexing
- Topic Set Encryption

Starting mktopics

The `mktopics` tool is used to build a topic set using the topic definitions contained in an outline (otl) file. When you run `mktopics` to build a new topic set, the following functions are performed:

- Topic definition syntax is checked. If an error is detected, the `mktopics` tool returns the line(s) containing the error(s), and provides possible options for correcting the error.
- If no errors are detected in the outline file, `mktopics` builds the topic set in the directory that you specify

`mktopics` also supports maintenance functions, as described later in this chapter.

Building a Topic Set

To build a topic set:

1. Open an ASCII editor, and create and save an outline file. See “Using Topic Outline Files” on page 101 for instructions.

   Save the file in any directory. Use the `-outline` command modifier to indicate the path and name of the outline file to be used with the `mktopics` command. (See step 3.)

2. Open a command window.

3. In the command window, issue the `mktopics` command, as in:

```
mktopics -topicset topicset_dir -outline file.otl
[-collection collection|@list] [-indexType normal|namedOnly]
```

`-topicset` *`topicset_dir`*	(Required) Identifies the directory name (*`topicset_dir`*) where the topic set will be created or updated.
`-outline` *`file.otl`*	(Required) Identifies the topic outline file (*`file.otl`*) where the topics are defined.

-collection *collection*|*@list*	(Optional) Identifies a collection (*collection*) or a list of collections (*list*) that will contain the topic index in the topicidx subdirectory. The list of collections is an ASCII file that lists the full path to each collection directory, on a separate line.
-indexType *normal*|*namedOnly*	(Optional) Specifies the type of index for the collection. A *normal* index indexes all topics with a precedence rating of incremental or lower. Indexes search and retrieval performance varies depending on the structure of the topic. A *namedOnly* index indexes only named topics. It takes longer to build, and uses more disk space than a normal index, but search and retrieval are almost instantaneous.

Sample mktopics Command

To create a new topic set, use a command similar to:

```
mktopics -quiet -topicset /dir1/mytopics -outline topics.otl
```

This `mktopics` command builds a new topic set in quiet mode in the directory called `/dir1/mytopics` from an outline file called `topics.otl`.

mktopics Syntax

A topic set can be built and updated using the `mktopics` tool. Some Verity applications support the use of multiple topic sets. For those applications, you need to run `mktopics` for each topic set desired.

mktopics Syntax Summary

The command-line syntax for the `mktopics` tool is shown below.

```
mktopics -topicset topicset_dir -outline file.otl
```

See "mktopics Syntax Descriptions" for descriptions of the options for `mktopics`.

mktopics Syntax Descriptions

Table 8-1 describes the syntax elements for `mktopics`.

Table 8-1 mktopics Syntax Elements

Element	Description
`-charmap`	The optional `-charmap` argument specifies the character set used to display messages and other mktopics screen output. Use a character set that your system can display properly. For information on the supported locales and relevant character sets, see the *Verity Locale Configuration Guide*.
`-collection` *`collection`*`|`*`@list`*	The optional `-collection` argument specifies a collection directory or an ASCII file containing a list of collection directories (each on a separate line); the name of such a list file must be preceded by an `at` sign. This argument specifies which collection(s) the topic set is indexed against. When specified, the topic set index is updated in the specified collection directories. Maintaining a topic index in a collection facilitates quick and efficient searches over the collection data when using topics.
`-deep`	The optional `-deep` argument specifies that a dump of a topic set to an outline file dumps each top-level topic as far down as possible. Only meaningful when used with `-fullotl`. This is the default.

Table 8-1 mktopics Syntax Elements (continued)

Element	Description
`-encrypt` *`keyfile`*	The optional `-encrypt` argument encrypts the topic set using the specified key file. Generate the key file with the `mkenc` tool.
`-fullotl` *`filename`*	The optional `-fullotl` argument exports topic definitions to an OTL file. The argument is followed by the full or relative path and name of the file to which you want to export a copy of the topic set. Use .otl as the file name extension. Additional arguments for `-fullotl` are `-topic`, `-deep`, and `-shallow`.
`-indexType normal|namedOnly`	The optional `-indexType` argument specifies the type of topic set index to be built when the topic set is indexed against a collection using operator precedence rules. Valid values are: ▪ `normal` for indexing topics in the outline file with an incremental precedence or lowest precedence ▪ `namedOnly` for indexing only named topics in the outline file The default is `normal`. For information about topic precedence ratings, refer to "Operator Precedence Rules."
`-locale`	The optional `-locale` argument specifies the locale for the topic set. The locale used must match the locale of the document collection associated with the topic set. For information on the supported locales and relevant character sets, see the *Verity Locale Configuration Guide*.
`-logfile` *`filename`*	The optional `-logfile` argument followed by a log filename indicates that a log file will be generated. For the log filename, you can specify the filename and path. If a path is not specified with the filename, the log file is put in the current working directory.
`-nowarnundef`	The optional `-nowarnundef` argument specifies that you will not be warned if there are any undefined topics when importing topic definitions from an outline file. Only meaningful when used with `-outline`. The default is `-warnundef`.
`-noprecres`	The optional `-noprecres` argument specifies that topic precedence checking will not occur when the topic set is built. If this argument is set, then topics with precedence errors are rewritten at query time, making the performance of topic searching slow. The default is `-precres`. Only meaningful when used with `-outline`.

Table 8-1 mktopics Syntax Elements (continued)

Element	Description
`-optimize`	The optional `-optimize` argument specifies that the topic set be optimized. The optimization rebuilds the topic set structure, recovering space from deleted nodes, and so on. After optimizing a topic set, you must update any topic indexes that depend on that topic set.
`-outline` *`file`*`.otl`	The full or relative path and name of the outline file from which the new topic set will be built. Use `.otl` as the filename extension.
`-precres`	The optional `-precres` argument specifies that topic precedence checking will occur when the topic set is built or updated. This argument is the default. Only meaningful when used with `-outline`. For information about topic precedence ratings, refer to "Operator Precedence Rules."
`-quiet`	The optional `-quiet` argument suppresses status messages. By default, `mktopics` runs in verbose mode.
`-reset`	The optional `-reset` argument deletes and replaces an existing topic set with an empty topic set. This option does not update a topic set. **Note:** The -reset argument temporarily increases the .std and .sid files in the topic set directory. The cleanup of old files occurs less frequently because the files might being referenced by multiple collections. This generally does not affect performance.
`-shallow`	The optional `-shallow` argument specifies that a dump of a topic set to an outline file dumps each top-level topic down to the next named topic. Only meaningful when used with `-fullotl`.
`-topic` *`name`*	The optional `-topic` argument is followed by the name of the topic in the specified topic set that you want to export to a topic outline file. This argument must be specified with `-fullotl`.
`-topicset` *`topicset_dir`*	The required `-topicset` argument specifies the name of a new or existing topic set directory, depending on the other `mktopics` syntax supplied. For example, to export un–encrypted topic sets use the following syntax: `[-topicset` *`topname`* `-fullotl` *`exportfilename`*`]`
`-warnundef`	The optional `-warnundef` argument specifies that you will be warned if there are any undefined topics when importing topic definitions from an outline file. Only meaningful when used with `-outline`. This is the default.

Checking Topic Precedence Rules

Topics must be structured according to a set of precedence rules. These rules state that certain operators have precedence over others, and some operators cannot be used to define child topics depending on the parent topic's definition. For complete information about topic precedence rules and how to define topics, refer to "Precedence Evaluation" on page 39 and "Elements of Topic Design" on page 93.

By default, `mktopics` does topic precedence checking and resolution when it builds a topic set. If topic precedence errors are found by `mktopics`, then error messages are reported and the topic set will not be built.

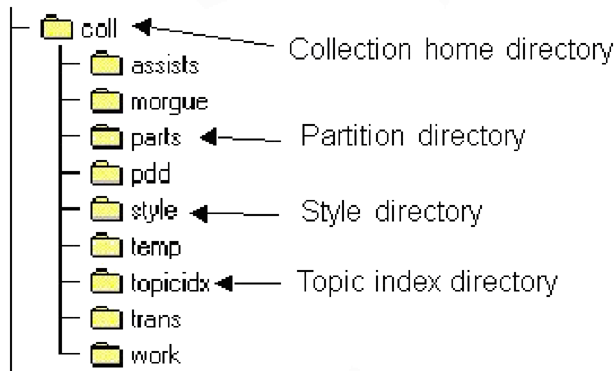
If you want to override the default behavior, you can use the `mktopics -noprecres` argument. When `-noprecres` is used, no precedence checking occurs when the topic set is built or updated, and the checking is done when the Verity search engine processes a topic query at search time. If a precedence rule violation is found, then the Verity search engine recompiles the query automatically. For example, a precedence rule violation will occur if you use the `ANY` or `ALL` operator in a parent topic and a child topic includes one of the concept operators such as `AND`, `OR`, `ACCRUE`. This violation will occur because the `ANY` and `ALL` operators cannot have variable weights assigned to them. The operators `AND`, `OR`, `ACCRUE` allow for variable weights.

If you index a topic set over a collection, using the `-collection` and `-noprecres` options, and the engine encounters precedence problems when processing the topic query, the topic with the precedence problem will not be written to the topic set index. This means that when a topic set is used, any query that uses a topic with a precedence problem needs to be recompiled at search time to resolve the precedence problem. This recompile needs to occur every time the topic is used and it slows down the search performance. For this reason, it is recommended that you always use the default `-precres` behavior, unless you do not plan to index the topic set against a collection.

When running `mktopics`, if you get many precedence errors with the default `-precres` behavior, you can use the `-logfile` argument to save the errors into a log file, so that you can review the errors at a later time when you are correcting the topic outline file.

Topic Set Indexing

A topic set can be indexed against a collection to shorten search and retrieval time. To index a topic set against a collection, you specify the `-collection` argument with the `mktopics` command. When a topic set is indexed against a collection, the collection size will increase to include the topic information. The topic index is stored in a subdirectory of its root collection.



Two types of topic set indexes, “normal” and “named only,” can be built by `mktopics` using the `-collection` argument. The normal index, which `mktopics` builds by default, takes less time to build and uses less disk space than the named only index. When a normal index is built, the search and retrieval performance varies according to the topic structure. By contrast, when a named only index is built, the search and retrieval performance is more consistent among similar sized topics and considerably faster.

The difference between the normal and named only index types has to do with which topics are indexed. For a normal index, all topics with a precedence rating of incremental or lowest are indexed. For a named only index, only named topics are indexed.

A topic set can be indexed against a collection using only one of the two available index types. After a topic set index has been created, you cannot change the index type. Using the `-reset` argument with `mktopics` does not change the index type for a topic set.

Topic Set Encryption

A topic set can be encrypted.

To set up topic set encryption, you create the encryption key file and place it where you will remember it. You can create a directory called `encryption files` and place it there for retrieval later. Then you create and encrypt the topic set using `mktopics`. The sections below describe how to set up and maintain encrypted topic sets.

If you perform back-to-back `mktopics`, both commands are executed without any errors, but only the key from the first command is retained in the topic set.

For example, if you type:

```
mktopics -topicset  t1 -outline 1.otl  -encrypt ekey1
mktopics -topicset  t1 -outline 2.otl  -encrypt ekey2
```

`t1` will be encrypted with key `ekey1`.

Before You Begin

In order to build an encrypted topic set, you must use the command-line tools `mkenc`, and `mktopics`. You must also have an `.otl` file that you can use to create the encrypted topic set from. If you do not have an `.otl` file, you can open a classification tool, such as Intelligent Classifier, and export a topic set to an `.otl` file.

The following exercise shows you how to create and encrypt a topic set using Intelligent Classifier. Before starting, you should decide where you are going to keep the encryption file. For this exercise create a directory in the `samples` directory of the Intelligent Classifier installation. For a default installation the directory should be located here:

`C:\program files\verity\intelligent classifier\samples\`*`encryption files`*

Where *`encryption files`* is the new directory within the samples directory.

For a custom installation the path will be:

`install_dir``\verity\intelligent classifier\samples\`*`encryption files`*

Where `install_dir` is the installation directory for Intelligent Classifier, and `encryption files` is the new directory within the samples directory.

Creating an Encryption File

1. Open a new (DOS) command window.
2. Change directory to the encryption files directory. The reason for this is when you invoke the `-out` command, it creates the file in the directory shown in the (DOS) command window. Your directory should look like this:

```
C:\program files\verity\intelligent classifier\samples\
encryption files>
```

3. Type in the following command (including the quotes around the description) to create the encryption key file. For this exercise the syntax has been chosen for you; however, you can change the filename, key name, and description when you create your own encryption files.

```
mkenc -out encfile -key bardic -desc ‘to be or not to be’
```

4. Press **Return**.

   If the command was typed in correctly, you should get the following text after pressing Return:

```
mkenc done - encryption file encfile
Created key file on Tue Jan 22 11:32:06 2002
Create a backup of this file and keep it in a safe place
```

   If you receive an error message, retype the command checking that the syntax is correct.

Encrypting a Topic Set

This section discusses how to encrypt a topic set using the `tutorial_taxonomy` topic set shipped with Intelligent Classifier. You do not have to save the new encrypted topic set to the same directory; the new encrypted topic set will not overwrite the older one. For this command it is necessary to point to the encryption file to ensure proper encryption of the topic set. Create this topic set in the `samples` directory as follows:

1. Change the directory to `samples`. When you are done, the prompt window should show this directory structure:

```
C\:program files\verity\intelligent classifier\samples>
```

2. Type in the following command:

```
C:\Program Files\Verity\Intelligent Classifier\samples>mktopics
-topicset tutorial_taxonomy -outline tutorial_taxonomy.otl
-encrypt "C:\program files\verity\intelligent classifier\samples\
encryption files\encfile"
```

   Remember that `C:\Program Files` is the default installation directory. Your installation might have been installed elsewhere; just substitute your directory path for the default path.

Note This command is written as a one-line command. The only return should be at the end of the command.

3. Press **Return**. You should see the following message:

```
mktopics - Verity, Inc. Version 6.0.0 (_nti40, Jul 12 2005)
Using Message Database ..\common\english\vdk30.rsd from product
install area ..
Warn    E3-0327 (Vdk Info): Topicset directory tutorial_taxonomy
does not exist.  Creating it
Building system topic dataset: tutorial_taxonomy/00000000.std
             from text topics: tutorial_taxonomy.otl
Reading
Parsing
Loading Pass 1
Load pass 2 not necessary -- skipping
Index Layout
mktopics done
```

If you receive an error message, retype the command checking that the syntax is correct. Notice that `mktopics` knows that the topic set directory `tutorial_taxonomy` does not exist, and therefore creates one for you.

To view what an encrypted topic set looks like in Intelligent Classifier:

1. Open Intelligent Classifier.
2. If the Wizard starts, click **Cancel**.
3. Select **File | Open Workspace**.
4. Browse to and select **tutorial_taxonomy**.
5. Click **Open**.
6. Select **Topic | Open Topic Set | Topic Set**.

7. Browse to the samples directory and select **tutorial_taxonomy**.

If you test the topic PR, you see that it returns the same documents as the unencrypted topic set.

Encryption allows only the top-level topics to be seen by the end–user. The topics cannot be added to or modified in any way. You now have a secure topic set that has a protected design.

Table 8-2 lists the elements and descriptions for `mkenc`.

Table 8-2 mkenc Syntax Elements

Syntax element	Description
mkenc	The name of the command-line tool used to generate the encryption key file.
-out *filename*	The output encryption key file. The encryption key file must be named based upon the level of encryption desired.
[-key *mykey*]	The key for the encryption key file. The key must be assigned a name, or any combination of letters and numbers. The level of encryption is based upon the character string length. For example, if you want to encrypt the key in 40–bit, you would use a character string of no less than 5 characters. If you want 128–bit encryption, you would use a character string of no less than 15 characters.
[-128]	A flag used to specify 128-bit encryption. If not specified, 40-bit encryption is used.
[–desc (optional)]	You can use the description to identify different encryption keys. The description is created in the encryption file, and can be viewed with a text editor. **Note:** You can only add to the description in the encryption file. If you want to change the encryption level of a topic set you must create a new encryption file with that level, or use an already created one to generate the new encrypted topic set. It is not recommended to edit the encryption file, so decide beforehand what the description will be for that file.

APPENDIXES

- **Appendix A: Query Parsers**
- **Appendix B: Query Limits**
- **Appendix C: Creating a Custom Thesaurus**

Query Parsers

A user's query is interpreted by one of five predefined query parsers included with Verity products. The simple query parser is very popular because it allows users to enter simple words and phrases separated by commas, with or without additional query language.

This appendix covers the following topics:

- Simple Queries
- Simple Query Parser
- Query-By-Example (QBE) Parser
- Internet-Style Parser
- BooleanPlus Parser
- Using Query Parsers Programatically

Simple Queries

This section introduces how to write queries for interpretation by the simple query parser.

Words and Phrases Separated by Commas

A simple query is specified as words and phrases, separated by commas. To see documents about using text editors to create Web documents, start with a single-word query, such as:

```
editor
```

Your query finds all the documents that include the word "editor." However, this search would include not only documents about text editors, but also documents about people who are editors. (You don't have to specify the plural form, because a simple search includes stemmed variations, such as "editors.") Documents about the Web that did not include the word "editor" would not be retrieved.

For more specific results, enter several words or phrases, separated by commas, that describe the subject more precisely, such as:

```
text editor, document, web
```

Case-Sensitivity

The search engine attempts to match the case-sensitivity provided in the query expression, when mixed case is used. For search terms entered completely in lowercase or uppercase, the search engine looks for all mixed-case variations.

Search terms with mixed case automatically become case-sensitive. For example, the query of Apple behaves as if you had specified <case>Apple (which would find only the precise string Apple), while the query of apple finds all of the following: APPLE, Apple, apple.

The CASE modifier preserves case-sensitivity of the query. For example, if you want to search for the term "OCX" and want to find instances of "OCX" in uppercase only, you could enter this query:

```
<CASE> <WORD> OCX
```

The search engine would interpret the previous query expression to mean: find all documents containing one or more instances of the word "OCX" spelled in uppercase, not mixed case.

How to Search Hyperlink Contents

Using the Verity operators `IN` and `WHEN`, you can search for all documents that refer to a particular HTML document by following HREF links in the source document. The following syntax can be used:

```
* <IN> A <WHEN> HREF <SUBSTRING> searchterm
```

The previous query is evaluated in distinct query segments, as follows:

Query segment	Interpreted as
* <IN> A	The expression including the IN operator evaluates any contents (*) in the A tags (zones).
<WHEN> HREF <SUBSTRING> searchterm	The expression including the WHEN operator further qualifies the query for a specified HTML attribute, in this case HREF. The *`searchterm`* variable is a word or phrase. The SUBSTRING operator matches the character string you specify with strings in the target HREF.

The SUBSTRING operator can be substituted with the CONTAINS or MATCHES operator. These three operators have different ways of performing string comparisons. For more information about the operators, see "Operators" on page 47.

Simple Query Parser

The simple query parser supports searching over the full text of documents in addition to searching over collection fields and zones. Sometimes the simple parser is referred to as the "full text" parser. The simple parser interprets Verity query language.

A unique feature of the simple query parser is that it can translate a query expression supplied by the user into a more robust query form without requiring the user to specify a lot of syntax. For example, if a user enters a single word, the simple query parser applies the MANY modifier and STEM operator to the word by default. This more robust query form, specifically "<MANY> <STEM> word" causes the search engine to search for a broader range of documents containing evidence of the user's query.

Behaviors of the simple query parser are described in the following list.

- An individual word is interpreted as a stemmed word or a topic name, unless the word is surrounded by double quotation marks. When processing the search, the

search engine first checks to see whether the word matches a topic name, and if a match is found, the topic is used. If a match is not found or if topics are not implemented, the word is interpreted as a stemmed word. The MANY modifier and the STEM operator are applied to a single word.

- When a word is interpreted as a stemmed word, the search engine broadens the search to include the word itself along with the stemmed variations of the word. For example, if a user enters the word "meet" (without double quotation marks, as in: meet) in the search form and then starts a search, the search engine will look for these words: "meet," "meets," and "meeting."
- When matching a query expression (including two or more words) against topic names, spaces are interpreted as hyphens. This means that if a phrase named BIG COMPANIES is supplied as part of a query expression, the application looks for the topic BIG-COMPANIES. If a topic name match is found, the topic is used to process the search.
- To specify a literal word so that words that have the same stem will not be considered in the search, the user can surround the word with double quotation marks. For example, to search for documents that contain the word "tropic" and not consider words that have the same stem, such as "tropics" or "tropical," the word "tropic" needs to be surrounded with double quotation marks.
- Double quotation marks in VQL represent the `<WORD>` operator. They do not affect case sensitivity in a search.
- The PHRASE operator is applied to a phrase where a phrase is defined to be two or more words separated by spaces.
- Queries are case-insensitive when the query terms are entered in lowercase or uppercase characters. Queries are case-sensitive when the query terms are entered in mixed case characters. To force case-sensitive searches for words and phrases, users can use the CASE modifier in queries.
- Special meaning is assigned to the following words in a query expression: AND, OR, NOT. These words are interpreted as Verity query language, unless they are enclosed in double quotation marks.

  For example, to search for the phrase "recycle and reuse," ensuring that the word "and" is not interpreted as an operator, the following query can be used:

  ```
  recycle "and" reuse
  ```

  Note Special characters such as "&" and "|" must also be enclosed in quotes.

- The Verity query language can be used to perform zone and field searches. The zones that are available for searching depends on the type of documents in the collection.

Query-By-Example (QBE) Parser

The query-by-example (QBE) parser supports searching for similar documents, a search method sometimes referred to as similarity searching. The QBE parser supports searching over the full text of documents only. The QBE parser does not support searching over collection fields and zones. The QBE parser does not support Verity query language except topics.

IMPORTANT The Verity products and documentation also refers to the QBE parser as the Free Text Parser.

Meaningful words are automatically treated as if they were preceded by the MANY modifier and the STEM operator. By implicitly applying the STEM operator, the search engine searches not only for the meaningful words themselves, but also for words that have the same stem. By implicitly applying the MANY modifier, the search engine calculates each document's score based on the word density it finds for meaningful words; the denser the occurrences of a word in a document, the higher the document's score.

By default, common words (such as "the," "has," and "for") are stripped away, and the query is built based on the more significant words (such as "personnel," "interns," "schools," and "mentors"). Therefore, the results of a query-by-example search are likely to be less precise than a search performed using the simple or BooleanPlus parser.

The QBE query parser interprets topic names as topic objects. This means that if the specified text block contains a topic name, the query expression represented by the topic is considered in the search.

Internet-Style Parser

With the internet-style query parser (IQP), users can search entire documents or parts of documents (zones and fields) using a command syntax similar to the syntax used in many Web search engines.

Search Terms

In a search form enabled with the internet-style query parser, users can enter words, phrases, and plain language. The internet-style parser does not support the Verity query language (VQL).

However, if you are developing an application using the Verity Developer's Kit Application Programming Interface (VDK API), you can combine VQL and IQP syntax.

See the section "Using Query Parsers Programatically" on page 145 for more information.

Words

To search for multiple words, separate them with spaces.

Phrases

To search for an exact phrase, surround it with double quotation marks.

A string of capitalized words is assumed to be a name. Separate a series of names with commas.

Commas aren't needed when the phrases are surrounded by quotation marks.

The following example searches for a document that contains the phrases "San Francisco" and "sourdough bread".

```
San Francisco "sourdough bread"
```

Plain Language

To search with plain language, enter a question or concept.

The Verity internet-style Query Parser identifies the important words and searches for them. For example, enter a question such as:

```
Where is the sales office in San Francisco?
```

This query produces the same results as entering:

```
sales office San Francisco
```

Including and Excluding Search Terms

You can limit searches by excluding or requiring search terms, or by limiting the areas of the document that are searched.

A minus sign (–) immediately preceding a search term (word or phrase) excludes documents containing the term.

A plus sign (+) immediately preceding a search term (word or phrase) means returned documents are guaranteed to contain the term.

If neither sign is associated with the search term, the results may include documents that do not contain the specified term as long as they meet other search criteria.

Search Scope

The internet-style parser supports searching over the full text of documents in addition to document zones and fields.

Zone Searches

The internet-style parser allows users to perform zone searches. Zones that are available for searching depend on the type of documents in the collection.

Zones are available in Markup Language documents (such as HTML and SGML) as well as Internet Message format documents (such as standard email and Usenet newsgroup messages).

To search a document zone, type the name of the zone, a colon (:), and the search term with no spaces.

```
zone:term
```

If you enter a minus sign (–) immediately preceding `zone`, documents containing the specified term will be excluded from the search results. For example, if you enter `-zone:term`, documents containing `term` are excluded from the results of the search of `zone`.

If you enter a plus sign (+) immediately proceeding the zone search specification, such as `+zone:term`, documents are included in the zone search results only if the term is present.

Field Searches

The internet-style parser allows users to perform field searches. The fields that are available for searching depend on field extraction rules based on document type of documents in the collection.

To search a document field, type the name of the field, a colon (:), and the search term with no spaces.

```
field:term
```

If you enter a minus sign (–) immediately proceeding `field`, documents containing the specified term will be excluded from the search results. For example, if you enter `-field:term`, documents containing the specified term in the specified field will be excluded from the results of the search.

If you enter a plus sign (+) immediately proceeding the field search specification, such as `+field:term`, documents will be included in the search results only if the search term is present in the specified field.

Field searches are enabled by the `enableField` parameter in a template file. This parameter, set to 0 by default, must be set to 1 to allow searching a document field.

The `enableField` is the only thing in a template file that should be changed without prior consultation with Verity Technical Support.

Template Files

Template files are located in a locale-specific subdirectory of your Verity `installdir`.

- If you have a K2 system that uses the `uni` locale, the templates are located in the `<installDir>/k2/common/uni` subdirectory.
- If you have a VDK system that uses the `uni` locale, this directory is located in the `<installDir>/vdk/common/uni` subdirectory.

Template files allow the Verity engine to balance the performance of query parsing with the goal of increased relevance.

It also means you can customize the Verity engine to provide a query for different kinds of documents.

Table A-1 lists the templates.

Table A-1 Templates

Template Name	Filename	Description	Use
`Internet_Basic`	`basic.iqp`	Leverage the title and location information from the document to boost relevancy-ranking.	Minimize search time
`Internet_BasicWeb`	`basicweb.iqp`	In addition to Internet_Basic ranking, uses summarization, keywords, and anchor text information.	Search documents with anchor text
`Internet_Advanced`	`advanced.iqp`	In addition to Internet_Basic ranking, uses summarization, keywords and document formatting information.	Documents are mostly WYSIWIG document types
`Internet_AdvancedWeb`	`advweb.iqp`	In addition to Internet_Advanced ranking, uses link analysis information to boost relevancy.	Search targets are mostly HTML documents
`Internet`	`legacy.iqp`		Backward compatibility

The **Template Name** is the identifier that is used to specify a template in the **`rcvdk`** command-line tool; in the VDK API, it is also the value returned by the `VdkQParserGetInfo` function, as demonstrated in the section "Using Query Parsers Programatically" on page 145.

Use the `Internet_Advanced` template except under the following circumstances:

- If most searches will be directed at HTML documents, use the `Internet_AdvancedWeb` template.
- If search performance is unacceptable, use the `Internet_Basic` template.

Note To use the `Internet_BasicWeb` and `Internet_AdvancedWeb` templates, the collection must be created using K2Spider with the `CollectLinkInfo` and `CollectAnchorText` parameters set to `true` in the jobs file. For more information about setting these parameters, see the *Verity Command-Line Indexing Reference*.

Query Syntax

The query syntax is very similar to the syntax users expect to use on the Web. All queries produce valid results; therefore, be careful to form your query to produce the results you expect. Queries are interpreted according to the following rules:

- Individual search terms are separated by whitespace characters, such as a space, tab, or comma, as in the following example:

  ```
  cake recipes
  ```

- Search phrases are entered within double quotes, as in the following example:

  ```
  "chocolate cake" recipe
  ```

- Exclude terms with the negation operator, minus (-), or the `not` operator, as in the following example:

  ```
  cake recipes -rum
  ```

  or

  ```
  cake recipes not rum
  ```

- Require a compulsory term with the unary inclusion operator, plus (+), as in the following example:

  ```
  cake recipes +chocolate
  ```

  This example requires the term `chocolate` to be present.

- Require compulsory terms with the binary inclusion operator, `and`, as in the following example:

  ```
  cake recipes and chocolate
  ```

  This example requires both the term `recipes` and `chocolate` to be present.

Zone and Field Searches

You can search fields or zones by specifying *`name`*`:`*`term`*, where *`name`* is the name of the field or zone and *`term`* is an individual search term or phrase, such as

```
bakery city:"San Francisco"
```

or

```
bakery city:Sunnyvale
```

Pass-Through of Terms

Search terms are passed through to the VDK-level and are interpreted as Verity Query Language (VQL) syntax. No issues arise if the terms contain only alphabetic or numeric characters. Other kinds of characters may be interpreted by the locale. If a term contains a character that is not handled by the locale, it may be interpreted as VQL; for example, a search term that includes an asterisk (`*`) would be interpreted as a wildcard.

Stop Words

The configurable Internet-style query parser uses its own stop-word list, `qp_inet.stp`, to specify terms to ignore for natural language processing.

Note You can override the “stop out” by using quotation marks around the word.

For example, the following stop words are provided in the query parser’s stop word file for the `english` locale:

a	did	i	or	what
also	do	i’m	should	when
an	does	if	so	where
and	find	in	than	whether
any	for	is	that	which
am	from	it	the	who
are	get	its	there	whose
as	got	it’s	to	why
at	had	like	too	will
be	has	not	want	with
but	have	of	was	would
can	how	on	were	<or>

Verity provides a populated stop-word file for the `english` and `uni` locales; you need not modify the `qp_inet.stp` file for these locales. If you use the configurable Internet-style query parser for another locale, you must provide your own `qp_inet.stp` file containing the stop words you want to ignore in the locale. This stop word file must contain, at a minimum, the locale-equivalent words for `or` and `<or>`.

Note The configurable Internet-style query parser's stop word file contains a different word list than the `vdk30.stp` word file, which is used for other purposes, such as summarization.

Testing the Templates

To see which templates are available to you, run the `rcvdk` command-line tool.

After the tool is loaded, enter the `x` command to enable expert mode, then enter the `qparser` command to see the list of query parsers currently available. In the following example, the tool is run from the `colls` subdirectory of a K2 data directory, and the example collection `verity_doccoll` is specified:

```
host:/users/user> rcvdk verity_doccoll
rcvdk  Verity, Inc. Version 5.0.0
Attaching to collection: verity_doccoll
Successfully attached to 1 collection.
Type 'help' for a list of commands.
RC> x
Expert mode enabled
RC> qparser
Available query parsers:
  Name                 Description
  ----                 -----------
  Simple               Simple Query Syntax
  BoolPlus             BooleanPlus Query Syntax
  Prefix               Explicit Prefix Query Syntax
  FreeText             Natural Language Query Syntax
  Internet_Basic       Basic Relevance Factors.
  Internet_BasicWeb    Basic Relevance Factors with Anchor Text.
  Internet_Advanced    Advanced Relevance Factors.
  Internet_AdvancedWeb Advanced Relevance Factors for Web Gateway
                       Collections
  Internet             Legacy Internet Query Parser.
RC>
```

To select a query parser, enter the `qparser` command with the name of the parser:

```
qparser Internet_Basic
```

BooleanPlus Parser

The BooleanPlus query parser supports searching over the full text of documents in addition to searching over collection fields and zones. Sometimes the BooleanPlus parser is referred to as the “explicit” parser.

The BooleanPlus parser is similar to the simple parser in that it interprets all of the Verity query language and can interpret field and zone searches. Unlike the simple parser, queries can not be interpreted using the BooleanPlus parser unless explicit query syntax is used. For this reason, the BooleanPlus query parser typically is not used in end user search forms.

Using Query Parsers Programatically

If you are developing your own application, you can use the Verity Developer’s Kit Application Programming Interface (VDK API) to access query parsers, and to combine VQL syntax with the Internet Query Parser (IQP) syntax.

Obtaining a Query Parser Using the VDK API

The following example shows how to use the `VdkQParserGetInfo` API function to obtain the name and description of a query parser and how to obtain a handle to a query parser, given its name.

This example also shows how to use the `VdkQParserGetInfoFree` function.

```
/*------------------------------------------------------
 * This example demonstrates how to list available
 * query parsers.
 *------------------------------------------------------*/
VdkError ListQParsers(VdkSession session)
{
  VdkError            error = VdkSuccess;
  VdkSessionGetArgRec sesArg;
  VdkSessionGetOut    sesOut;
  VdkQParserGetArgRec qpArg;
  VdkQParserGetOut    qpOut;
```

```
  VdkInt2             i;

  VdkStructInit(&sesArg);
  sesArg.requestQparserBase = VdkFlag_On;

  if (error = VdkSessionGetInfo(session, &sesArg, &sesOut))
    goto abort;

  printf("Available Query Parsers:\n");

  for (i = 0; i < sesOut->qparserBaseCount; i++)
  {
    VdkQParser qp = sesOut->qparserBaseArray[i];
    VdkStructInit(&qpArg);

    if (error = VdkQParserGetInfo(session, qp, &qpArg, &qpOut))
      goto abort;

    /* print out the name and description of each parser */
    printf("  %-20s %s\n", qpOut->name, qpOut->description);

    VdkQParserGetInfoFree(qpOut);
  }
  VdkSessionGetInfoFree(sesOut);

abort:
  return error;
}

/*------------------------------------------------------
 * This example demonstrates how to get the handle of
 * a query parser given a string.
 *------------------------------------------------------*/
VdkError StrToParser(
  VdkSessionsession,
  VdkCString  name,
  VdkQParser *parser)
{
  VdkError            error = VdkSuccess;
  VdkSessionGetArgRec sesArg;
  VdkSessionGetOut    sesOut;
  VdkQParserGetArgRec qpArg;
  VdkQParserGetOut    qpOut;
  VdkQParser          found = NULL;
```

```
  VdkInt2             i;

  VdkStructInit(&sesArg);
  sesArg.requestQparserBase = VdkFlag_On;

  if (error = VdkSessionGetInfo(session, &sesArg, &sesOut))
    goto abort;

  for (i = 0; i < sesOut->qparserBaseCount && !found; i++)
  {
    VdkQParser qp = sesOut->qparserBaseArray[i];
    VdkStructInit(&qpArg);

    if (error = VdkQParserGetInfo(session, qp, &qpArg, &qpOut))
      goto abort;

    if (!stricmp(qpOut->name, name))

    /* found a parser */
    found = qp;

    VdkQParserGetInfoFree(qpOut);
  }

  VdkSessionGetInfoFree(sesOut);

 abort:
  if (found)
    printf("Located query parser %s\n", name);
  else
    printf("Did not locate query parser %s\n", name);
  *parser = found;
  return error;
}
```

Using VQL with the Internet Query Parser

You can use the VDK API to combine the functionality of VQL with the IQP as follows:

- Set the `queryQParser` member of a `VdkSearchNewArgRec` data structure equal to one of the Internet Query Parsers. See the example code in the section "Obtaining a Query Parser Using the VDK API" on page 145.

- Set the `queryQuestion` member of the data structure equal to a query using Internet Query Parser syntax, such as `red +apple +computer`.
- Set the `sourceQParser` member of the data structure to a query parser that supports VQL.
- Set the `sourceQuestion` member of the data structure to a VQL expression, such as `Date >01-01-2003`.
- Call the API function `VdkSearchNew`.

The following snippet of C code demonstrates this sequence. The `question` argument contains a query using IQP syntax, while the `source` argument contains an additional query criteria using VQL syntax. The `qryParser` argument contains the IQP parser. The example is hard-coded to use the VDK API designation for the Simple parser as the `sourceQParser` value.

```
void UseSourceQuery(
                    VdkSession sess,       /
* from VdkSessionNew */
                    const char* source,    /* Source query     */
                    VdkQParser qryParser,  /* Query Parser     */
                    const char* question)  /* ASCII string     */
{
  VdkSearchNewArgRec searchNew;
  VdkSearch search;
  VdkError error;

  VdkStructInit(&searchNew);
  searchNew.maxDocs        = 100;
  searchNew.queryQuestion  = (VdkCString)question;
  searchNew.queryQParser   = qryParser;
  searchNew.sourceQuestion = (VdkCString)source;
  searchNew.sourceQParser  = VdkQParser_Simple;

  error = VdkSearchNew(sess, &search, &searchNew);
  if (error == VdkSuccess)
  {
    /*------------------------------------------------
     * Display the results of the search.
     * When  done with the search object, free it.
     *-----------------------------------------------*/

    VdkSearchFree(search);
  }
}
```

B

Query Limits

The overall limit on the size of a topic set is 5 million nodes and 8 million links. In addition, there are some search-time limitations on the size of a single topic. These limits apply to the topic that is built from the query you type in, which may be a combination of query terms and predefined topics from a topic set.

This section covers:

- Search Time Limits
- Operator Limits

Search Time Limits

Search-time limitations are combinations of implementation limitations of various portions of the search engine, rather than a simple limit on the physical number of nodes or links allowed.

The Verity search engine includes the concept that topics represent search terms. Queries that go beyond a single word or phrase typically involve the ACCRUE-class operators (ACCRUE, AND, OR) to combine several branches of evidence in a topic tree. At search time, the combined evidence is evaluated by a stack-based engine.

The stack engine imposes some restrictions for ACCRUE-class topics. Its limited stack space imposes the restriction of 1,024 children for any single ACCRUE-class node and about 5,300 total notes (16,000/3 to be precise) in a topic. The engine detects these limits while building a query and returns an error if they are exceeded.

You can work around the limit on total nodes in a topic by creating named subnodes in the topic and building the topic set with this `mktopics` option:

```
-indextype namedonly
```

Using this option causes separate queries to be built for each named subtopic, resulting in fewer nodes for each query. Carrying this process to the extreme, however, can reduce the effectiveness of the topic index for the top-level topic.

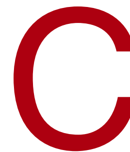
Operator Limits

Note the following limits on the use of operators:

- There can be a maximum of 32,764 children for the ANY operator. If a topic exceeds this limit, the search engine does not always return an error message.
- The NEAR operator can evaluate only 64 children. If a topic exceeds this limit, the search engine does not return an error message.

For example, assume you have created a large topic that uses the ACCRUE operator with 8365 children. This topic exceeds the 1024 limit for any ACCRUE-class topic and the 16000/3 limit for the total number of nodes.

In this case, you cannot substitute ANY for ACCRUE, because that would cause the topic to exceed the 8,000 limit for the maximum number of children for the ANY operator. Instead, you can build a deeper tree structure by grouping topics and creating some named subnodes.



Creating a Custom Thesaurus

Synonym search is a type of search that locates occurrences of either the search term or any of its synonyms. For example, a synonym search for **brave** might return documents that contain **brave** or **courageous** or **fearless**. A search application specifies a synonym search by adding the VQL `THESAURUS` operator to the user's search term.

Synonym search requires the use of a *thesaurus* file, which lists groups of synonyms. Verity K2 includes a default English thesaurus that may be adequate for most purposes in English. To construct a thesaurus for use in other locales, or to create a custom English thesaurus, follow the instructions in this appendix.

The `THESAURUS` operator is described in "THESAURUS" on page 59.

This appendix includes the following sections:

- Creating a Thesaurus Control File
- Compiling a Thesaurus with mksyd
- Integrating the Thesaurus with Verity

Creating a Thesaurus Control File

A Verity thesaurus is a compiled file with a `.syd` extension. To create or modify a thesaurus, you need to first create or edit a text file called a *thesaurus control file*, which has a `.ctl` extension. You then compile the control file into a locale-specific thesaurus file with the `mksyd` command-line tool.

When creating a thesaurus, you can

- Create a complete control file using a text editor.
- Edit an existing control file to add or remove synonyms.
- Purchase a commercial thesaurus, then turn it into a thesaurus control file by adding the statements described here.

Note You can recreate a control file from a thesaurus; see "Creating a Control File from an Existing Thesaurus" on page 154.

Control-File Structure

A thesaurus control file contains synonym lists. Each list is defined by the `list` keyword. The list contains synonyms and, optionally, *keys*. Keys are words that must appear in the search term for the synonym list to be used. In other words, if a search term consists of one of the non-key words in a synonym list, the term itself is searched for, but none of its synonyms is. A list with specified key terms is an *asymmetric* list.

If a given list has no keys, every synonym in the list is considered a key, and the list is *circular*.

The following is an example of a small thesaurus control file.

```
$control:1
synonyms:
{
list: "abort,miscarry,terminate,halt,end,fail"
list: "cease,stop,desist,terminate,end,discontinue"
list: "karma <or> fate <or> destiny"
  /keys = "karma"
}
```

The first two lists are circular; the third is asymmetric. A synonym search for any term in the first list, for example, will locate that term plus any of its synonyms. Likewise, a synonym search for **karma** will find all occurrences of **karma**, **fate**, or **destiny**. However, a synonym search for **fate** will find only occurrences of **fate**.

If a key word (explicit or implicit) appears in more than one list, all lists for which it is a key are included in the synonym search. For example, note that the words **terminate** and **end** are keys in two lists in this example. In this case, a thesaurus query for either **terminate** or **end** results in an expanded query containing both lists:

```
"(cease,stop,desist,terminate,end,discontinue) <or>
(abort,miscarry,terminate,halt,end,fail)"
```

A list can be more than a simple comma-separated set of terms. Note that the third list in this example includes the query expression `"karma <or> fate <or> destiny"`. You can use query expressions in a thesaurus control file to apply sophisticated search logic to synonyms or to override default the default query expansion of synonym lists. See "The qparser Keyword" on page 154 for more information.

Note A thesaurus definition cannot contain punctuation defined by the locale configuration files `loc0.lng` and `separator.cfg`. For example, the term **R.F.P**. must be defined in the thesaurus as **RFP**. Otherwise, the definition is split into the simple tokens **R**, **F**, and **P**.

The control Directive.

The `$control:1` directive must be the first non-comment line in the control file.

The synonyms Keyword

The `synonyms` keyword is required in a thesaurus control file. It must appear directly after the `$control:1` directive.

The list Keyword

The `list:` keyword specifies the synonyms in a list, either in query form or in a list of words or phrases separated by commas. The optional modifier `/keys` specifies the keys list, which must be a list of words separated by commas. If `/keys` is absent, all synonyms in the list become keys. The optional modifier `/op-default` defines the fallback operator to use if there is no match for a thesaurus query.

The maximum length for a single list is 32,000 characters.

Note If you separate your list into multiple lines (inserting new lines), you must include a backslash (\) at the end of each line so that the lines are treated as one list.

The following is a sample `list` statement:

```
list:"happy, joyous, joyful, glad, blithe, merry,\
cheerful, contented, blissful, delighted, satisfied,\
pleased, favored, lucky, fortunate, propitious,\
appropriate, felicitous, befitting"
```

The qparser Keyword

The synonym lists in a thesaurus control file are parsed and expanded as queries when the thesaurus is created.

The default expansion applied during thesaurus creation is different from the default expansion applied to user queries by applications that use the simple query parser. For example, the simple query parser expands a list of words separated by commas (the default combination operator) by applying the `<ACCRUE>` operator to the list. In default thesaurus query expansion, however, the comma-separated list is expanded by applying the `<ANY>` operator to it.

The following table lists the default values for expansion operators during thesaurus creation.

Type of Expansion	Operator in thesaurus creation	Comment
leaf operator	`<STEM>`	Synonyms are stemmed for searching
combination operator	`<ANY>`	Synonym searches are not ranked
phrase operator	`<PHRASE>`	Phrases are searched as phrases

To make sure that the same expansion operators are used during thesaurus expansion as are used during search, you can use the `qparser` keyword in your control file to specify a query parser. For example:

```
qparser: simple
```

Creating a Control File from an Existing Thesaurus

The `mksyd` command-line tool is primarily used to compile a thesaurus from a control file (see “Compiling a Thesaurus with mksyd” on page 157), but you can also use it to de-compile (export) a thesaurus, turning it back into a control file.

The easiest way to create a custom thesaurus in a locale for which you already have a thesaurus is to export the thesaurus to a text file, modify it, and then recompile it as a .syd file.

Existing thesaurus files are stored in the directory *verity_product*/common/*locale_name*, where *verity_product* is the directory containing the component of Verity that has been installed (for example, usr/verity/K2 for K2 Services), and *locale_name* is the name of the thesaurus's locale.

To use mksyd to create a control file from an existing thesaurus file, execute this command from within the directory that holds the existing thesaurus file:

```
mksyd -locale locale_name -charmap charset -dump -syd vdk30.syd -f
ctrl_file.ctl
```

where

- *locale_name* is the name of the locale whose thesaurus you are de-compiling.

  (This option is not required if the thesaurus is in the default locale.)

- *charset* is the character set you want the control file to use. It must be one of the character sets supported by *locale_name*, as listed in Appendix A of the *Verity Locale Configuration Guide*.

  (This option is not required if the control-file's character set is to be the default character set for *locale_name*.)

- vdk30.syd is the name of the thesaurus file that you want to de-compile.

- *ctrl_file*.ctl is the name you want to give to the control file (note the extension .ctl).

The resulting file is in control-file format:

```
$control: 1
synonyms:
{
  list: "word1, synonym1-1, synonym1-2, synonym1-3"
  list: "word2, synonym2-1, synonym2-2, synonym2-3"
  list: "word3, synonym3-1, synonym3-2, synonym3-3"
...
}
```

You can then edit the control file as needed and re-compile it as explained next.

Using the LANG/ID Modifier in the Thesaurus Control File

You can use the LANG/ID VQL modifier in the thesaurus control file. For example:

```
$control:1
synonyms:
{
  ## the code page must be in UTF8 for uni locale
  ##
  list: "karma <or> fate <or> destiny"
  /keys = "karma"

  ## use multi <lang/id> operators in thesaurus file
  list: "<lang/en> hello <or> <lang/fr> world"
  /keys = "lang"

  ## use <lang/id> for same word in different languages
  list: "<lang/en> en_dog, <lang/fr> fr_dog, <lang/ja> ja_dog"

  ## the <lang/fr> will apply to all items in list, that is,
  ## it's the same as "<lang/fr>test, <lang/fr> testa, <lang/fr>
  teste"
  list: "test, <lang/fr> testa, teste"

  ## if there are more than 2 <lang/id> in list, any item without
  <lang/id> will use
  ## the default lang/id from vdk session, in this sample it's
  "javat"
  list: "<lang/en>java, <lang/fr> jave, javat"

  ## use default lang from Vdk session to apply to all items
  list:
  "Arcadian,bucolic,country,pastoral,provincial,rural,rustic"
  list:
  "Cain,butcher,cutthroat,homicide,killer,murderer,slaughterer,
  slayer"
  list: "Casanova,philanderer,womanizer"
  list: "Goliath,behemoth,giant,mammoth,monster,titan"
  list: "Judas,betrayer,traitor"
  list: "Philistine,barbarian,boor,churl"
  list: "Pollyanna,optimist"
  list: "wrench,wrest,wring"
}
```

Compiling a Thesaurus with mksyd

After you have created a thesaurus control file, you can use the `mksyd` command-line tool to compile it into a thesaurus. The control file must have the file-name extension `.ctl`.

Execute the following command from within the directory that holds the thesaurus control file:

```
mksyd -locale locale_name -charmap charset -f control_file.ctl
-syd vdk30.syd
```

where

- *`locale_name`* is the locale of the thesaurus, which must be the locale of any collections that the thesaurus is to be used with.

  (This option is not required if the thesaurus is in the default locale.)

- *`charset`* is the character set of the control file. *`charset`* must be a character set supported by the locale *`locale_name`*.

  The *`charset`* option is optional; leave it off if the control file's character set is the internal character set of the thesaurus's locale. For a list of the supported character sets and internal character set for each locale, see Appendix A of the *Verity Locale Configuration Guide*.

- *`control_file`* is the name (minus file extension) of the control file to compile.
- *`vdk30.syd`* is the name of the thesaurus file that you want to create.

Integrating the Thesaurus with Verity

After you have created a new thesaurus, placed it in the appropriate directory for use.

Naming and Installing the Thesaurus

First, the thesaurus file must have the appropriate filename. Regardless of its locale, every thesaurus file must be named `vdk30.syd`.

Note Only one active thesaurus file is allowed per locale. Only one `vdk30.syd` file can be present in a *`locale_name`* directory. If you are creating a thesaurus for a locale that has a default thesaurus provided by Verity, move the default thesaurus from the *`locale_name`* directory, or else rename it, before adding your new thesaurus. It is recommended that you do not permanently remove the default thesaurus.

To integrate your custom thesaurus into your search application, move the compiled thesaurus file to the locale's directory:

```
verity_product/common/locale_name
```

where *`verity_product`* is the installation directory (such as `/usr/verity/k2`) of your Verity component, and *`locale_name`* is the name of the directory containing the locale for which you are creating the thesaurus.

WARNING! All application processes, including user searches, must be terminated before you remove or change the contents of the `common` directory or any of its subdirectories. The new thesaurus will be available when the application is started or restarted.

Using a Knowledge Base Map to Point to a Thesaurus File

You can also use a knowledge-base map to point to a `.syd` file. This is a sample map file:

```
$control:1
kbases:
{
kb: "Thesaurus"
/kb-path = "vdk30.syd"
}
```

In K2, point to this map file either through the client in a local context, or through the server configuration file in a remote context.

No thesaurus operator is necessary in queries using a knowledge-base map. The query works like a topic, so any word in the thesaurus that you enter automatically maps to its synonym list.

Index

Symbols

$control
  1 keyword 103
.otl files 96

A

ACCRUE operator 48
advanced.iqp 141
advweb.iqp 141
ALL operator 49
AND operator 50
angle brackets
  delimiters 43
  operator/modifier names 35
ANY operator 50
asterisks (*) as indentation characters 112
automatic case-sensitive searches 35

B

basic.iqp 141
basicweb.iqp 141
BooleanPlus parser 145
bottom-up design 99
braces 43
branches of evidence 95
BUTNOT operator 51

C

CASE modifier 71
category definition rules 29
collections, effect on 56
command line tool
  mktopics 122
comment lines
  topic outline file 107
COMPLEMENT operator 82
concept operators
  ACCRUE 48
  AND 50
  description 24
  OR 55
CONTAINS operator 64
control: 1 keyword
  topic set 107
creating a topic 109
  evidence topics 111
  subtopics 110
  top-level 110

D

database field evidence topics 115
  affect on document scores 117
  defining 115
  placing 117
  relational operators 116
  specifying ranges 106
  using to include and exclude documents 105
DATE field 69
DATE modifier 79
default weight 49
defining topics
  evidence topics 111
  subtopics 110
  top-level 110
definition modifier
  /definition 108
  /wordclass 108
  /zonespec 108
delimiters in expressions 43
document

including and excluding 105
document zone, defined 51
documents, excluding from results list 75
double quotation marks, reserved words 43

E

encryption 127
ENDS operator 65
evidence operators
description 20
SOUNDEX 58
STEM 59
THESAURUS 59
TYPO/N 60
WILDCARD 61
WORD 63
evidence topics 94, 95
abbreviated 114
SOUNDEX operator 114
STEM operator 114
WORD operator 114
assigning the CASE modifier 113
assigning the MANY modifier 113
assigning the NOT modifier 113
assigning weight 113
defining 111
defining database fields 115
relational operators 116
specifying database field ranges 106
excluding documents 75
explicit parser 145
explicit syntax 36
exporting taxonomies 102
expressions
braces 43
delimiters 43
parentheses 42

F

field evidence topic 116
field searches
available fields 139
internet-style query 140
using relational operators 22
filter documents 117
free text parser 137
FREETEXT operator 86

I

IN operator 51
infix notation 42
instance data, searching for 45
internet-style query
- (minus sign) 139
limited queries 138
plain language 138
+ (plus sign) 139
interpretation of a topic name 109

K

kbm files 97
knowledge base 94

L

LANG/ID modifier 72
language-specific search 72
legacy.iqp 141
LIKE operator 87
locale_name metavariable 155
LOGSUM and LOGSUM/n operators 83

M

main functions of topics 96
making topics available 97
MANY modifier 74
MATCHES operator 65

- minus sign (–) 139
- mkenc command line tool 127
- mksyd command-line tool
  - control file from custom thesaurus 155
  - create custom thesaurus 157
- mktopics command line tool 119, 122, 127
  - suppressing status messages 124
  - syntax 122
  - topic precedence checking 125, 126
  - using 122
- modifiers
  - CASE 71
  - DATE 79
  - description 25
  - LANG/ID 72
  - MANY 74
  - NOT 75
  - NUMERIC 78
  - ORDER 76
  - WHEN 77
  - ZONE 78
- MULT/n operator 84

N

- natural language operators
  - FREETEXT 86
  - LIKE 87
- NEAR operator 54
- NEAR/N operator 55
- negative example 87
- negex 87
- NOT modifier 111
- NOT modifier 75
- NUMERIC modifier 78

O

- operator precedence rules 98
- operator types
  - concept 24
  - evidence 20
  - natural language 86
  - proximity 21
  - relational 22
  - score 82
- operator used by a parent topic 114
- operators
  - != (Not Equals) 67
  - < (Less Than) 68
  - <= (Less Than Or Equal To) 69
  - = (Equals) 67
  - > (Greater Than) 68
  - >= (Greater Than Or Equal To) 68
  - ACCRUE 48
  - ALL 49
  - AND 50
  - ANY 50
  - BUTNOT 51
  - COMPLEMENT 82
  - CONTAINS 64
  - ENDS 65
  - FREETEXT 86
  - IN 51
  - LIKE 87
  - LOGSUM and LOGSUM/n 83
  - MATCHES 65
  - MULT/n 84
  - NEAR 54
  - NEAR/N 55
  - OR 55
  - PARAGRAPH 56
  - PHRASE 57
  - PRODUCT 85
  - SENTENCE 57
  - SOUNDEX 58
  - STARTS 66
  - STEM 59
  - SUBSTRING 67
  - SUM 85
  - THESAURUS 59

TYPO/N 60
WILDCARD 61
WORD 63
YESNO 85
operators and modifiers 94
precedence of 39
OR operator 55
ORDER modifier 76
outline file
creating 104
elements 102
comment lines 107
definition modifiers 107
indentation characters 109
outline file elements
control: 1 keyword 107

P

PARAGRAPH operator 56
parent and child relationship 96
parentheses in expressions 42
phrase evidence topics 105
PHRASE operator 57
+ (plus sign) 139
posex 87
precedence of operators 39
precedence rules 39, 98
prefix notation 42
preindexing topic sets 97
preindexing topics 97
PRODUCT operator 85
proximity operators
ALL 49
ANY 50
BUTNOT 51
description 21
IN 51
NEAR 54
NEAR/N 55
PARAGRAPH 56
PHRASE 57
SENTENCE 57
punctuation in queries 45

Q

QBE specification 87
qualify instance data 108
qualify instance data, searching for 45
qualify instance syntax 45
queries
commas between words 134
precedence rules 39
punctuation 45
shorthand notation 37
simple 133
using wildcards 61
queries, internet-style
limited queries 138
minus sign (–) 139
plain language 138
plus sign (+) 139
query expression
defined 34
query parsers
BooleanPlus 145
explicit 145
free text 137
internet-style 137
query-by-example 137
simple 135
query-by-example
description 87
negative example 87
parser 137
quotation marks
double 43
in FreeText 86

R

relational operators 116
- CONTAINS 64
- description 22
- ENDS 65
- MATCHES 65
- STARTS 66
- SUBSTRING 67

rules
- category definition 29

S

score operators
- COMPLEMENT 82
- LOGSUM and LOGSUM/n 83
- MULT/n 84
- PRODUCT 85
- SUM 85
- YESNO 85

score selected documents 117
scoring
- affected by field evidence topics 117

search concepts 95
searches
- assigning weights to search terms 37
- Boolean 24
- excluding documents 75
- excluding stemmed variations and topics 36

SENTENCE operator 57
simple parser 135
simple queries 133
simple syntax 34
SOUNDEX operator 58
STARTS operator 66
STEM operator 59
stem topic 104
stemmed variations
- how to exclude 36
- how to include 34
- language-related 72

storage methods, topic sets 96
strategy, topic design 99
style.lex file 58
subject areas of a topic 95
SUBSTRING operator 67
subtopics 95
- assigning the NOT modifier 111
- assigning weight 105, 111
- defined by the PHRASE operator 114
- defining 110
- using the same in several places 104

SUM operator 85
syntax
- explicit 36
- simple 34

T

templates
- advanced.iqp 141
- advweb.iqp 141
- basic.iqp 141
- basicweb.iqp 141
- legacy.iqp 141

text comparisons 22
<THESAURUS> operator 151
thesaurus
- creating a control file from 155

THESAURUS operator 59
top-down design 99
topic
- creating 109
  - evidence topics 111
  - subtopics 110
  - top-level 110
- defined 28, 94
- definition modifiers 107
  - /annotation 108
  - /author 108

/date 108
/wordtext 108
elements 94
maximum number 149
name case sensitivity 98
name characters 109
name length 98, 109
naming 109
operator precedence 98
outline file
creating 104
elements 102
outline file elements
comment lines 107
control: 1 keyword 107
definition modifiers 107
indentation characters 109
relationships to subtopics 96
size limits 149
specifying names 37
types
evidence topics 94, 95
subtopics 95
top-level 95
topic design strategy 99
topic outline file 102
See .otl files
topic set
defined 29
maximum number of topics 149
replace existing 124
size limits 149
topic set storage methods 96
topic set, definition 94
topic sets
encryption of 127
indexing topics 126
preindexing 97
stored in .otl files 96
stored in directory format 96
topic precedence checking 125
topic subject areas 95
topic weights 105
topic, definition 94
topics 96
benefits 101
best performance 97
inetsrch.ini setting 97
top-level topics 95
top-level topics, defining 110
TYPO/N operator 60

V

vdk30.syd 158

W

weight
assigning to evidence topics 113
assigning to subtopics 111
weight, default 49
WHEN modifier 77
WILDCARD operator 61
word evidence topics 105
WORD operator 63

Y

YESNO operator 85

Z

ZONE modifier 78
zone search
description 139
document 51
internet-style query 139