MarkLogic Server

Search Developer's Guide

MarkLogic 6
September, 2012

Last Revised: 6.0-5, March, 2014

Copyright © 2014 MarkLogic Corporation. All rights reserved.

Table of Contents

Search Developer's Guide

1.0 Developing Search Applications in MarkLogic Server 13
1.1 Overview of Search Features in MarkLogic Server 13
1.1.1 Search API Library Module 13
1.1.2 Built-In Search Functions in the cts and xdmp Namespaces 13
1.1.3 Full XPath Search Support 14
1.1.4 Lexicon and Range Index-Based APIs 14
1.1.5 Stemming, Wildcard, Spelling, and Much More Functionality 14
1.1.6 Alerting API and Built-Ins 14
1.2 Where to Find Search Information 15

2.0 Search API: Understanding and Using 16
2.1 Understanding the Search API 16
2.1.1 XQuery Library Module 17
2.1.2 Simple search:search Example and Response Output 17
2.1.3 Automatic Query Text Parsing and Grammar 18
2.1.4 Constrained Searches and Faceted Navigation 20
2.1.5 Built-In Snippetting 21
2.1.6 Search Term Completion 22
2.1.7 Search Customization Via Options and Extensions 22
2.1.8 Speed and Accuracy 23
2.2 Controlling the Search With the Options Node 24
2.2.1 Checking an Options Node With search:check-options 24
2.2.2 Constraint Options 25
2.2.3 Operator Options 35
2.2.4 Return Options 36
2.2.5 Searchable Expression Option 37
2.2.6 Fragment Scope Option 37
2.2.7 Modifying Your Snippet Results 38
2.2.8 Other Search Options 41
2.3 Search Term Completion Using search:suggest 41
2.3.1 default-suggestion-source Option 42
2.3.2 Choose Suggestions With the suggestion-source Option 42
2.3.3 Use Multiple Query Text Inputs to search:suggest 43
2.3.4 Make Suggestions Based on Cursor Position 44
2.3.5 search:suggest Examples 44
2.4 Creating a Custom Constraint 45
2.4.1 Implementing the parse Function 45
2.4.2 Implementing the start-facet Function 47

2.4.3 Implementing the finish-facet Function 48
2.4.4 Example: Creating a Simple Custom Constraint 50
2.4.5 Example: Creating a Custom Constraint for Structured Queries 51
2.4.6 Example: Creating a Custom Constraint Geospatial Facet 53
2.5 Search Grammar 56
2.5.1 Basic Search Grammar Syntax 57
2.5.2 Modifying and Extending the Search Parsing Grammar 58
2.6 Using Structured Search as an Alternate to cts:query 61
2.7 Returning Lexicon Values With search:values 62
2.8 JSON Support in the Search API 62
2.9 More Search API Examples 64
2.9.1 Buckets Example 64
2.9.2 Computed Buckets Example 66
2.9.3 Sort Order Example 68

3.0 Searching Using Structured Queries 69
3.1 Structured Query Overview 69
3.2 Constructing a Structured Query 70
3.3 Syntax Summary 71
3.4 Examples of Structured Queries 72
3.4.1 Example: Simple Structured Search 72
3.4.2 Example: Structured Search With Constraint References as Text 73
3.4.3 Example: Structured Search With Constraint References 73
3.5 Syntax Reference 75
3.5.1 query 76
3.5.2 term-query 76
3.5.3 and-query 78
3.5.4 or-query 79
3.5.5 and-not-query 81
3.5.6 not-query 83
3.5.7 not-in-query 84
3.5.8 near-query 86
3.5.9 properties-query 88
3.5.10 directory-query 91
3.5.11 collection-query 92
3.5.12 document-query 93
3.5.13 document-fragment-query 95
3.5.14 locks-query 96
3.5.15 range-constraint-query 97
3.5.16 value-constraint-query 100
3.5.17 word-constraint-query 103
3.5.18 collection-constraint-query 105
3.5.19 element-constraint-query 107
3.5.20 properties-constraint-query 110
3.5.21 custom-constraint-query 111
3.5.22 geospatial-constraint-query 114

3.5.23 operator-state 117

4.0 Composing cts:query Expressions 121
4.1 Understanding cts:query 121
4.1.1 cts:query Hierarchy 122
4.1.2 Use to Narrow the Search 122
4.1.3 Understanding cts:element-query 123
4.1.4 Understanding cts:element-word-query 123
4.1.5 Understanding Field Word and Value Query Constructors 124
4.1.6 Understanding the Range Query Constructors 124
4.1.7 Understanding the Reverse Query Constructor 124
4.1.8 Understanding the Geospatial Query Constructors 124
4.1.9 Specifying the Language in a cts:query 125
4.2 Combining multiple cts:query Expressions 125
4.2.1 Using cts:and-query and cts:or-query 126
4.2.2 Proximity Queries using cts:near-query 126
4.2.3 Using Bounded cts:query Expressions 126
4.2.4 Matching Nothing and Matching Everything 127
4.3 Joining Documents and Properties with cts:properties-query or cts:document-fragment-query 127
4.4 Registering cts:query Expressions to Speed Search Performance 128
4.4.1 Registered Query APIs 129
4.4.2 Must Be Used Unfiltered 129
4.4.3 Registration Does Not Survive System Restart 129
4.4.4 Storing Registered Query IDs 130
4.4.5 Registered Queries and Relevance Calculations 130
4.4.6 Example: Registering and Using a cts:query Expression 130
4.5 Adding Relevance Information to cts:query Expressions: 131
4.6 XML Serializations of cts:query Constructors 131
4.6.1 Serializing a cts:query to XML 131
4.6.2 Add Arbitrary Annotations With cts:annotate 132
4.6.3 Function to Construct a cts:query From XML 132
4.7 Example: Creating a cts:query Parser 132

5.0 Relevance Scores: Understanding and Customizing 136
5.1 Understanding How Scores and Relevance are Calculated 136
5.1.1 log(tf)*idf Calculation 137
5.1.2 log(tf) Calculation 137
5.1.3 Simple Term Match Calculation 138
5.1.4 Random Score Calculation 138
5.1.5 Term Frequency Normalization 138
5.2 How Fragmentation and Index Options Influence Scores 139
5.3 Using Weights to Influence Scores 139
5.4 Proximity Boosting With the distance-weight Option 140
5.4.1 Example of Simple Proximity Boosting 140

5.4.2 Using Proximity Boosting With cts:and-query Semantics 141
5.4.3 Using cts:near-query to Achieve Proximity Boosting 142
5.5 Interaction of Score and Quality 143
5.6 Using cts:score, cts:confidence, and cts:fitness 143
5.7 Relevance Order in cts:search Versus Document Order in XPath 144
5.8 Sample cts:search Expressions 145
5.8.1 Magnify the Score Boost for Documents With Quality 145
5.8.2 Increase the Score for some Terms, Decrease for Others 145

6.0 Browsing With Lexicons 146
6.1 About Lexicons 146
6.2 Creating Lexicons 147
6.3 Word Lexicons 148
6.3.1 Word Lexicon for the Entire Database 148
6.3.2 Element/Element-Attribute Word Lexicons 149
6.3.3 Field Word Lexicons 149
6.4 Element/Element-Attribute/Path Value Lexicons 149
6.5 Field Value Lexicons 150
6.6 Value Co-Occurrences Lexicons 151
6.7 Geospatial Lexicons 153
6.8 Range Lexicons 154
6.9 URI and Collection Lexicons 154
6.10 Performing Lexicon-Based Queries 155
6.10.1 Lexicon APIs 155
6.10.2 Constraining Lexicon Searches to a cts:query Expression 156
6.10.3 Using the Match Lexicon APIs 157
6.10.4 Determining the Number of Fragments Containing a Lexicon Term 157

7.0 Using Range Queries in cts:query Expressions 159
7.1 Overview of Range Queries 159
7.1.1 Uses for Range Queries 159
7.1.2 Requirements for Using Range Queries 160
7.1.3 Performance and Coding Advantages of Range Queries 160
7.2 Range Query cts:query Constructors 161
7.3 Examples of Range Queries 161

8.0 Using Aggregate Functions 163
8.1 Introduction to Aggregate Functions 163
8.2 Using Builtin Aggregate Functions 163
8.3 Using Aggregate User-Defined Functions 165

9.0 Highlighting Search Term Matches 167
9.1 Overview of cts:highlight 167
9.1.1 All Matching Terms, Including Stemmed, and Capitalized 167

9.2 General Search and Replace Function 168
9.3 Built-In Variables For cts:highlight 168
9.3.1 Using the $cts:text Variable to Access the Matched Text 169
9.3.2 Using the $cts:node Variable to Access the Context of the Match 169
9.3.3 Using the $cts:queries Variable to Feed Logic Based on the Query 170
9.3.4 Using $cts:start to Capture the String-Length Position 171
9.3.5 Using $cts:action to Stop Highlighting 171
9.4 Using cts:highlight to Create Snippets 171
9.5 cts:walk Versus cts:highlight 172
9.6 Common Usage Notes 172
9.6.1 Input Must Be a Single Node 173
9.6.2 Using xdmp:set Side Effects With cts:highlight 173
9.6.3 No Highlighting with cts:similar-query or cts:element-attribute-*-query 174

10.0 Geospatial Search Applications 175
10.1 Overview of Geospatial Data in MarkLogic Server 175
10.1.1 Terminology 175
10.1.2 Coordinate System 176
10.1.3 Types of Geospatial Queries 176
10.1.4 XQuery Primitive Types And Constructors for Geospatial Queries 177
10.1.5 Well-Known Text (WKT) Markup Language 177
10.2 Understanding Geospatial Coordinates and Regions 179
10.2.1 Understanding the Basics of Coordinates and Points 179
10.2.2 Understanding Geospatial Boxes 180
10.2.3 Understanding Geospatial Polygons: Polygons, Complex Polygons, and Linestrings 181
10.2.4 Understanding Geospatial Circles 183
10.3 Geospatial Indexes 183
10.3.1 Different Kinds of Geospatial Indexes 184
10.3.2 Geospatial Index Positions 186
10.3.3 Geospatial Lexicons 186
10.4 Using the API 186
10.4.1 Basic Procedure for Performing a Geospatial Query 186
10.4.2 Geospatial cts:query Constructors 187
10.4.3 Geospatial Value Constructors for Regions 187
10.4.4 Geospatial Format Conversion Functions 187
10.4.5 Geospatial Operations 188
10.5 Simple Geospatial Search Example 188

11.0 Marking Up Documents With Entity Enrichment 191
11.1 Overview of Entity Enrichment 191
11.2 Built-In Entity Enrichment 191
11.2.1 Installing the Entity Enrichment Libraries 191
11.2.2 Details of MarkLogic Entity Enrichment 193

11.3 Entity Enrichment Pipelines 194
11.3.1 MarkLogic Server Entity Enrichment Pipeline 194
11.3.2 Sample Pipelines Using Third-Party Technologies 195
11.3.3 Custom Entity Enrichment Pipelines 195

12.0 Creating Alerting Applications 196
12.1 Overview of Alerting Applications in MarkLogic Server 196
12.2 cts:reverse-query Constructor 197
12.3 XML Serialization of cts:query Constructors 197
12.4 Security Considerations of Alerting Applications 197
12.4.1 Alert Users, Alert Administrators, and Controlling Access 198
12.4.2 Predefined Roles for Alerting Applications 198
12.5 Indexes for Reverse Queries 199
12.6 Alerting API 199
12.6.1 Alerting API Concepts 200
12.6.2 Using the Alerting API 201
12.6.3 Using CPF With an Alerting Application 204
12.7 Alerting Sample Application 206

13.0 Using fn:count vs. xdmp:estimate 207
13.1 fn:count is Accurate, xdmp:estimate is Fast 207
13.2 The xdmp:estimate Built-In Function 207
13.3 Using cts:remainder to Estimate the Size of a Search 208
13.4 When to Use xdmp:estimate 208
13.4.1 When Estimates Are Good Enough 210
13.4.2 When XPaths Meet The Right Criteria 210
13.4.3 When Empirical Tests Demonstrate Correctness 211

14.0 Understanding and Using Stemmed Searches 212
14.1 Stemming in MarkLogic Server 212
14.2 Enabling Stemming 213
14.3 Stemmed Searches Versus Word Searches 214
14.4 Using cts:highlight or cts:contains to Find if a Word Matches a Query 215
14.5 Interaction With Wildcard Searches 215

15.0 Custom Dictionaries for Tokenizing and Stemming 216
15.1 Custom Dictionaries in MarkLogic Server 216
15.2 Dictionary and Entry Schemas 217
15.3 Custom Dictionary Functions 218
15.3.1 Get All Licensed Languages 218
15.3.2 Get A Custom Dictionary 219
15.3.3 Add/Write A Custom Dictionary 219
15.3.4 Delete A Custom Dictionary 219
15.4 Usage Examples 219

16.0 Extracting Metadata and Text From Binary Documents 222
16.1 Metadata and Text Extraction Overview 222
16.2 Usage Examples 222
16.2.1 Microsoft Word 223
16.2.2 File Archives 224
16.2.3 PowerPoint 226
16.3 Supported Binary Formats 227
16.3.1 **Archives 228**
16.3.2 **Databases 228**
16.3.3 Email and Messaging 228
16.3.4 Multimedia 228
16.3.5 Other 228
16.3.6 .....Presentation 229
16.3.7 Raster Image 229
16.3.8 Spreadsheet 229
16.3.9 Text and Markup 229
16.3.10Vector Image 229
16.3.11Word Processing and General Office 230

17.0 Understanding and Using Wildcard Searches 231
17.1 Wildcards in MarkLogic Server 231
17.1.1 Wildcard Characters 231
17.1.2 Rules for Wildcard Searches 231
17.2 Enabling Wildcard Searches 232
17.2.1 Specifying Wildcards in Queries 233
17.2.2 Recommended Wildcard Index Settings 233
17.2.3 Understanding the Different Wildcard Indexes 234
17.3 Interaction with Other Search Features 235
17.3.1 Wildcarding and Stemming 236
17.3.2 Wildcarding and Punctuation Sensitivity 236

18.0 Collections 241
18.1 The collection() Function 241
18.2 Collections Versus Directories 242
18.3 Defining Collections 243
18.3.1 Implicitly Defining Unprotected Collections 243
18.3.2 Explicitly Defining Protected Collections 244
18.4 Collection Membership 245
18.5 Collections and Security 245
18.5.1 Unprotected Collections 246
18.5.2 Protected Collections 247
18.6 Performance Characteristics 247
18.6.1 Number of Collections to Which a Document Belongs 248
18.6.2 Adding/Removing Existing Documents To/From Collections 248

19.0 Using the Thesaurus Functions 249
19.1 The Thesaurus Module 249
19.2 Function Reference 249
19.3 Thesaurus Schema 250
19.4 Capitalization 250
19.5 Managing Thesaurus Documents 250
19.5.1 Loading Thesaurus Documents 251
19.5.2 Lowercasing Terms When Inserting a Thesaurus Document 251
19.5.3 Loading the XML Version of the WordNet Thesaurus 252
19.5.4 Updating a Thesaurus Document 252
19.5.5 Security Considerations With Thesaurus Documents 253
19.5.6 Example Queries Using Thesaurus Management Functions 253
19.6 Expanding Searches Using a Thesaurus 256

20.0 Using the Spelling Correction Functions 257
20.1 Overview of Spelling Correction 257
20.2 Function Reference 257
20.2.1 The Spelling Built-In Functions 258
20.2.2 The Spelling Dictionary Management Module Functions 258
20.3 Dictionary Documents 259
20.4 Capitalization 259
20.5 Managing Dictionary Documents 260
20.5.1 Loading Dictionary Documents 260
20.5.2 Loading one of the Sample XML Dictionaries 260
20.5.3 Updating a Dictionary Document 261
20.5.4 Security Considerations With Dictionary Documents 262
20.6 Testing if a Word is Spelled Correctly 262
20.7 Getting Spelling Suggestions for Incorrectly Spelled Words 263

21.0 Distinctive Terms and cts:similar-query 264
21.1 Understanding cts:similar-query 264
21.2 Finding the Distinctive Terms of a Set of Nodes 264
21.3 Understanding the cts:distinctive-terms Output 265
21.4 Example Design Pattern: Making a Tag Cloud 266

22.0 Training the Classifier 268
22.1 Understanding How Training and Classification Works 268
22.1.1 Training and Classification 268
22.1.2 XML SVM Classifier 268
22.1.3 Hyper-Planes and Thresholds for Classes 269
22.1.4 Training Content for the Classifier 273
22.2 Classifier API 273
22.2.1 XQuery Built-In Functions 273
22.2.2 Data Can Reside Anywhere or Be Constructed 274

22.2.3 API is Extremely Tunable 274
22.2.4 Supports Versus Weights Classifiers 274
22.2.5 Kernels (Mapping Functions) 275
22.2.6 Find Thresholds That Balance Precision and Recall 275
22.3 Leveraging XML With the Classifier 275
22.4 Creating a Training Set 275
22.4.1 Importance of the Training Set 276
22.4.2 Defining Labels for the Training Set 276
22.5 Methodology For Determining Thresholds For Each Class 277
22.6 Example: Training and Running the Classifier 278
22.6.1 Shakespeare's Plays: The Training Set 279
22.6.2 Comedy, Tragedy, History: The Classes 279
22.6.3 Partition the Training Content Set 279
22.6.4 Create Labels on the First Half of the Training Content 280
22.6.5 Run cts:train on the First Half of the Training Content 280
22.6.6 Run cts:classify on the Second Half of the Content Set 281
22.6.7 Use cts:thresholds to Compute the Thresholds on the Second Half 282
22.6.8 Evaluating Your Results, Make Changes, and Run Another Iteration 282
22.6.9 Run the Classifier on Other Content 283

23.0 Results Clustering Using cts:cluster 284
23.1 Understanding cts:cluster 284
23.2 Options to cts:cluster 285
23.2.1 Clustering (cts:cluster) Options 285
23.2.2 Indexing (db:) Options 286
23.3 Understanding the cts:cluster Output 286
23.4 Example that Creates an HTML Report of the Cluster 288

24.0 Language Support in MarkLogic Server 292
24.1 Overview of Language Support in MarkLogic Server 292
24.2 Tokenization and Stemming 293
24.2.1 Language-Specific Tokenization 293
24.2.2 Stemmed Searches in Different Languages 295
24.3 Language Aspects of Loading and Updating Documents 296
24.3.1 Tokenization and Stemming 296
24.3.2 xml:lang Attribute 296
24.3.3 Language-Related Notes About Loading and Updating Documents 297
24.4 Querying Documents By Languages 298
24.4.1 Tokenization, Stemming, and the xml:lang Attribute 298
24.4.2 Language-Aware Searches 298
24.4.3 Unstemmed Searches 299
24.4.4 Unknown Languages 300
24.5 Supported Languages 301
24.6 Generic Language Support 302

25.0 Encodings and Collations 303
25.1 Character Encoding 303
25.2 Collations 304
25.2.1 Overview of Collations 304
25.2.2 Two Common Collation URIs 305
25.2.3 Collation URI Syntax 305
25.2.4 Backward Compatibility with 3.1 Range Indexes and Lexicons 309
25.2.5 UCA Root Collation 309
25.2.6 How Collation Defaults are Determined 309
25.2.7 Specifying Collations 311
25.3 Collations and Character Sets By Language 311

26.0 Data Visualization Widgets 315
26.1 Overview of Visualization Widgets 316
26.2 Working with the Visualization Widgets 319
26.2.1 Building The Oscars Example Application 319
26.2.2 General Operation of Widgets 321
26.2.3 Working with the Line Chart Widget 322
26.2.4 Working with the Bar and Column Chart Widgets 323
26.2.5 Working with the Pie Chart Widget 325
26.2.6 Working with the Map Widget 326
26.3 Overview of the Widget Architecture 329
26.3.1 Widgets 330
26.3.2 Controller 330
26.3.3 Identification 330
26.3.4 Interaction within a Widget 331
26.3.5 Shadow Queries 331
26.4 Adding Visualization Widgets to an HTML Page 332
26.4.1 Common JavaScript And CSS In The <head> Element 332
26.4.2 Common Code In The <body> Element 333
26.4.3 Initialize the Display 334
26.5 Example: Adding Widgets to Applications 335
26.6 Visualization Widget Limitations 342
26.7 Set Up A Proxy 342
26.8 Widget API Reference 343
26.8.1 ML.controller Methods 343
26.8.2 ML.createWidget (Null Widget) Method 345
26.8.3 General Widget Methods 346
26.8.4 ML.chartWidget Method 347
26.8.5 ML.mapWidget Method 348
26.8.6 Widget Events 350
26.8.7 Widget Query Structure 351
26.8.8 Widget Query Update Structure 351
26.8.9 Search Results Format 351
26.8.10Facet Structure 352

26.8.11Internal Widget Events 353

27.0 Technical Support 354

Combined Product Notices 355
COPYRIGHT 355
TRADEMARK NOTICE 367

1.0 Developing Search Applications in MarkLogic Server

This chapter provides an overview of developing search applications in MarkLogic Server, and includes the following sections:

- Overview of Search Features in MarkLogic Server
- Where to Find Search Information

1.1 Overview of Search Features in MarkLogic Server

MarkLogic Server includes rich full-text search features. All of the search features are implemented as extension functions available in XQuery. This section provides a brief overview some of the main search features in MarkLogic Server and includes the following parts:

- Search API Library Module
- Built-In Search Functions in the cts and xdmp Namespaces
- Full XPath Search Support
- Lexicon and Range Index-Based APIs
- Stemming, Wildcard, Spelling, and Much More Functionality
- Alerting API and Built-Ins

1.1.1 Search API Library Module

The Search API is an XQuery library module designed to simplify creating search applications. The Search API makes it easy to create complex search applications that include faceted navigation, complex and extensible search grammar, and many other search features. The Search API uses the core text search capabilities of MarkLogic Server, but abstracts those APIs from the developer. It is designed to be easy to use in the normal cases, and to be flexible enough to allow complex customization. For details on the Search API, see “Search API Library Module” on page 13.

1.1.2 Built-In Search Functions in the cts and xdmp Namespaces

The core search functionality in MarkLogic Server are a set of built-in XQuery functions to perform full-text search. The `cts:search` function returns nodes matching a `cts:query`, which is a composable search query which allows you to perform fine-grained searches. MarkLogic Server is designed to scale to extremely large databases (100s of terabytes or more), and the search operates directly against the database, no matter what the database size. As part of loading a document, full-text indexes are created making arbitrary searches fast. Searches automatically use the indexes, and the `xdmp:estimate` function and the `unfiltered` option to `cts:search` allow you to return results directly out of the indexes in MarkLogic Server.

The built-in search capabilities are designed to be extensible and to work in a large number of applications. For example, the Search API is built using the core text search features such as `cts:search`, `cts:word-query`, `cts:element-value-query`, and so on.

1.1.3 Full XPath Search Support

MarkLogic Server implements the XQuery language, which includes XPath 2.0. XPath expressions are searches which can search across the entire database. For example, consider the following XPath expression:

```
/my-node/my-child[fn:contains(., "hello")]
```

This expression searches across the entire database returning `my-child` nodes that match the expression. XPath expressions take full advantage of the indexes in the database and are designed to be fast.

1.1.4 Lexicon and Range Index-Based APIs

MarkLogic Server has range indexes which index named XML structures such as elements and attributes. There are also range indexes over geospatial values. Each of these range indexes has lexicon APIs associated with them. The lexicon APIs allow you to return values directly from the indexes. Lexicons are very useful in constructing facets and in finding fast counts of element or attribute values. The Search API makes extensive use of the lexicon APIs. For details about lexicons, see “Browsing With Lexicons” on page 146.

1.1.5 Stemming, Wildcard, Spelling, and Much More Functionality

MarkLogic Server search supports a wide range of full-text features. These features include stemming, wildcarded searches, diacritic-sensitive/insensitive searches, case-sensitive/insensitive searches, spelling correction functions, thesaurus functions, geospatial searches, advanced language and collation support, and much more. These features are all designed to build off of each other and work together in an extensible and flexible way.

1.1.6 Alerting API and Built-Ins

You can create applications that notify users when new content is available that matches a predefined query. There is an API to help build these applications as well as a built-in `cts:query` constructor (`cts:reverse-query`) and indexing support to build large and scalable alerting applications. For details on alerting applications, see “Creating Alerting Applications” on page 196.

1.2 Where to Find Search Information

The *MarkLogic XQuery and XSLT Function Reference* contains the XQuery function signatures and descriptions, as well as many code examples. This *Search Developer's Guide* contains descriptions and technical details about the search features in MarkLogic Server, including:

- "Search API: Understanding and Using" on page 16
- "Composing cts:query Expressions" on page 121
- "Relevance Scores: Understanding and Customizing" on page 136
- "Browsing With Lexicons" on page 146
- "Using Range Queries in cts:query Expressions" on page 159
- "Highlighting Search Term Matches" on page 167
- "Geospatial Search Applications" on page 175
- "Marking Up Documents With Entity Enrichment" on page 191
- "Creating Alerting Applications" on page 196
- "Using fn:count vs. xdmp:estimate" on page 207
- "Understanding and Using Stemmed Searches" on page 212
- "Understanding and Using Wildcard Searches" on page 231
- "Collections" on page 241
- "Using the Thesaurus Functions" on page 249
- "Using the Spelling Correction Functions" on page 257
- "Language Support in MarkLogic Server" on page 292
- "Encodings and Collations" on page 303

For other information about developing applications in MarkLogic Server, see the *Application Developer's Guide*. For information about XQuery in MarkLogic Server, see the *XQuery and XSLT Reference Guide*.

2.0 Search API: Understanding and Using

This chapter describes the Search API, which is an XQuery API designed to make it easy to create search applications that contain facets, search results, and snippets. This chapter includes the following sections:

- Understanding the Search API
- Controlling the Search With the Options Node
- Search Term Completion Using search:suggest
- Creating a Custom Constraint
- Search Grammar
- Using Structured Search as an Alternate to cts:query
- Returning Lexicon Values With search:values
- JSON Support in the Search API
- More Search API Examples

This chapter provides background, design patterns, and examples of using the Search API. For the function signatures and descriptions, see the Search documentation under XQuery Library Modules in the *MarkLogic XQuery and XSLT Function Reference*.

2.1 Understanding the Search API

The Search API is an XQuery library that combines searching, search parsing, search grammar, faceting, snippeting, search term completion, and other search application features into a single API. You can interact with the Search API through XQuery, REST, and Java, using string queries, structured queries, and `cts:query`.

The Search API makes it easy to create search applications without needing to understand many of the details of the underlying `cts:search` and `cts:query` APIs. The Search API is designed for large-scale, production applications.

This section provides an overview and describes some of the features of the Search API, and contains the following topics:

- XQuery Library Module
- Simple search:search Example and Response Output
- Automatic Query Text Parsing and Grammar
- Constrained Searches and Faceted Navigation
- Built-In Snippetting

- Search Term Completion
- Search Customization Via Options and Extensions
- Speed and Accuracy

2.1.1 XQuery Library Module

The Search API is implemented as an XQuery library module. You can use it directly from XQuery. You can also access most of the Search API features through the REST and Java API's; for details, see *REST Application Developer's Guide* or *Java Application Developer's Guide*.

To use the Search API from XQuery, import the Search API library module into your XQuery module with the following prolog statement:

```
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";
```

The Search API uses the prefix `search:`, which is not predefined in the server. The Search API has the following core functions to perform searches and provide search results, snippets, and query-completion suggestions: `search:search`, `search:snippet`, and `search:suggest`. There are also other functions to perform these activities at finer granularities and to provide convenience tools.

For the Search API function signatures and details about each individual function, see the *MarkLogic XQuery and XSLT Function Reference* for the Search API.

2.1.2 Simple search:search Example and Response Output

The `search:search` function takes search terms, parses them into an appropriate `cts:query`, and returns a response with snippets and URIs for matching nodes in the database. You can get started with the Search API with a very simple query:

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

search:search("hello world")
=>
<search:response total="1" start="1" page-length="10" xmlns=""
  xmlns:search="http://marklogic.com/appservices/search">
  <search:result index="1" uri="/hello.xml"
    path="doc(&quot;/hello.xml&quot;)" score="136"
    confidence="0.67393" fitness="0.67393">
    <search:snippet>
      <search:match path="doc(&quot;/hello.xml&quot;)/hello">This is
        where you say "<search:highlight>Hello</search:highlight>
```

```
          <search:highlight>World</search:highlight>".
        </search:match>
      </search:snippet>
    </search:result>
    <search:qtext>hello world</search:qtext>
    <search:metrics>
      <search:query-resolution-time>PT0.328S
        </search:query-resolution-time>
      <search:total-time>PT0.352S</search:total-time>
    </search:metrics>
  </search:response>
```

The output is a `search:response` element, and it contains everything needed to build a search results page. It includes an estimate of the total number of documents that match the search, the URI and XPath for each result, pagination of the search results, a snippet of the result content, the original query text submitted, and metrics on the response time.

To try the Search API on your own content, run a simple search like the above example against a database of your own content, and then examine the search results.

The `search:search` function is highly customizable, but by default it includes sensible settings that will provide good results for many applications. With the results of `search:search`, it is easy to build useful results pages that are as simple or as complex as you like.

2.1.3 Automatic Query Text Parsing and Grammar

In a typical search application, a user enters query text into a search box in a browser. This text is a *string query*. The Search API automatically parses a string query into a `cts:query` for efficient and powerful searches. You can use string queries in XQuery, Java, and REST, through interfaces such as the following:

- XQuery: The `search:search` and `search:parse functions`
- Java: The `com.marklogic.client.query.QueryManager` class
- REST: The `/search` service

By default, the query text is parsed using a grammar similar to the Google grammar. For example, double-quoted phrases in query text such as the following are treated as phrases in a search:

```
"this is a phrase"
```

The default grammar also supports `AND`, `OR`, grouping with parenthesis (`( )`), negation with a minus sign (`-`), and user-configured constraints with a colon (`:`). The following is a summary of the default grammar:

- Terms may be free standing:

  ```
  cat
  ```

- `AND` and `OR` operators, with `AND` having higher precedence.
- Parentheses operators can override default precedence:

```
(cat OR dog) AND horse
```

- Multiple terms are combined as an `AND`:

```
cat dog
```

- Phrases are surrounded by double-quotes:

```
"cat and dog"
```

- Terms are excluded through a leading minus:

```
cat –dog
```

- Colon operators indicate configured constraint or operator searches (for details, see "Constraint Options" on page 25 and "Operator Options" on page 35):

```
tag:value
```

- Constraint and operator searches may operate over phrases:

```
tag:"a phrase value"
```

- A query text can comprise any number of these types of searches in any order.
- The default precedence for a search order provides preference to explicitly ordered (with parenthesis, for example) then for implicitly ordered. Therefore, multi-term queries using the explicit `AND` operator do not parse as equivalent to the same string using the implicit `AND` because there is a difference in the way that precedence is applied. For example, `A OR B AND C` parses to the equivalent of `A OR (B AND C)`, while `A OR B C` parses to the equivalent of `(A OR B) and C`.

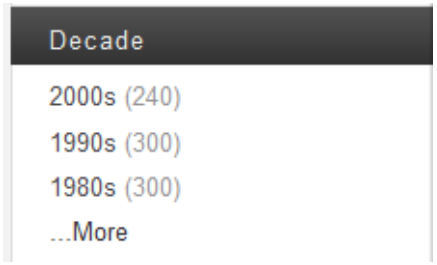
The query text parsing happens automatically, with no additional coding. The parsing takes into account constraints and operators specified in an options node at search runtime. Additionally, you can change, extend, and modify the default search parsing grammar in the options node. Most applications will not need to modify the search grammar, as the default grammar is quite robust and full-featured. For details on modifying the default grammar, see "Search Grammar" on page 56. For details on the options node for the Search API, see "Controlling the Search With the Options Node" on page 24.

2.1.4 Constrained Searches and Faceted Navigation

The Search API makes it easy to constrain your searches to a subset of the content. For example, you can create a search that only returns results for documents with titles that include the word `hello`, or you can create a search that constrains the results to a particular decade. Furthermore, the Search API makes it easy to express these kinds of searches in a simple query text string. For example, you can write a query such that the following query text represents a search that constrains to a particular decade:

```
decade:2000s
```

These types of searches are useful in creating facets, which allow a user to drill down by narrowing the search criteria. Facets also typically have counts of the number of results that match, and the Search API returns these counts to use in facets. The following is an example of a facet in an end-user application:



Users can click on any of the links to narrow the results of the search by decade. For example, the query generated by clicking the top link contains the string `decade:2000s`, and constrains the search to that decade.

The facet also includes counts for each constraint value. The number to the right of the link represents the number of search results returned if you constrain it to that decade.

The Search API returns XML in its response that contains all of the information to create a facet like the above example. The REST and Java APIs can return this information as XML or JSON. The facets returned from the Search API include the counts and values needed to generate the user interface. For example, the following XML, returned from the Search API, was used to create the above facet:

```
<search:response total="2370" start="1" page-length="10" xmlns=""
   xmlns:search="http://marklogic.com/appservices/search">
  <search:facet name="decade">
    <search:facet-value name="2000s" count="240">
     2000s</search:facet-value>
    <search:facet-value name="1990s" count="300">
     1990s</search:facet-value>
    <search:facet-value name="1980s" count="300">
     1980s</search:facet-value>
    <search:facet-value name="1970s" count="300">
     1970s</search:facet-value>
    <search:facet-value name="1960s" count="299">
     1960s</search:facet-value>
    <search:facet-value name="1950s" count="300">
     1950s</search:facet-value>
    <search:facet-value name="1940s" count="324">
     1940s</search:facet-value>
    <search:facet-value name="1930s" count="245">
     1930s</search:facet-value>
    <search:facet-value name="1920s" count="61">
     1920s</search:facet-value>
  </search:facet>
</search:response>
```

The counts and values in the response are also filtered by any other active query in the search, so they represent the counts for that particular search. There are many kinds of constraints and facets you can build with the Search, REST, and Java APIs. For more details about constraints, see “Constraint Options” on page 25.

2.1.5 Built-In Snippetting

A search results page typically shows portions of matching documents with the search matches highlighted, perhaps with some text showing the context of the search matches. These search result pieces are known as *snippets*. For example, a search for `MarkLogic Server` might produce the following snippet:

```
MarkLogic Server is an XML Server that provides the agility you need
to build and ... Use MarkLogic Server's geospatial capability to
create new dynamic ...
```

The Search API includes snippets in the search:response output, and makes it easy to create search results pages that show the matches in the context of the document. Providing the best snippet for a given content set is often very application specific, however. Therefore, the Search API allows you to customize the snippets, either using the built-in snippetting algorithm or by adding your own snippetting code. For details on ways to customize the snippetting behavior for your searches, see "Modifying Your Snippet Results" on page 38.

2.1.6 Search Term Completion

Search applications often offer suggestions for search terms as the user types into the search box. The suggestions are based on terms that are in the database, and are typically used to make the user interface more interactive and to quickly suggest search terms that are appropriate to the application. The `search:suggest` function in the Search API is designed to supply the terms to a search-completion user interface. For more details on how to use search term completion, see "Search Term Completion Using search:suggest" on page 41.

2.1.7 Search Customization Via Options and Extensions

The Search API is designed to make it easy to customize your searches. A wide range of customizations are available directly through the options that you pass into the search. There are a large number of options controlling nearly every aspect of the search you are performing.

For cases where the built-in options do not do what you need, there is an extension mechanism built into the Search API. The mechanism includes hooks in the Search API which allow you to call out to your own XQuery code. The hooks allow you to specify the location and name of the function containing your own implementation of a function to replace the implementation of that function in the Search API. The Search API uses function values to pass your custom function as a parameter, replacing the default Search API functionality. For details on function values, see Function Values in the *Application Developer's Guide*.

The basic pattern to specify your extension function using the attributes `apply`, `ns`, and `at` as attributes on various elements in the `search:options` node. These correspond to the localname of your implemented function, the namespace of the function, and the location of the function library module in which the code exists, respectively. For example, consider the following:

```
<transform-results apply="my-snippet" ns="my-namespace"
     at="/my-module.xqy" />
```

In this example, the `transform-results` option specifies to use the `my-snippet` function in the library module `my-module` under your App Server root instead of the default snippeting function that the Search API uses. For additional details about working with `transform-results`, see "Modifying Your Snippet Results" on page 38.

Any search option that has an `apply` attribute can use this extension pattern to point to your own implementation for the functionality of that option, including `transform-results`, several `grammar` options, `custom` constraints, and so on.

2.1.8 Speed and Accuracy

The Search API is designed to be fast. When creating any search application, you make trade-offs between speed and guaranteed accuracy. The values of various options in the Search API control things like filtered versus unfiltered search, diacritic and case-sensitivity, and other options. These options affect the accuracy of search estimates in MarkLogic Server. The default values of these type of options in the Search API are designed to be sensible for most application. All applications are different, however, and the Search API gives you the tools to control what makes sense for your specific application.

Range constraints use lexicons to get fast accurate unique values and counts. Keep in mind, however, that certain operations might not produce accurate counts in all cases. For example, when you pass a `cts:query` into a lexicon API (which the Search API does in some cases), it filters the lexicon calls based on the index resolution of the `cts:query`, not on the filtered search values, and the index resolution is not guaranteed to be accurate for all queries. For details on how search index resolution works, see Fast Pagination and Unfiltered Searches in *Query Performance and Tuning Guide*.

Other factors such as fragmentation and what you search for (`searchable-expression` in the Search API options) can also contribute to whether the index resolution for a search is correct, as can various options to lexicons. The Search API default values for these various options make the trade-offs that are sensible for many search applications. For example, the value of the `total` attribute in the `search:response` output is the result of an `cts:remainder`, which will always be fast but is not guaranteed to be accurate for all searches. For details, see “Using fn:count vs. xdmp:estimate” on page 207.

2.2 Controlling the Search With the Options Node

The `search:search` function and most of the other functions in the Search API take an optional options node as a parameter. The options node allows you to specify the behavior of the Search API. If you do not specify an options node, the API uses a set of defaults that are designed to be sensible for many applications. You can use the `search:get-default-options` function to see the default options. The options node allows you to specify constraints, custom grammar, search options, what parts of the response to return, what expression to search over, and so on.

The options node is in the following namespace:

```
http://marklogic.com/appservices/search
```

This section describes the following portions of the options node:

- Checking an Options Node With search:check-options
- Constraint Options
- Operator Options
- Return Options
- Searchable Expression Option
- Fragment Scope Option
- Modifying Your Snippet Results
- Other Search Options

For details the syntax of each option, see the `search:search` function documentation in *MarkLogic XQuery and XSLT Function Reference*.

2.2.1 Checking an Options Node With search:check-options

The options XML node can be fairly complex, and there is a `search:check-options` function that reports errors in your options node. The `search:check-options` function validates your options node and reports any errors it finds. It returns empty if the options node is valid. If it finds errors, they are returned in the form of one or more `search:report` nodes.

It is a good idea to only use the `search:check-options` function in development, as it will slow down queries to check the options on every search. You can also use the `<debug>true</debug>` option in the `search:options` node, which will return the output of `search:check-options` as part of your response.

One common design pattern is to add a `$debug` option to your code that defaults to `false`, and when `true`, have your code run `search:check-options` on the options node or add the `debug` option to the options node. If you have a variable called `$debug` in your code that is normally set to `false`, then setting it to `true` results in checking your options node. Then in production, you can set it back to `false`.

2.2.2 Constraint Options

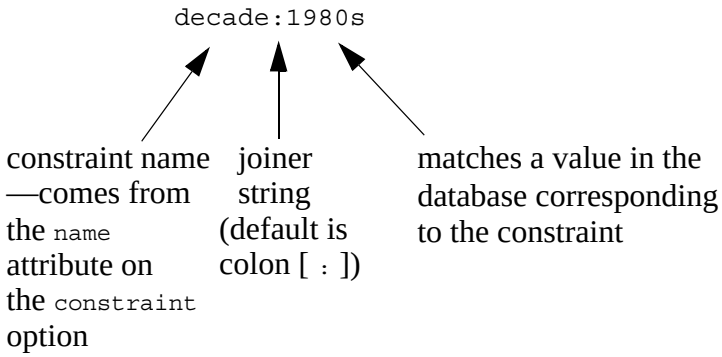
A *constraint* is a mechanism the Search API uses to define ways of constraining a search based on a slice of the database. Constraints provide the Search API with information about your database and specify how to query against those details of the database. They are designed to take advantage of range indexes, other configuration objects (such as word lexicons, collection lexicons, and fields) that exist in the database, and the structures of documents in the database (for example, element values, attribute values, words, and so on). Constraints are primarily used for the following purposes:

- To provide a way to specify the constraint in the search grammar. For information on search parsing and grammar, see "Automatic Query Text Parsing and Grammar" on page 18.
- To return information designed to be used in creating facets in an application. For information on facets, see "Constrained Searches and Faceted Navigation" on page 20.
- To enhance search suggestions made by `search:suggest`. For information on search suggestions, see "Search Term Completion" on page 22.

Each constraint is named, and the name must be unique across all operators and constraints in your options node. Constraint names must not contain whitespace. When you specify a constraint as query text in a Search API call, you use the name as a constraint in the search grammar followed by the `apply="constraint"` *joiner* string (a colon character [`:`] by default). The joiner string joins the constraint (or the operator) with its value. For example, the following query text:

```
decade:1980s
```

specifies the constraint named `decade` with a value of `1980s`. The following figure shows each portion of the constraint query text:



For more details about the search grammar, see "Automatic Query Text Parsing and Grammar" on page 18 and "Search Grammar" on page 56.

The following table lists the types of constraints you can build with the Search API.

<table>
<tr><th>Constraint</th><th>Description</th><th>cts:query Equivalent for constraint</th><th>Lexicon API Equivalent for Facets</th></tr>
<tr><td rowspan="2">value</td><td>Constrains on an element value or on an attribute value or on a field value.</td><td>cts:element-value-query, cts:element-attribute-value-query, cts:field-value-query</td><td>No facets for value constraints.</td></tr>
<tr><td colspan="3">Example value constraint:

<pre>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="my-value">
    <value>
      <element ns="my-namespace" name="my-localname"/>
    </value>
  </constraint>
</options>
</pre>
For more details, see "Value Constraint Example" on page 31</td></tr>
<tr><td rowspan="2">word</td><td>Constrains on a word-query of either element, attribute, or field.</td><td>cts:element-word-query, cts:element-attribute-word-query, cts:field-word-query</td><td>No facets for word constraints.</td></tr>
<tr><td colspan="3">Example word constraint:

<pre>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="name">
    <word>
      <element ns="http://authors-r-us.com" name="name"/>
    </word>
  </constraint>
</options>
</pre>
For more details, see "Word Constraint Examples" on page 31</td></tr>
</table>

<table>
<tr><th>Constraint</th><th>Description</th><th>cts:query Equivalent for constraint</th><th>Lexicon API Equivalent for Facets</th></tr>
<tr><td rowspan="2">collection</td><td>Requires the collection lexicon to be enabled in the database.</td><td>cts:collection-query</td><td>cts:collections</td></tr>
<tr><td colspan="3">Example collection constraint:

<pre><options xmlns="http://marklogic.com/appservices/search">
  <constraint name="subject">
    <collection prefix="/my-collections/"/>
  </constraint>
</options></pre>
For more details, see “Collection Constraint Example” on page 32</td></tr>
<tr><td>range</td><td>Requires the underlying range index to exist in the database. All range constraints are type aware for the element or attribute values or for the field values, and the constraint can optionally include either bucket or computed-bucket elements. For examples, see “Bucketed Range Constraint Example” on page 33, “Buckets Example” on page 64, “Computed Buckets Example” on page 66. and the search:search options node description in the MarkLogic XQuery and XSLT Function Reference.</td><td>The lexicon APIs, such as cts:element-range-query, cts:element-attribute-range-query, cts:path-range-query, and cts:field-range-query</td><td>cts:element-values, cts:element-attribute-values, cts:values, cts:field-values, cts:element-value-ranges, cts:element-attribute-value-ranges, cts:value-ranges, cts:values cts:field-value-ranges</td></tr>
</table>

<table>
<tr><th>Constraint</th><th>Description</th><th>cts:query Equivalent for constraint</th><th>Lexicon API Equivalent for Facets</th></tr>
<tr><td rowspan="2">element-query</td><td>Restricts qtext to a particular cts:element-query. Requires position indexes enabled on the database for the best performancce.</td><td>cts:element-query</td><td>No facets for element-query constraints.</td></tr>
<tr><td colspan="3">Example element-query constraint:

<pre><options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-element-constraint">
    <element-query name="title" ns="http://my/namespace" />
  </constraint>
</options></pre></td></tr>
<tr><td rowspan="2">properties</td><td>Finds matches on the corresponding properties documents.</td><td>cts:properties-query</td><td>No facets for properties constraints.</td></tr>
<tr><td colspan="3">Example properties constraint:

<pre><options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-property-constraint">
    <properties />
  </constraint>
</options></pre></td></tr>
</table>

<table>
<tr><th>Constraint</th><th>Description</th><th>cts:query Equivalent for constraint</th><th>Lexicon API Equivalent for Facets</th></tr>
<tr><td rowspan="2">geo-attr-pair
geo-elem-pair
geo-elem</td><td>These geospatial constraints find matches on geospatial data. To use as a facet, the <constraint> element requires a <heatmap> child.</td><td>cts:element-attribute
-pair-geospatial
-query
cts:element-pair
-geospatial-query
cts:element-
geospatial-query
cts:element-child-
geospatial-query</td><td>cts:element-attribute
-pair-geospatial
-boxes
cts:element-pair
-geospatial-boxes
cts:element-
geospatial-boxes</td></tr>
<tr><td colspan="3">Example geo-* constraints:

<pre>
<options xmlns="http://marklogic.com/appservices/search">
<constraint name="my-geo-attr-pair">
<!-- Uses cts:element-attribute-pair-geospatial-query, and
cts:element-attribute-pair-geospatial-boxes for the
heatmap facet. -->
  <geo-attr-pair>
    <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"
             latdivs="4" londivs="4"/>
    <facet-option>empties</facet-option>
    <parent ns="ns1" name="elem1"/>
    <lat ns="ns2" name="attr2"/>
    <lon ns="ns3" name="attr3"/>
  </geo-attr-pair>
</constraint>
<constraint name="geo-elem-child">
  <geo-elem>
    <parent ns="" name="g-elem-child-parent" />
    <element ns="" name="g-elem-child-point" />
  </geo-elem>
</constraint>
</options>
</pre></td></tr>
<tr><td>custom</td><td>Create your own type of constraint by implementing your own functions for parsing and for creating facets. For an example, see “Creating a Custom Constraint” on page 45.</td><td>Depends on what your custom code implements</td><td>Depends on what your custom code implements</td></tr>
</table>

Constraints are designed to be fast. When they have facets, they must generate fast and accurate counts and distinct values. Therefore the constraints that allow facets require a range index on the element or attribute on which they apply, or require a particular lexicon to exist in the database. Other constraints (value and word constraints) do not require any special indexing, and they cannot be used to create facets.

When the Search API parses a constraint (using `search:parse` or `search:search` for example), it looks for the joiner string and then applies the value to the right of the joiner string, parsing the value as a `cts:query`. If the constraint is not defined in your options node, then the Search API treats the joiner string as part of the whitespace-separated string. For example:

```
search:parse('unrecognized-constraint:hello')
=>
<cts:word-query qtextref="cts:text"
       xmlns:cts="http://marklogic.com/cts">
   <cts:text>unrecognized-constraint:hello</cts:text>
</cts:word-query>
```

If the constraint is not defined and your options node and the value is quoted text, then the Search API ignores the constraint and the joiner when parsing the query, but saves the original text as an attribute. For example:

```
search:parse('unrecognized-constraint:"hello world"')
=>
<cts:word-query qtextpre="unrecognized-constraint:&quot;"
     qtextref="cts:text" qtextpost="&quot;"
     xmlns:cts="http://marklogic.com/cts">
   <cts:text>hello world</cts:text>
</cts:word-query>
```

The following examples show constraints of the following types:

- Value Constraint Example
- Word Constraint Examples
- Collection Constraint Example
- Bucketed Range Constraint Example
- Exact Match (Unbucketed) Range Constraint Example

For an example of a custom constraint, see “Creating a Custom Constraint” on page 45.

2.2.2.1 Value Constraint Example

The following options node defines two value constraints: one for an element and one for an attribute.

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="my-value">
    <value>
      <element ns="my-namespace" name="my-localname"/>
    </value>
  </constraint>
  <constraint name="my-attribute-value">
    <value>
      <attribute ns="" name="my-attribute"/>
      <element ns="my-namespace" name="my-localname"/>
    </value>
  </constraint>
</options>
```

Using these constraints, you can issue query text such as the following (from `search:search` or `search:parse`, for example) to use these constraints:

```
my-value:"This is an element value."
my-attribute-value:123456
```

Both parts of the above query text would match the following document:

```
<my-document xmlns="my-namespace">
  <my-localname>This is an element value.</my-localname>
  <my-localname my-attribute="123456"/>
</my-document>
```

2.2.2.2 Word Constraint Examples

The following options node defines two word constraints: one for a `cts:element-word-query` and one for a `cts:field-word-query`:

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="name">
    <word>
      <element ns="http://authors-r-us.com" name="name"/>
    </word>
  </constraint>
  <constraint name="description">
    <word>
        <field name="my-field"/>
    </word>
  </constraint>
</options>
```

Using these constraints, you can issue query text such as the following (from `search:search` or `search:parse`, for example) to use these constraints:

```
name:raymond
description:author
```

The first query text above would match the following document (because a `cts:word-query("raymond")` would match):

```
<my-document xmlns="http://authors-r-us.com">
  <name>Raymond Carver</name>
</my-document>
```

The second query text above matches the above document if the `name` element was part of the field named `my-field`. For details on fields, see Fields Database Settings in the *Administrator's Guide*.

2.2.2.3 Collection Constraint Example

The following options node defines a collection constraint, which allows you to constrain your search to documents that are in a specified collection. To use this constraint, the collection lexicon must be enabled in the database, otherwise an exception is thrown. If `prefix` is an attribute to the `collection` element in the constraint, then the collection name is derived from the `prefix` concatenated with the constraint value.

One use for a collection constraint is to allow faceted navigation based on collections. For example, if you have collections based on subjects (for example, one called `history`, one called `math`, and so on), then you can use a collection constraint to narrow the search to one of the subjects.

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="subject">
    <collection prefix="/my-collections/"/>
  </constraint>
</options>
```

Assuming that all documents in your database have collection URIs that begin with the string `/my-collections/` like the following:

```
/my-collections/math
/my-collections/economics
/my-collections/zoology
```

Then the following query text examples will match documents in the corresponding collections:

```
subject:math
subject:economics
subject:zoology
```

If the database contains no documents in the specified collection, then the search returns no matches. For information on collections, see "Collections" on page 241.

2.2.2.4 Bucketed Range Constraint Example

Range constraints operate on typed element or attribute values that have a corresponding range index in the database. Without the correct range index, range constraints will throw a runtime exception. Range constraint values can match on either all of the individual values for the element or attribute, or on specified *buckets*, which are named ranges of values. There are two types of buckets, specified with the `bucket` and `computed-bucket` elements in the `range` constraint specification. The `bucket` specification takes absolute ranges, and the `computed-bucket` specification takes ranges that are relative to a given time. For more information about `computed-bucket` range constraints, see "Computed Buckets Example" on page 66.

The following example uses `search:parse` with an options node that contains a `bucket` range constraint. The following example is generated from the Oscars sample application, built using Application Builder:

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

search:parse('decade:1980s',
<options xmlns="http://marklogic.com/appservices/search">

  <constraint name="decade">
    <range type="xs:gYear" facet="true">
      <bucket lt="1930" ge="1920" name="1920s">1920s</bucket>
      <bucket lt="1940" ge="1930" name="1930s">1930s</bucket>
      <bucket lt="1950" ge="1940" name="1940s">1940s</bucket>
      <bucket lt="1960" ge="1950" name="1950s">1950s</bucket>
      <bucket lt="1970" ge="1960" name="1960s">1960s</bucket>
      <bucket lt="1980" ge="1970" name="1970s">1970s</bucket>
      <bucket lt="1990" ge="1980" name="1980s">1980s</bucket>
      <bucket lt="2000" ge="1990" name="1990s">1990s</bucket>
      <bucket ge="2000" name="2000s">2000s</bucket>
      <facet-option>limit=10</facet-option>
      <attribute ns="" name="year"/>
      <element ns="http://marklogic.com/wikipedia" name="nominee"/>
    </range>
  </constraint>
</options>)
```

This query returns the following `cts:query`:

```
<cts:and-query qtextconst="decade:1980s"
    xmlns:cts="http://marklogic.com/cts">
  <cts:element-attribute-range-query qtextconst="decade:1980s"
      operator="&gt;=">
    <cts:element xmlns:_1="http://marklogic.com/wikipedia">
     _1:nominee</cts:element>
    <cts:attribute>year</cts:attribute>
    <cts:value xsi:type="xs:gYear"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    1980</cts:value>
  </cts:element-attribute-range-query>
  <cts:element-attribute-range-query qtextconst="decade:1980s"
    operator="&lt;">
    <cts:element xmlns:_1="http://marklogic.com/wikipedia">
     _1:nominee</cts:element>
    <cts:attribute>year</cts:attribute>
    <cts:value xsi:type="xs:gYear"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     1990</cts:value>
  </cts:element-attribute-range-query>
</cts:and-query>
```

See the Oscars sample application that you generate from Application Builder for sample data against which you can run this query. For other range constraint examples, see “Buckets Example” on page 64 and “Computed Buckets Example” on page 66, and the following example.

2.2.2.5 Exact Match (Unbucketed) Range Constraint Example

The following example shows an exact match year range constraint against the Oscars sample application. It returns results that match the year 1964. To see the output, run this query against the Oscars database.

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

let $options :=
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="year">
   <range type="xs:gYear" facet="true">
   <facet-option>limit=10</facet-option>
   <attribute ns="" name="year"/>
   <element ns="http://marklogic.com/wikipedia"
            name="nominee"/>
   </range>
  </constraint>
</options>
return
search:search("year:1964", $options)
```

2.2.3 Operator Options

Search operators allow you to specify in the search grammar operators to provide runtime, user-controlled configuration and search choices. A typical search operator might control sorting, thereby allowing the user to specify the sort order directly in the query text. For example, you might have an operator named `sort` that allows you to sort by relevance or by date, with the following options XML:

```
<options xmlns="http://marklogic.com/appservices/search">
 <search:operator name="sort">
   <search:state name="relevance">
      <search:sort-order>
         <search:score/>
      </search:sort-order>
   </search:state>
   <search:state name="date">
      <search:sort-order direction="descending" type="xs:dateTime">
         <search:element ns="my-ns" name="date"/>
      </search:sort-order>
      <search:sort-order>
         <search:score/>
      </search:sort-order>
   </search:state>
 </search:operator>
</options>
```

This operator options XML allows you to add text like the following to the search string, and the Search API will parse the string and sort it according to the operator specification.

```
sort:date

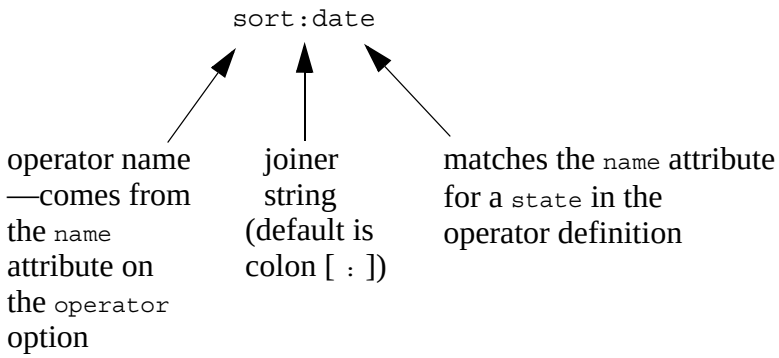
sort:relevance
```

Each operator is named, and the name must be unique across all operators and constraints in your options node. When you specify an operator as query text in a Search API call, you use the name as an operator in the search grammar followed by the `apply="constraint"` *joiner* string (a colon character [`:`] by default). The joiner string joins the operator (or the constraint) with its value. For example, the following query text:

```
sort:date
```

specifies using the operator named `sort` with a value of `date`.

The following figure shows each portion of the operator query text:



For more details about the search grammar, see "Automatic Query Text Parsing and Grammar" on page 18 and "Search Grammar" on page 56.

The `search:state` options element is a child of the `search:operator` element, and the following options XML elements are allowed as a child of `search:state` element:

- `additional-query`
- `debug`
- `forest`
- `page-length`
- `quality-weight`
- `searchable-expression`
- `sort-order`
- `transform-results`

Operators use the same syntax as constraints, but control other aspects of the search (for example, the sort order) besides which results are returned.

2.2.4 Return Options

You can specify a number of options that control what is returned from the Search API. These include the following boolean options:

- `<return-constraints>`
- `<return-facets>`
- `<return-metrics>`
- `<return-qtext>`
- `<return-query>`
- `<return-results>`
- `<return-similar>`

Setting each option to `true` returns the specified option in the `search:search` response element, setting to `false` omits them from the response. For example, the following specifies to return query statistics and facets in the result, but not to return the search hits:

```
<options xmlns="http://marklogic.com/appservices/search">
   <return-metrics>true</return-metrics>
   <return-facets>true</return-facets>
   <return-results>false</return-results>
</options>
```

Only the needed parts of the response are computed, so if you do not return results (as in the above example) or do not return something else, then the work needed to perform that part of the response is not done, and the search runs faster.

For details on each return option, including their default values, see the `search:search` function documentation in *MarkLogic XQuery and XSLT Function Reference*.

2.2.5 Searchable Expression Option

Use the `<searchable-expression>` option to specify what expression to search over and what is returned in the search results. The expression corresponds to the first parameter to `cts:search`, and must be a fully searchable expression. For details on fully searchable expressions, see Fully Searchable Paths and cts:search Operations in *Query Performance and Tuning Guide*.

By default, the Search API searches over the whole database (`fn:collection()`). In most cases, your searchable-expression should search over fragment roots, although searching below fragment roots is allowed.

The following example shows a searchable expression that searches over both CITATION elements and html elements:

```
<searchable-expression xmlns:xh="http://www.w3.org/1999/xhtml">
    /(xh:html | CITATION)
</searchable-expression>
```

If an expression is not fully searchable, it will throw an `XDMP-UNSEARCHABLE` exception at runtime.

2.2.6 Fragment Scope Option

You can specify a `<fragment-scope>` option which controls the fragments over which a search or a constraint operates. A `fragment-scope` can be either `documents` or `properties`. By default, the scope is `documents`. A `fragment-scope` of `documents` searches over documents fragments, and a `fragment-scope` of `properties` searches over properties fragments.

There are two types of `fragment-scope` options: a global fragment scope, which applies to the both the search and any constraints in the search, and a local fragment scope, which applies to a given constraint. A global `fragment-scope` is specified as a child of `<options>`, and a local fragment scope is specified as a child of a contraint kind (for example, a child of `<range>`, `<value>`, or `<word>`). Any local fragment scope will override the global fragment scope.

A local fragment scope of `properties` on a range constraint with a global fragment scope of `documents` allows you to create a facet on data that is in a properties fragment. For example, the following query returns results from documents and a dateTime last-modified facet from the `prop:last-modified` system property:

```
xquery version "1.0-ml";
import module namespace search =
"http://marklogic.com/appservices/search"
     at "/MarkLogic/appservices/search/search.xqy";

search:search("the",
<options xmlns="http://marklogic.com/appservices/search">
<fragment-scope>documents</fragment-scope>
  <constraint name="last-modified">
    <range type="xs:dateTime">
      <element ns="http://marklogic.com/xdmp/property"
               name="last-modified"/>
      <fragment-scope>properties</fragment-scope>
    </range>
  </constraint>
  <debug>true</debug>
</options>)
```

2.2.7 Modifying Your Snippet Results

The `transform-results` option allows you to specify options for the *snippet* code for your application. A *snippet* is the search result blurb (an abbreviated and highlighted summary) that typically comes up in search results. A snippet is created by taking the matching search result node and running it through transformation code. The transformation typically displays the portion of the result you want in your results page, perhaps highlighting the query matches and showing some text around it, often discarding the rest of the result. This section describes the following ways to control and modify the snippet results from the Search API:

- Specifying transform-results Options
- Specifying Your Own Code in transform-results

2.2.7.1 Specifying transform-results Options

By default, the Search API has its own code to take search result matches and transform them into snippets used in the search results. By default, the Search API uses the `apply="snippet"` attribute on the `transform-results` option. Snippets tend to be very application specific, and the built-in `apply="snippet"` option has several parameters that you can control with a `transform-results` options node.

The following is the default `transform-results` options node:

```
<transform-results apply="snippet">
    <per-match-tokens>30</per-match-tokens>
    <max-matches>4</max-matches>
    <max-snippet-chars>200</max-snippet-chars>
    <preferred-elements/>
</transform-results>
```

The following table describes the `transform-results` options when `apply="snippet"`, each of which is configurable at search runtime by specifying your own values for the options:

`transform-results` Child Element	Description
`per-match-tokens`	Maximum number of tokens (typically words) per matching node that surround the highlighted term(s) in the snippet.
`max-matches`	The maximum number of nodes containing a highlighted term that will display in the snippet.
`max-snippet-chars`	Limit total snippet size to this many characters.

`preferred-elements`	Specify zero or more elements that the snippet algorithm looks in first to find matches. For example, if you want any matches in the `TITLE` element to take preference, specify `TITLE` as a preferred element as in the following sample:
	`<transform-results apply="snippet">`
	`  <preferred-elements>`
	`  <element ns="" name="TITLE"/>`
	`  </preferred-elements>`
	`</transform-results>`

There are also three other built-in snippetting options, which are exposed as attributes on the `transform-results` options node:

- `apply="raw"`
- `apply="empty-snippet"`
- `apply="metadata-snippet"`

Note: The `apply` attribute for the `transform-results` element is only applicable to the `search:search` and `search:resolve` functions; `search:snippet` always uses the default snippitting option of `snippet` and ignores anything specified in the `apply` attribute.

The `apply="raw"` snippetting option looks as follows:

```
<transform-results apply="raw" />
```

The `apply="raw"` option returns the whole node (with *no* highlighting) in the `search:response` output. You can then take the node and do your own transformation on it, or just return it as-is, or whatever else makes sense for your application.

The `apply="empty-snippet"` snippetting option is as follows:

```
<transform-results apply="empty-snippet" />
```

The `apply="empty-snippet"` option returns no result node, but does return an empty `search:snippet` element for each `search:result`. The `search:result` wrapper element does have the information (for example, the URI and path to the node) needed to access the node and perform your own transformation on the matching search node(s), so you can write your own code outside of the Search API to process the results.

The `apply="metadata-snippet"` snippetting option is as follows:

```
<transform-results apply="metadata-snippet">
  <preferred-elements>
    <!-- Specify namespace and localname for elements that exist
         in properties documents -->
    <element ns="http://my.namespace" name="my-local-name"/>
  </preferred-elements>
</transform-results>
```

The `apply="metadata-snippet"` option returns the specified preferred elements from the properties documents. If no `<preferred-elements>` element is specified, then the `metadata-snippet` option returns the `prop:last-modified` element for its snippet, and if the `prop:last-modified` element does not exist, it returns an empty snippet.

2.2.7.2 Specifying Your Own Code in transform-results

If the default snippet code does not meet your application requirements, you can use your own snippet code to use for a given search.

To specify your own snippet code, use the design pattern described in “Search Customization Via Options and Extensions” on page 22. The function you implement must have a signature compatible with the following signature:

```
declare function search:snippet(
   $result as node(),
   $ctsquery as schema-element(cts:query),
   $options as element(search:transform-results)?
) as element(search:snippet)
```

The Search API will pass the function the result node and the `cts:query` XML representation and your custom function can transform it any way you see fit. An options node that specifies a custom transformation looks as follows:

```
<options xmlns="http://marklogic.com/appservices/search">
  <transform-results apply="my-snippet" ns="my-namespace"
      at="/my-snippet.xqy">
  </transform-results>
</options>
```

If you create a custom function, you can optionally pass in options to your function by adding them as children of the `transform-results` option. The Search API will pass the `transform-results` element into your function, and if you want to use any part of the option, you can write code to parse the option and extract whatever you need from it.

2.2.8 Other Search Options

There are several other options in the Search API, including `additional-query` (an additional `cts:query` combined as an and-query to the active query in your search), `term-option` (pass any of the `cts:query` options such as `case-sensitive` to your `cts:query`), and others. For details on what the other options do, see the *MarkLogic XQuery and XSLT Function Reference*.

2.3 Search Term Completion Using search:suggest

The `search:suggest` function returns suggestions that match a wildcarded string, and it is used in query-completion applications. For an example of an application that uses search:suggest, see the Oscars sample application that you can generate with Application Builder, as described in Building the Oscars Sample Application in the *Application Builder Developer's Guide*.

A typical way to use the `search:suggest` function in an application is to have a Javascript event listen for changes in the text box, and then upon those changes it asynchronously submits a `search:suggest` call to MarkLogic Server. The result is that, after every letter is typed in, new suggestions appear in the user interface. The remainder of this sections describes the following details of the search:suggest function:

- default-suggestion-source Option
- Choose Suggestions With the suggestion-source Option
- Use Multiple Query Text Inputs to search:suggest
- Make Suggestions Based on Cursor Position
- search:suggest Examples

2.3.1 default-suggestion-source Option

To use `search:suggest`, it is best to specify a `default-suggestion-source`. The Search API uses the `default-suggestion-source` to look for search term suggestions. If no `default-suggestion-source` is specified, then any call to `search:suggest` returns only suggestions for constraints and operators, or if there are none, then it returns the empty sequence. The `search:suggest` function suggests constraint and operator names if they match the query text string, and in the case of range index-based constraints, it will suggest matching constraint values. For details on the syntax of the `default-suggestion-source` option, see the `search:search` options documentation in the *MarkLogic XQuery and XSLT Function Reference*.

For best performance, especially on large databases, use with a `default-suggestion-source` with a `range` or `collection` instead of one with a `word` lexicon.

The following `default-suggestion-source` example uses the string range index on the attribute named `my-attribute` as a source for suggesting terms. Range suggestion sources tend to perform the best, especially for large databases. The range index must exist or an exception is thrown at search runtime.

```
<default-suggestion-source>
  <range type="xs:string">
    <element ns="my-namespace" name="my-localname"/>
    <attribute ns="" name="my-attribute"/>
   </range>
</default-suggestion-source>
```

The following example specifies using a field lexicon to look for search term suggestions. Fields can work well for suggestion sources, especially if the field is a relatively small subset of the whole database. A field word lexicon for the specified field must exist or an exception is thrown at search runtime.

```
<default-suggestion-source>
    <word collation="http://marklogic.com/collation/">
        <field name="my-field"/>
    </word>
</default-suggestion-source>
```

2.3.2 Choose Suggestions With the suggestion-source Option

For some applications, you want to have a very specific list from which to choose suggestions for a particular constraint. For example, you might have a constraint named name that has millions of unique values, but perhaps you only want to make suggestions for a specific 500 of them. In such cases, you can specify the `suggestion-source` option to override the suggestions that `search:suggest` returns for query text matching values in that constraint.

You specify the constraint to override in the in the `name` attribute of the `suggestion-source` element. For example, the following options specify to use the values from the `short-list-name` element instead of from the `name` element when make suggestions for the `name` constraint.

```
<constraint name="name">
   <range collation="http://marklogic.com/collation"
          type="xs:string" facet="true">
      <element ns="my-namespace" name="fullname"/>
   </range>
 </constraint>
 <suggestion-source ref="name">
     <range collation="http://marklogic.com/collation"
          type="xs:string" facet="true">
      <element ns="my-namespace" name="short-list-name"/>
   </range>
 </suggestion-source>
```

For cases where you have a named constraint to use for searching and facets, but might want to use a slightly (or completely) different source for type-ahead suggestions without needing to re-parse your search terms, use the `suggestion-source` option.

If you want a particular constraint to not return suggestion, add an empty `suggestion-source` for that constraint:

```
<suggestion-source ref="socialsecuritynumber" />
```

2.3.3 Use Multiple Query Text Inputs to search:suggest

You can specify one or more query text parameters to `search:suggest`. When you specify a sequence of more than one query text for `search:search`, the first item (or the one corresponding to the `$focus` parameter) specifies the text to match against the suggestion source. Each of the other items in the sequence is parsed as a `cts:query`, and that query is used to constrain the search suggestions from the text-matching query text. Note that this is different from the other Search API functions, which combine multiple query texts with a `cts:and-query`.

Consider a user interface that looks as follows:



The search text box on top is where the user types text. The lower check box might be another control that the user can use to specify the decade. The `decade:1980s` text shown might be the query text that is the result of that user interface control (possibly from a facet, for example). You can then construct a `search:suggest` call from this user interface that uses the `decade:1980s` text as a constraint to the terms matching `comp` (from the specified suggestion source). The following is a `search:suggest` call that can be generated from this example:

```
search:suggest(("comp", "decade:1980s"), $options)
```

This ends up returning suggestions that match `comp*` on fragments that match `search:parse("decade:1980s")`. For example, it might return a sequence including the words `competent`, `component`, and `computer`.

2.3.4 Make Suggestions Based on Cursor Position

The `search:suggest` function makes search suggestions based on the position of the cursor (which you specify with the `$cursor-position` parameter. The idea is that when the user changes the cursor position, you should suggest terms based on where the user is currently entering text.

2.3.5 search:suggest Examples

The following are some example `search:suggest` queries with sample output.

Assume a constraint named filesize for the following example:

```
query:suggest("fi", $options)

(: Returns the "filesize" constraint name first, followed
   by words from the default source of word suggestions:

  ("filesize:", "field", "file", "fitness", "five",)  :)
```

The following example shows how search:suggest works with bucketed `range` constraints:

```
(: Assume $options contains the following:
  <constraint name="date">
   <range type="xs:dateTime">
      <bucket name="today">
      <bucket name="yesterday">
      <bucket name="thismonth">
      <bucket name="thisyear">
...

:)
search:suggest("date:", $options)
(: bucket names from the "date" range constraint are
   used to create suggestions

("date:thismonth", "date:thisyear", "date:today", "date:yesterday") :)
```

2.4 Creating a Custom Constraint

By default, the Search API supports many, but not all, types of constraints. If you need to create a constraint for which there is not one pre-defined in the Search API, there is a mechanism to extend the Search API to use your own constraint type. This type of constraint, called a `custom` constraint, requires you to write XQuery functions to implement your own custom parsing and to generate your own custom facets. You specify your function implementations in the options XML as follows:

```
<constraint name="my-custom">
    <custom facet="true"> <!-- or false -->
       <parse apply="parse" ns="..." at="..." />
       <start-facet apply="start" ns="..." at="..." />
       <finish-facet apply="finish" ns="..." at="..." />
    </custom>
</constraint>
```

The three functions you need to implement are `parse`, `start-facet`, and `finish-facet`. The `apply` attribute specifies the localname of the function, the `ns` attribute specifies the namespace, and the `at` attribute specifies the location of the module containing the function. This section describes how to create a custom constraint and includes some example code for creating a custom geospatial constraint. This section includes the following parts:

- Implementing the parse Function
- Implementing the start-facet Function
- Implementing the finish-facet Function
- Example: Creating a Simple Custom Constraint
- Example: Creating a Custom Constraint for Structured Queries
- Example: Creating a Custom Constraint Geospatial Facet

2.4.1 Implementing the parse Function

The purpose of the `parse` function is to parse the custom constraint and generate the correct `cts:query` from the query text.

String queries and structured queries do not use the same custom constraint parsing interface, so you must know which type(s) of query your function will support. If you want to support both types of query, you can create a parse function capable of handling both, define different custom constraints in the same query options, or use different query options.

This section covers the following topics:

- Implementing a String Query parse Function
- Implementing a Structured Query parse Function

- Implementing a Multi-Format parse Function

2.4.1.1 Implementing a String Query parse Function

For parsing your custom constraint in a string query, the custom function you implement must have a signature compatible with the following signature:

```
declare function example:parse-string(
  $constraint-qtext as xs:string,
  $right as schema-element(cts:query))
as schema-element(cts:query)
```

You can use any namespace and localname for the function, but the number and order of the parameters must be compatible and the return type must be compatible.

The `$constraint-qtext` parameter is the constraint name and joiner part of the query text *for the portion of the query pertaining to this constraint*. For example, if the constraint name is `geo` and the joiner is the default joiner, then the value of `$constraint-qtext` will be `geo:`. The `$constraint-qtext` value is used in the `qtextconst` attribute, which is needed by `search:unparse` to re-create the query text from the annotated `cts:query`.

The `$right` parameter contains the value of the constraint parsed as a `cts:query`. In other words, it is the text to the right of what is passed into `$constraint-qtext` in the query text, and then that text is parsed by the Search API as a `cts:query`, and returned to the parse function as the XML representation of a `cts:query`. The value of `$right` is what the parse function uses for generating its custom `cts:query`. For details on how `cts:query` constructors work, see “Composing cts:query Expressions” on page 121.

The `parse` function you implement takes the `cts:query` from the `$right` parameter, parses it as you see fit, and then returns a `cts:query` XML element. For example, if the value of `$right` is as follows:

```
<cts:word-query>
  <cts:text>1@2@3@4</cts:text>
</cts:word-query>
```

Your code must process the `cts:text` element to construct the cts:query you need. For example, you can tokenize on the `@` character of the `cts:text` element, then use each value to construct a part of the query. As part of constructing the `cts:query`, you can optionally add `cts:annotation` elements and annotation attributes to the `cts:query` you generate. These annotations allow the Search API to unparse the `cts:query` back into its original form. If you do not add the proper annotations, then `search:unparse` might not return the original query text. For a sample function that does something similar, see “Example: Creating a Custom Constraint Geospatial Facet” on page 53.

2.4.1.2 Implementing a Structured Query parse Function

For parsing your custom constraint in a structured query, the custom function you implement must have a signature compatible with the following signature:

```
declare function example:parse-structured(
  $query-elem as element(),
  $options as element(search:options))
as schema-element(cts:query)
```

You can use any namespace and localname for the function, but the number and order of the parameters must be compatible and the return type must be compatible.

The `$query-elem` parameter is `custom-constraint-query` structured query that references your constraint. For details, see “custom-constraint-query” on page 111.

2.4.1.3 Implementing a Multi-Format parse Function

You can create a single parse function capable of handling either a string query or a structured query as input by generalizing the parse function interface to accomodate both and using the XQuery `instance of` operator to determine the query type.

The following parse function skeleton generalizes the input query as an `item()` and the second parameter, which can be either a `cts:query` or `search:options`, to `element()`, and then uses `instance of` to detect the actual input query type:

```
declare function example:combo-parser(
  $query as item(),
  $right-or-option as element())
as schema-element(cts:query)
{
  if ($query instance of element(search:query))
  then ... (: handle as structured query :)
  else if ($query instance of xs:string)
  then ... (: handle as string query :)
  else ... (: error :)
};
```

Once you determine the input query type, coerce the second parameter to the correct type and parse your query as you would in the appropriate string or structured query parse function, as described in “Implementing a String Query parse Function” on page 46 and “Implementing a Structured Query parse Function” on page 47.

2.4.2 Implementing the start-facet Function

The sole purpose of the `start-facet` function is to make a lexicon API call that returns the values and counts that are used in constructing a facet. For details on lexicons, see “Browsing With Lexicons” on page 146. The custom function you implement must have a signature compatible with the following signature:

```
declare function my-namespace:start-facet(
  $constraint as element(search:constraint),
  $query as cts:query?,
  $facet-options as xs:string*,
  $quality-weight as xs:double?,
  $forests as xs:unsignedLong*)
as item()*
```

You can use any namespace and localname for the function, but the number and order of the parameters must be compatible and the return type must be compatible.

Each of the parameters is passed into the function by the Search API. The `$query` parameter includes any custom query your `parse` function implements, combined with any other query that the Search API generates (which depends on other options passed into the original search such as `additional-query`). All other parameters are specified in the `search:options` XML node passed into the Search API call. You can choose to use them or not, as is needed to perform your custom action.

When implementing a lexicon call in the `start-facet` function, you must add the `"concurrent"` option to the `$facet-options` parameter and use the combined sequence as input to the `$options` parameter of the lexicon API. The `"concurrent"` option takes advantage of concurrency, and can greatly speed performance, especially for applications with many facets. For a sample function, see “Example: Creating a Custom Constraint Geospatial Facet” on page 53.

Note: The `start-facet` function is optional, but is the recommended way to create a custom facet that uses any of the MarkLogic Server lexicon functions. If you do not use the `start-facet` function, then the `finish-facet` function must do all of the work to construct the facet (including constructing the values for the facet). For details on the lexicon functions, see the *MarkLogic XQuery and XSLT Function Reference* and “Browsing With Lexicons” on page 146.

2.4.3 Implementing the finish-facet Function

The `finish-facet` function takes input from the `start-facet` function (if it is used) and constructs the `facet` element. This function must have a signature compatible with the following signature:

```
declare function my-namespace:finish-facet(
  $start as item()*,
  $constraint as element(search:constraint),
  $query as cts:query?,
  $facet-options as xs:string*,
  $quality-weight as xs:double?,
  $forests as xs:unsignedLong*)
as element(search:facet)
```

You can use any namespace and localname for the function, but the number and order of the parameters must be compatible and the return type must be compatible.

The parameters are passed into the function by the Search API. The `$query` parameter includes any custom query your `parse` function implemented, combined with any other query that the Search API generates (which depends on other options passed in to the original search such as `additional-query`). All of the remaining parameters are specified in the `search:options` XML passed into the Search API call. You can choose to use them or not, as is needed to perform your custom action. For a sample function, see "Example: Creating a Custom Constraint Geospatial Facet" on page 53.

If you do not use a `start-facet` function, then the empty sequence is passed in for the `$start` parameter. If you are not using a `start-facet` function, then the `finish-facet` function is responsible for constructing the values and counts used in the facet, as well as creating the facet XML.

2.4.4 Example: Creating a Simple Custom Constraint

The following is a library module that implements a very simple custom constraint. This constraint adds a `cts:directory-query` for the values specified in the constraint. This constraint has no facets, so it does not need the `start-facet` and `finish-facet` functions. This code does very minimal parsing; your actual code might parse the `$right` query more carefully.

```
xquery version "1.0-ml";
module namespace my="my-namespace";

declare variable $prefix := "/mydocs/" ;

declare function part(
  $constraint-qtext as xs:string,
  $right as schema-element(cts:query))
as schema-element(cts:query)
{
let $query :=
<root>{
  let $s := fn:string($right//cts:text/text())
  let $dir :=
    if ( $s eq "book")
    then fn:concat($prefix, "book-dir/")
    else if ( $s eq "api")
    then ( fn:concat($prefix, "api-dir1/"),
           fn:concat($prefix, "api-dir2/") )
    (: if it does not match, just constrain on the prefix :)
    else $prefix
  return
  (: make these an or-query so you can look through several dirs :)
    cts:or-query((
    for $x in $dir
    return
      cts:directory-query($x, "infinity")
    ))
    }
</root>/*
return
(: add qtextconst attribute so that search:unparse will work -
   required for some search library functions :)
element { fn:node-name($query) }
  { attribute qtextconst {
      fn:concat($constraint-qtext, fn:string($right//cts:text)) },
    $query/@*,
    $query/node()}
} ;
```

If you put this module in a file named `my-module.xqy` your App Server root, you can run this constraint with the following options node:

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="part">
    <custom facet="false">
      <parse apply="part" ns="my-namespace" at="/my-module.xqy"/>
    </custom>
  </constraint>
</options>
```

The following query text results in constraining this search to the `/mydocs/book-dir/` directory:

```
part:book
```

2.4.5 Example: Creating a Custom Constraint for Structured Queries

The following is a library module that implements a very simple custom constraint to be used with structured queries. This constraint adds a `cts:directory-query` for the values specified in the constraint. This constraint has no facets, so it does not need the `start-facet` and `finish-facet` functions.

```
xquery version "1.0-ml";

module namespace my = "my-namespace";
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

declare variable $prefix := "/mydocs/" ;

declare function part(
  $query-elem as element(),
  $options as element(search:options)
) as schema-element(cts:query)
{
let $query :=
<root>{
  let $s := $query-elem/search:text/text()
  let $dir :=
    if ( $s eq "book")
    then fn:concat($prefix, "book-dir/")
    else if ( $s eq "api")
    then ( fn:concat($prefix, "api-dir1/"),
           fn:concat($prefix, "api-dir2/") )
    (: if it does not match, just constrain on the prefix :)
    else $prefix
  return
  (: make these an or-query so you can look through several dirs :)
    cts:or-query((
    for $x in $dir
    return
```

```
      cts:directory-query($x, "infinity")
    ))
    }
</root>/*
return
(: add qtextconst attribute so that search:unparse will work -
   required for some search library functions :)
element { fn:node-name($query) }
  { attribute qtextconst {
      fn:concat(
        $query-elem/search:constraint-name, ":",
        $query-elem/search:text/text()) },
    $query/@*,
    $query/node()}
} ;
```

If you put this module in a file named `my-module.xqy` your App Server root, you can run this constraint with the following options node:

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="part">
    <custom facet="false">
      <parse apply="part" ns="my-namespace" at="/my-module.xqy"/>
    </custom>
  </constraint>
</options>
```

The following structured query constrains the search to the `/mydocs/book-dir/` directory:

```
<query xmlns="http://marklogic.com/appservices/search">
  <custom-constraint-query>
    <constraint-name>part</constraint-name>
    <text>book</text>
  </custom-constraint-query>
</query>
```

You can use the `return-query` query option to see the `directory-query` generated by the custom constraint. For example, if you add the following to your options node:

```
<return-query>true</return-query>
```

Then the search response will include a query similar to the following:

```
<search:response ...>
  <search:query>
    <cts:or-query xmlns:cts="http://marklogic.com/cts">
      <cts:directory-query depth="infinity">
        <cts:uri>/mydocs/book-dir/</cts:uri>
      </cts:directory-query>
    </cts:or-query>
  </search:query>
```

```
  ...
</search:response>
```

2.4.6 Example: Creating a Custom Constraint Geospatial Facet

The following is a library module that implements a geospatial facet that uses a custom constraint. It tokenizes the constraint value on the `@` character to produce input to the geospatial lexicon function. This is a simplified example, meant to demonstrate the design pattern, not meant for production, as it does not do any error checking to make it more robust at handling user input.

Note: While you could use the code in this example, it is meant as an example of the design patterns you use to create custom constraints. If you want to use a geospatial constraint, use the build-in geospatial contraint types (`geo-attr-pair`, `geo-elem-pair`, and `geo-elem`) as described in "Constraint Options" on page 25.

```
xquery version "1.0-ml";
module namespace geoexample = "my-geoexample";
(:
  Sample custom constraint for this example :

  <constraint name="geo">
     <custom>
       <parse apply="parse" ns="my-geoexample"
              at="/geoexample.xqy"/>
       <start-facet apply="start-facet" ns="my-geoexample"
                    at="/geoexample.xqy"/>
       <finish-facet apply="finish-facet" ns="my-geoexample"
                    at="/geoexample.xqy"/>
        <annotation>
            <regions>
               <region label="A">[0, -180, 30, -90]</region>
               <region label="B">[0, -90, 30, 0]</region>
               <region label="C">[30, -180, 45, -90]</region>
               <region label="D">[30, -90, 45, 0]</region>
               <region label="E">[45, -180, 60, -90]</region>
               <region label="F">[45, -90, 60, 0]</region>
               <region label="G">[45, 90, 60, 180]</region>
               <region label="H">[60, -180, 90, -90]</region>
               <region label="I">[60, -90, 90, 0]</region>
               <region label="J">[60, 90, 90, 180]</region>
            </regions>
        </annotation>
     </custom>
  </constraint>

  This example assumes the presence of an element-pair
  geospatial index, on data structured as follows (note lat/lon
  children of quake):

    <quake>
     <area>0</area>
     <perimeter>0</perimeter>
```

```
      <quakesx020>2</quakesx020>
      <quakesx0201>26024</quakesx0201>
      <catalog_sr>PDE</catalog_sr>
      <year>1994</year>
      <month>6</month>
      <day>11</day>
      <origin_tim>164453.48</origin_tim>
      <lat>61.61</lat>
      <lon>168.28</lon>
      <depth>9</depth>
      <magnitude>4.3</magnitude>
      <mag_scale>mb</mag_scale>
      <mag_source/>
      <dt>1994-06-11T16:44:53.48Z</dt>
    </quake>
:)

declare namespace search = "http://marklogic.com/appservices/search";
(:
   The Search API calls the parse function during the parsing of the
   query text.  It accepts the parsed-so-far query text for this
   constraint, parses that query, and outputs a serialized cts:query
   for the custom part.  The Search API passes the parameters to this
   function based on the custom constraint in the search:options and
   the query text passed into search:search.
:)
declare function geoexample:parse(
  $qtext as xs:string,
  $right as schema-element(cts:query) )
as schema-element(cts:query)
{
    let $point := fn:tokenize(fn:string($right//cts:text), "@")
    let $s := $point[1]
    let $w := $point[2]
    let $n := $point[3]
    let $e := $point[4]
    return
        element cts:element-pair-geospatial-query {
            attribute qtextconst {
                fn:concat($qtext, fn:string($right//cts:text)) },
            element cts:annotation {
               "this is a custom constraint for geo" },
            element cts:element { "quake" },
            element cts:latitude {"lat"},
            element cts:longitude {"lon"},
            element cts:region {
                attribute xsi:type { "cts:box" },
                fn:concat("[", fn:string-join(($s, $w, $n, $e),
                                  ", "), "]")
            },
            element cts:option { "coordinate-system=wgs84" }
        }
};
```

```
(:
  The start-facet function starts the concurrent lexicon evaluation.
:)
declare function geoexample:start-facet(
  $constraint as element(search:constraint),
  $query as cts:query?,
  $facet-options as xs:string*,
  $quality-weight as xs:double?,
  $forests as xs:unsignedLong*)
as item()*
{
  let $latitude-bounds  := (0, 30, 45, 60, 90)
  let $longitude-bounds := (-180, -90, 0, 90, 180)
  return
  cts:element-pair-geospatial-boxes(
        xs:QName("quake"), xs:QName("lat"), xs:QName("lon"),
$latitude-bounds,
        $longitude-bounds, ($facet-options, "concurrent", "gridded"),
        $query, $quality-weight, $forests)
};

(:
  The finish-facet function constructs the facet, based on the
  values from $start returned by the start-facet function.
:)
declare function geoexample:finish-facet(
  $start as item()*,
  $constraint as element(search:constraint),
  $query as cts:query?,
  $facet-options as xs:string*,
  $quality-weight as xs:double?,
  $forests as xs:unsignedLong*)
as element(search:facet)
{
(: Uses the annotation from the constraint to extract the regions :)
  let $labels :=
$constraint/search:custom/search:annotation/search:regions
  return
  element search:facet {
    attribute name {$constraint/@name},
    for $range in $start
    return
    element search:facet-value{
        attribute name {
              $labels/search:region[. eq fn:string($range)]/@label },
        attribute count {cts:frequency($range)}, fn:string($range) }
  }
};
```

To run a custom constraint that references the above custom code, put the above module in the App Server root in a file names `geoexample.xqy` and run the following:

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

let $options :=
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="geo">
     <custom>
       <parse apply="parse" ns="my-geoexample"
              at="/geoexample.xqy"/>
       <start-facet apply="start-facet" ns="my-geoexample"
                    at="/geoexample.xqy"/>
       <finish-facet apply="finish-facet" ns="my-geoexample"
                    at="/geoexample.xqy"/>
        <annotation>
            <regions>
               <region label="A">[0, -180, 30, -90]</region>
               <region label="B">[0, -90, 30, 0]</region>
               <region label="C">[30, -180, 45, -90]</region>
               <region label="D">[30, -90, 45, 0]</region>
               <region label="E">[45, -180, 60, -90]</region>
               <region label="F">[45, -90, 60, 0]</region>
               <region label="G">[45, 90, 60, 180]</region>
               <region label="H">[60, -180, 90, -90]</region>
               <region label="I">[60, -90, 90, 0]</region>
               <region label="J">[60, 90, 90, 180]</region>
            </regions>
        </annotation>
      </custom>
   </constraint>
</options>
return
search:search("geo:1@2@3@4", $options)
```

2.5 Search Grammar

The Search API has a built-in grammar it uses to generate a search query from simple query text, which is typically text entered by an end-user in a simple HTML form. This section describes the default Search API grammar and provides information on how to extend and modify the grammar. It includes the following parts:

- Basic Search Grammar Syntax
- Modifying and Extending the Search Parsing Grammar

2.5.1 Basic Search Grammar Syntax

The basic, out-of-the-box Search API grammar allows you to write applications that take simple text from an application and automatically generate complex queries to perform searches against a database. The following table shows the main parts of the grammar. For more details, as well as information on modifying the default grammar, see "Modifying and Extending the Search Parsing Grammar" on page 58.

Keyword	Example	Description
any terms	`aardvark nose`	Searches for documents matching all terms, combined with a `cts:and-query`. The example matches documents that have both the term `aardvark` and the term `nose`.
`AND`	`aardvark AND nose`	Combines the terms on either side with a `cts:and-query`. This example is eqivalent to the previous example, as `AND` is the default way to combine terms.
`OR`	`aardvark OR nose`	Combines the terms on either side with a `cts:or-query`. The example matches documents that have at least one of either of the terms `aardvark` or `nose`.
`" "`	`"any phrase"`	Anything within the double-quote marks is treated as a phrase. The example matches documents having the phrase `"any phrase"` (without the double-quote marks).

Keyword	Example	Description
`NEAR`	`hello NEAR goodbye`	Matches terms on either side of the NEAR where they are within 10 terms of each other. The example matches documents where `hello` is within 10 terms of `goodbye`. If you add a `/` followed by a number to the keyword, then it uses the number to specify the number of terms for the distance. For example, `hello NEAR/2 goodbye` matches `hello` within 2 terms of `goodbye`.
`:`	`decade:1980s`	Indicates a constraint or operator, and the left side of the constraint or operator keyword (`:`) is the constraint or operator name, the right side is the value.
`-`	`cat -dog`	Indicates a `cts:not-query`. The example matches documents matching `cat` but not matching `dog`.
`( )`	`(cat OR dog) zebra`	Indicates grouping. The example matches documents that have at least one of the terms `cat` or `dog`, and also have the term `zebra`.

You can combine all of these elements of the grammar together to easily for complex queries.

2.5.2 Modifying and Extending the Search Parsing Grammar

You can customize the search parsing grammar by specifying a grammar element in the options XML. The following is the default search grammar (to see the defaults options, run `search:get-default-options()`).

```
<grammar xmlns="http://marklogic.com/appservices/search">
  <quotation>"</quotation>
  <implicit>
    <cts:and-query strength="20" xmlns:cts="http://marklogic.com/cts"/>
  </implicit>
  <starter strength="30" apply="grouping" delimiter=")">(</starter>
  <starter strength="40" apply="prefix"
element="cts:not-query">-</starter>
  <joiner strength="10" apply="infix" element="cts:or-query"
tokenize="word">OR</joiner>
  <joiner strength="20" apply="infix" element="cts:and-query"
tokenize="word">AND</joiner>
  <joiner strength="30" apply="infix" element="cts:near-query"
tokenize="word">NEAR</joiner>
```

```
  <joiner strength="30" apply="near2"
element="cts:near-query">NEAR/</joiner>
  <joiner strength="50" apply="constraint">:</joiner>
  <joiner strength="50" apply="constraint" compare="LT"
tokenize="word">LT</joiner>
  <joiner strength="50" apply="constraint" compare="LE"
tokenize="word">LE</joiner>
  <joiner strength="50" apply="constraint" compare="GT"
tokenize="word">GT</joiner>
  <joiner strength="50" apply="constraint" compare="GE"
tokenize="word">GE</joiner>
  <joiner strength="50" apply="constraint" compare="NE"
tokenize="word">NE</joiner>
</grammar>
```

The default grammar provides a robust ability to generate complex queries. The following are some examples of queries that use the default grammar:

- `(cat OR dog) NEAR vet`

  at least one of the terms `cat` or `dog` within 10 terms (the default distance for `cts:near-query`) of the word `vet`

- `dog NEAR/30 vet`

  the word `dog` within 30 terms of the word `vet`

- `cat -dog`

  the word `cat` where there is no word `dog`

The following table describes the concepts used in the search grammar:

Concept	Description
implicit	The *implicit* grammar element specifies the `cts:query` to use by default to join two search terms together. By default, the Search API uses a `cts:and-query`, but you can change it to any cts:query with the `implicit` grammar option.
starter	A *starter* is a string that appears before a term to denote special parsing for the term, for example, the minus sign (`-`) for negation. Additionally, when used with the `delimiter` attribute, a starter specifies starting and ending strings that separate terms for grouping things together, and allows the grammar to set an order of precedence for terms when parsing a string.

joiner	A *joiner* is a string that combines two terms together. The grammar uses joiners for things like boolean logic:
	`cat AND dog`
	`cat OR dog`
	It also uses joiners for the string that separates a constraint or operator from its value, as described in “Constraint Options” on page 25 and “Operator Options” on page 35. If the `tokenize="word"` attribute is present, then the terms and the joiner must be whitespace-separated; otherwise the parse looks for the joiner string anywhere in the query text.

quotation	The `quotation` string specifies the string to use to indicate the start and end of a phrase. For example, in the default grammar, the following is parsed as a phrase (instead of a sequence of terms combined with an `AND`): `"this is a phrase"`
strength	The *strength* attribute provides the parser with information on which tokens are processed first. Higher strength tokens or groups are processed before lower strength tokens or groups.

The `starter` elements define how to parse portions of the grammar. The `apply` attributes specify the functions to which the `starter` and the `delimiter` apply.

The `joiner` elements define how to parse various operators, constraints, and other operations and specifies the functions that define the joiner’s behavior. For example, if you wanted to change the `OR` joiner above, which joins tokens with a `cts:or-query`, to use the pipe character (`|`) instead, you would substitute the following `joiner` element for the one above:

```
<search:joiner strength="10" apply="infix" element="cts:or-query"
     tokenize="word">|</search:joiner>
```

The `tokenize="word"` attribute specifies that in order for that token to be recognized, it must have whitespace immediately before and after it. Without that attribute, if `OR` was the joiner, then a search for `CORN` would result in a search for `C OR N` (`cts:or-query(("C"), ("N"))`). With joiners used in constraints (for example, the colon character `:`), you probably do not want that, so the `tokenize` attribute is omitted, thus allowing searches like `decade:1990s` to parse as a constraint.

You can add a joiner string to specify the composable `cts:query` elements that take a sequence of queries (`cts:or-query`, `cts:and-query`, or `cts:near-query`) by specifying the element in the `element` attribute on an `apply="infix"` joiner. For example, the following `search:joiner` element specifies a joiner for `cts:near-query`, which would combine the surrounding terms with a `cts:near-query` (and would use the default distance of 10) using the joiner string `CLOSETO`:

```
<search:joiner strength="10" apply="infix" element="cts:near-query"
       tokenize="word">CLOSETO</search:joiner>
```

Using the above joiner specification, the following query text `bicycle CLOSETO shop` would return matches that have `bicycle` and `shop` within 10 words of each other.

By default, the search grammar is very powerful, and implements a grammar similar to the Google grammar. With the customization, you can make it even more powerful and customize it to your specific needs. To add custom parsing, you must implement a function and use the `apply`, `ns`, `at` design pattern (described in “Search Customization Via Options and Extensions” on page 22) and construct a `search:grammar` options node to point to the function(s) you implemented.

2.6 Using Structured Search as an Alternate to cts:query

The annotated `cts:query` that is generated by default from `search:parse` or `search:search` works well for many simple and complex case where you do not need to perform extensive modification to the query. If you want to generate your own query, or if you want to parse your query using different rules from the Search API grammar rules, there is an alternate query you can use called *structured query* (Structured Search). You can generate a structured query either from `search:parse` or by writing your own code that returns a structured query.

For details, see “Searching Using Structured Queries” on page 69.

2.7 Returning Lexicon Values With search:values

The `search:values` Search API function returns values from lexicons. You can optionally constrain the values with a Structured Search, calculate aggregates based on the lexicon values, and find co-occurrences using the `<tuples>` options. For more general information about lexicons, see "Browsing With Lexicons" on page 146.

The following shows how to return co-occurrences (tuples) from the URI lexicon and an element, constraing on a query for `hello AND goodbye`, pulling data exclusively out of the range index:

```
xquery version "1.0-ml";
import module namespace search =
     "http://marklogic.com/appservices/search"
     at "/MarkLogic/appservices/search/search.xqy";

let $options :=
<options xmlns="http://marklogic.com/appservices/search">
  <tuples name="hello">
    <uri/>
    <range type="xs:string"
      collation="http://marklogic.com/collation/">
      <element ns="" name="hello"/>
    </range>
  </tuples>
</options>
return
$values:= search:values("hello", $options,
  search:parse("hello goodbye", (), "search:query"))
```

2.8 JSON Support in the Search API

The options node in the Search API allows you to specify JSON keys when you have loaded JSON documents into the database and the values you are searching for are associated with JSON keys. The following options node shows some sample `json-key` specifications:

```
<!-- Example of enhanced options structures supporting json key -->

<options xmlns="http://marklogic.com/appservices/search">
<!-- range constraint -->
    <constraint name="foo">
        <range type="xs:int">
            <json-key>foo</json-key>
        </range>
    </constraint>
```

```
<!-- range values -->
    <values name="foo-values">
        <range type="xs:int">
            <json-key>foo</json-key>
        </range>
    </values>

<!-- range tuples -->
    <tuples name="foo-tuples">
        <range type="xs:int">
            <json-key>foo</json-key>
        </range>
        <range type="xs:string">
            <json-key>bar</json-key>
        </range>
    </tuples>

<!-- default term with word -->
    <term apply="term">
        <default>
            <word>
                <json-key>bar</json-key>
            </word>
        </default>
        <empty apply="all-results"/>
    </term>

    <constraint name="bar">
        <word>
            <json-key>bar</json-key>
        </word>
    </constraint>

    <constraint name="baz">
        <value>
            <json-key>baz</json-key>
        </value>
    </constraint>

    <operator name="sort">
        <state name="score">
            <sort-order direction="ascending">
               <score/>
            </sort-order>
        </state>
         <state name="foo">
            <sort-order type="xs:int" direction="ascending">
               <json-key>asc</json-key>
            </sort-order>
        </state>
    </operator>
    <sort-order type="xs:int" direction="descending">
        <json-key>desc</json-key>
    </sort-order>
```

```
    <transform-results apply="snippet">
        <preferred-element>
            <element ns="f" name="foo"/>
            <json-key>chicken</json-key>
        </preferred-element>
    </transform-results>

    <extract-metadata>
        <qname elem-ns="n" elem-name="p"/>
        <json-key>name</json-key>
        <json-key>title</json-key>
        <json-key>affiliation</json-key>
    </extract-metadata>
    <debug>true</debug>
    <return-similar>false</return-similar>
</options>
```

2.9 More Search API Examples

This section shows the following examples that use the Search API:

- Buckets Example
- Computed Buckets Example
- Sort Order Example

2.9.1 Buckets Example

The following example from the Oscars sample application shows how to create a search that defines several decades as buckets, and those buckets are used to generate facets and as a constraint in the search grammar. Buckets are a type of range constraint, which are described in “Constraint Options” on page 25.

This example defines a constraint that uses a range index of type `xs:gYear` on a Wikipedia `nominee/@year` attribute.

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

let $options :=
<search:options>
  <search:constraint name="decade">
    <search:range type="xs:gYear" facet="true">
      <search:bucket ge="2000" name="2000s">2000s</search:bucket>
      <search:bucket lt="2000" ge="1990"
        name="1990s">1990s</search:bucket>
      <search:bucket lt="1990" ge="1980"
```

```
            name="1980s">1980s</search:bucket>
          <search:bucket lt="1980" ge="1970"
             name="1970s">1970s</search:bucket>
          <search:bucket lt="1970" ge="1960"
             name="1960s">1960s</search:bucket>
          <search:bucket lt="1960" ge="1950"
             name="1950s">1950s</search:bucket>
          <search:bucket lt="1950" ge="1940"
             name="1940s">1940s</search:bucket>
          <search:bucket lt="1940" ge="1930"
             name="1930s">1930s</search:bucket>
          <search:bucket lt="1930" ge="1920"
             name="1920s">1920s</search:bucket>
          <search:facet-option>limit=10</search:facet-option>
          <search:attribute ns="" name="year"/>
          <search:element ns="http://marklogic.com/wikipedia"
             name="nominee"/>
        </search:range>
      </search:constraint>
    </search:options>
    return
    search:search("james stewart decade:1940s", $options)
```

The following is a partial response from this query:

```
<search:response total="2" start="1" page-length="10" xmlns=""
   xmlns:search="http://marklogic.com/appservices/search">
  <search:result index="1" uri="/oscars/843224828394260114.xml"
    path="doc(&quot;/oscars/843224828394260114.xml&quot;)" score="200"
    confidence="0.670319" fitness="1">
    <search:snippet>
      <search:match path=
        "doc(&quot;/oscars/843224828394260114.xml&quot;)/*:nominee
        /*:name"><search:highlight>James</search:highlight>
        <search:highlight>Stewart</search:highlight></search:match>
.......
    </search:snippet>
    <search:snippet>.......</search:snippet>
.......
  </search:result>
  <search:facet name="decade">
    <search:facet-value name="1940s"
count="2">1940s</search:facet-value>
  </search:facet>
  <search:qtext>james stewart decade:1940s</search:qtext>
  <search:metrics>
    <search:query-resolution-time>
     PT0.152S</search:query-resolution-time>
    <search:facet-resolution-time>
     PT0.009S</search:facet-resolution-time>
    <search:snippet-resolution-time>
     PT0.073S</search:snippet-resolution-time>
    <search:total-time>PT0.234S</search:total-time>
  </search:metrics>
</search:response>
```

2.9.2 Computed Buckets Example

The `computed-bucket` range constraint operates over `xs:date` and `xs:dateTime` range indexes. The constraint specifies boundaries for the buckets that are computed at runtime based on computations made at the current time. The `anchor` attribute on the `computed-bucket` element has the following values:

`<computed-bucket anchor="value">`	Description
`anchor="now"`	The current time.
`anchor="start-of-day"`	The time of the start of the current day.
`anchor="start-of-month"`	The time of the start of the current month.
`anchor="start-of-year"`	The time of the start of the current year.

These values can also be used in `ge-anchor` and `le-anchor` attributes of the `computed-bucket` element.

The following search specifies a computed bucket and finds all of the documents that were updated today (this example assumes the maintain last-modified property is set on the database configuration):

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

search:search('modified:today',
<options xmlns="http://marklogic.com/appservices/search">
  <searchable-expression>xdmp:document-properties()
  </searchable-expression>
  <constraint name="modified">
    <range type="xs:dateTime">
      <element ns="http://marklogic.com/xdmp/property"
               name="last-modified"/>
      <computed-bucket name="today" ge="P0D" lt="P1D"
       anchor="start-of-day">Today</computed-bucket>
      <computed-bucket name="yesterday" ge="-P1D" lt="P0D"
       anchor="start-of-day">yesterday</computed-bucket>
      <computed-bucket name="30-days" ge="-P30D" lt="P0D"
       anchor="start-of-day">Last 30 days</computed-bucket>
      <computed-bucket name="60-days" ge="-P60D" lt="P0D"
       anchor="start-of-day">Last 60 Days</computed-bucket>
      <computed-bucket name="year" ge="-P1Y" lt="P1D"
       anchor="now">Last Year</computed-bucket>
    </range>
  </constraint>
</options>)
```

The `anchor` attributes have a value of `start-of-day`, so the duration values specified in the `ge` and `lt` attributes are applied at the start of the current day. Note that this is not the same as the "previous 24 hours," as the `start-of-day` value uses 12 o'clock midnight as the start of the day. The notion of time relative to days, months, and years, as opposed to relative to the exact current time, is the difference between relative buckets (`computed-bucket`) and absolute buckets (`bucket`). For an example that uses absolute buckets, see "Buckets Example" on page 64.

2.9.3 Sort Order Example

The following search specifies a custom sort order.

```
xquery version "1.0-ml";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

let $options :=
<search:options>
  <search:operator name="sort">
    <search:state name="relevance">
      <search:sort-order>
        <search:score/>
      </search:sort-order>
    </search:state>
    <search:state name="year">
      <search:sort-order direction="descending" type="xs:gYear"
            collation="">
        <search:attribute ns="" name="year"/>
        <search:element ns="http://marklogic.com/wikipedia"
          name="nominee"/>
      </search:sort-order>
      <search:sort-order>
        <search:score/>
      </search:sort-order>
    </search:state>
  </search:operator>
</search:options>
return
search:search("lange sort:year", $options)
```

This search specifies to sort by year. The options specification allows you to specify `year` or `relevance`, and without specifying, sorts by score (which is the same as `relevance` in this example).

3.0 Searching Using Structured Queries

This chapter describes how to perform searches using structured queries expressed in XML or JSON. The annotated `cts:query` that is generated by default from `search:parse` or `search:search` works well for cases where you do not need to perform extensive modification to the query. If you want to generate your own query, or if you want to parse your query using different rules from the Search API grammar rules, there is an alternate query style called structured query.

This chapter includes the following topics:

- Structured Query Overview
- Constructing a Structured Query
- Syntax Summary
- Examples of Structured Queries
- Syntax Reference

3.1 Structured Query Overview

A *structured query* is an Abstract Syntax Tree representation of a search expression, expressed in XML or JSON. For example, the following is a structured query in XML that is equivalent to the string query “cat AND dog”.

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:and-query>
    <search:term-query>
      <search:text>cat</search:text>
    </search:term-query>
    <search:term-query>
      <search:text>dog</search:text>
    </search:term-query>
  </search:and-query>
</search:query>
```

Any time you want to intercept a query and either augment or manipulate it in some way, consider using a structured query. The following use cases are good candidates for structured queries:

- Queries that do not work well in a string query. For example, constraints such as a geospatial constraint of a complex polygon are designed to be machine generated.
- Search strings that include complex sets of rules, or a set of rules that do not map well to the string query grammar.
- Combining a pre-parsed structured query with a user-generated or dynamically constructed string query.

- Converting a non-MarkLogic query representation such as an in-house query format into a form consumable by MarkLogic Server.
- String queries that require application-specific validation.

You can generate a structured query using the XQuery function `search:parse` or by writing your own code that returns a structured query. You can use structured queries to search contents and metadata with the following MarkLogic Server APIs:

- XQuery Search API. For details, see `search:search` and `search:resolve`.
- Java API. For details, see Searching in the *Java Application Developer's Guide*.
- REST API. For details, see Using and Configuring Query Features in the *REST Application Developer's Guide*.

3.2 Constructing a Structured Query

You can construct a structured query in multiple ways:

- Manually, using the syntax described in "Syntax Reference" on page 75.
- In XQuery, by calling the XQuery function `search:parse` and supplying `"search:query"` as the 3rd parameter to see the XML representation.
- In Java, using the class `com.marklogic.client.query.StructuredQueryBuilder`.

The XQuery Search API only accepts structured queries in XML. The REST and Java APIs accept XML and JSON representations.

For XML, you can use the `search:parse` technique with Query Console to explore how a string query or serialized `cts:query` maps to a structured query, and then modify it according to your needs. For example:

```
xquery version "1.0-ml";
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

return search:parse("cat AND dog", (), "search:query")
```

If you run the above query in Query Console and display the results as XML, you get the XML representation of "cat AND dog" when parsed with the default search options in effect:

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:and-query>
    <search:term-query>
      <search:text>cat</search:text>
    </search:term-query>
    <search:term-query>
      <search:text>dog</search:text>
```

```
    </search:term-query>
  </search:and-query>
</search:query>
```

3.3 Syntax Summary

This section gives a brief summary of structure of a structured query. For details, see “Syntax Reference” on page 75

A structured query is a `search:query` XML element with children representing `cts:query` composers, `cts:query` scoping constructors, and abstractions for Search API components such as constraints and operators. When using the XQuery Search API, you must use the XML representation. In REST and Java, you can choose between the XML or JSON representations.

Like `cts:query` constructors in XQuery, structured queries are composable to make a complex search query. A structured query expressed as XML must have a `<search:query>` wrapper node. For example:

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:and-query>...</search:and-query>
</search:query>
```

Similarly, the JSON representation of a structured query has a `query` wrapper object. Within this wrapper, the sub-queries are enclosed in a `queries` array. The queries wrapper is used wherever multiple sub-queries can occur. For example:

```
{
  "query": {
    "queries": [
      { "and-query": ... }
    ]
  }
}
```

You can compose complex queries consisting of and-queries, or-queries, and so on, and they can contain any number of term-queries with terms to search for. You can constrain queries to an element, attribute, JSON key, or field; use range-queries; and so on. For background on how `cts:query` expressions work in MarkLogic Server, see “Composing cts:query Expressions” on page 121.

The REST API and the Java API support XML and JSON representations of structured queries. The REST and Java APIs internally use the JSON conversion features in MarkLogic to convert between JSON and XML. For details on this conversion, see Working With JSON in the *Application Developer’s Guide*.

3.4 Examples of Structured Queries

This section includes the following examples of Structured Search, with an XML example as well as the corresponding JSON example for each:

- Example: Simple Structured Search
- Example: Structured Search With Constraint References as Text
- Example: Structured Search With Constraint References

For additional examples, see “Syntax Reference” on page 75.

3.4.1 Example: Simple Structured Search

The following is a structured query XML node equivalent to a string query for the phrase `"pretend this is a complex search"`:

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:term-query>
    <search:text>pretend this is a complex search</search:text>
  </search:term-query>
</search:query>
```

The following is the JSON representation of the same query (for use in the REST API or the Java API):

```
{
  "query": {
    "queries": [{
      "term-query": {
        "text": [ "pretend this is a complex search" ]
      }
    }]
}}
```

With XQuery, you can generate the XML structured query from a string query using `search:parse`, and perform a search with the structured query using `search:resolve`, as shown in the following code:

```
xquery version "1.0-ml";
import module namespace search =
     "http://marklogic.com/appservices/search"
     at "/MarkLogic/appservices/search/search.xqy";

let $complex-search :=
  search:parse('"pretend this is a complex search"',
      (), "search:query")
return
search:resolve($complex-search)

=> a search:response element
```

3.4.2 Example: Structured Search With Constraint References as Text

The following is a slightly more complicated Structured Search query. It has an and-query to combine terms, and has references to a constraint defined in a search options node, as it would be if parsed using the default query grammar (for example, `decade:1940s` represents the `decade` constraint with the value `1940s`).

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:and-query>
    <search:term-query>
      <search:text>hepburn</search:text>
    </search:term-query>
    <search:term-query>
      <search:text>decade:1940s</search:text>
    </search:term-query>
  </search:and-query>
</search:query>
```

The following is the corresponding JSON representation:

```
{
  "query": {
    "queries": [{
      "and-query": {
        "queries": [
          { "term-query": { "text": [ "hepburn" ] } },
          { "term-query": { "text": [ "decade:1940s" ] } }
        ]
      }
    }]
}}
```

3.4.3 Example: Structured Search With Constraint References

The following example demonstrates a query that includes an explicit reference to a constraint defined in query options.

Assume the query options include a decade bucketed constraint definition, and one of the buckets is named 1940s:

```
<!-- for complete options, see "Buckets Example" on page 64 -->
<search:constraint name="decade">
    <search:range type="xs:gYear" facet="true">
      ...more buckets...
      <search:bucket lt="1950" ge="1940"
         name="1940s">1940s</search:bucket>
      <search:bucket lt="1940" ge="1930"
         name="1930s">1930s</search:bucket>
      ...more buckets...
      <search:facet-option>limit=10</search:facet-option>
      <search:attribute ns="" name="year"/>
      <search:element ns="http://marklogic.com/wikipedia"
         name="nominee"/>
    </search:range>
  </search:constraint>
</search:options>
```

When evaluated with the above options, the string query `"hepburn AND decade:1940s"` expresses a range constraint (`decade:1940s`) that limits matches to those that meet the criteria for the `1940s` bucket of the `decade` constraint. The following is the equivalent structured query, expressed in XML:

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:and-query>
    <search:term-query>
      <search:text>hepburn</search:text>
    </search:term-query>
    <search:range-constraint-query>
      <search:constraint-name>decade</search:constraint-name>
      <search:value>1940s</search:value>
    </search:range-constraint-query>
  </search:and-query>
</search:query>
```

The following is the corresponding JSON representation:

```
{
  "query": {
    "queries": [{
      "and-query": {
        "queries": [
          { "term-query": { "text": [ "hepburn" ] } },
          {
            "range-constraint-query": {
              "value": [ "1940s" ],
              "constraint-name": "decade"
            }
          }
        ]
```

```
            }
        }]
    }}
```

3.5 Syntax Reference

This section provides detailed syntax information on structured queries. There is a subsection for each top level element in a structured query. Each section includes detailed syntax, an explanation of the child elements, and an example. Begin with the top level `query` wrapper.

- query
- term-query
- and-query
- or-query
- and-not-query
- not-query
- not-in-query
- near-query
- properties-query
- directory-query
- collection-query
- document-query
- document-fragment-query
- locks-query
- range-constraint-query
- value-constraint-query
- word-constraint-query
- collection-constraint-query
- element-constraint-query
- properties-constraint-query
- custom-constraint-query
- geospatial-constraint-query
- operator-state

3.5.1 query

A `query` is the top level wrapper around a structured query definition. It can contain one or more subquery children. See the subsections on each child for details.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <!-- word or phrase query -->`
`  <term-query />`
`  <!-- cts:query composers -->`
`  <and-query />`
`  <or-query />`
`  <and-not-query />`
`  <not-query />`
`  <not-in-query />`
`  <near-query> />`
`  <!-- cts:query scoping ctors -->`
`  <properties-query />`
`  <directory-query />`
`  <collection-query />`
`  <document-query />`
`  <document-fragment-query />`
`  <locks-query />`
`  <!-- Search API abstractions -->`
`  <range-constraint-query />`
`  <value-constraint-query />`
`  <word-constraint-query />`
`  <collection-constraint-query />`
`  <element-constraint-query />`
`  <properties-constraint-query />`
`  <custom-constraint-query />`
`  <geospatial-constraint-query />`
`  <operator-state />`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [`
	`  term-query,`
	
	`  and-query,`
	`  or-query,`
	`  and-not-query,`
	`  not-query,`
	`  not-in-query,`
	`  near-query,`
	
	`  properties-query,`
	`  directory-query,`
	`  collection-query,`
	`  document-query,`
	`  document-fragment-query,`
	`  locks-query,`
	
	`  range-constraint-query,`
	`  value-constraint-query,`
	`  word-constraint-query,`
	`  collection-constraint-query,`
	`  element-constraint-query,`
	`  properties-constraint-query,`
	`  custom-constraint-query,`
	`  geospatial-constraint-query,`
	`  operator-state`
	`  ]`
	`  }`
	`}`

3.5.2 term-query

A query that matches one or more search terms or phrases.

- Syntax Summary
- Component Description
- Examples

3.5.2.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<term-query>`
`  <text>`*`term`*`-or-phrase</text>`
`  <weight>`*`value`*`</weight>`
`</term-query>`
	`"term-query": {`
	`  "text": [ "`*`term-or-phrase`*`" ]`
	`  "weight": "`*`value`*`"`
	`}`

3.5.2.2 Component Description

Element Name or JSON Key	Req'd?	Description
`text`	Y	The term or phrase to search for. The query can contain multiple `text` children.
`weight`	N	A weight for this query. Default: 1.0. Higher weights move search results up in the relevance order. The weight should be less than or equal to the absolute value of 16 (between -16 and 16, inclusive). Weights greater than 16 have the same effect as a weight of 16. Weights less than the absolute value of 0.0625 (between -0.0625 and 0.0625) are rounded to 0, which means that they do not contribute to the score.

3.5.2.3 Examples

The following example searches for documents containing either of the terms “dog” or “cat”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <term-query>`
`  <text>dog</text>`
`  <text>cat</text>`
`  </term-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "term-query": {`
	`  "text": [ "dog", "cat" ]`
	`  }`
	`  }]`
	`}}`

3.5.3 and-query

Find the intersection of matches specified by one or more sub-queries. For details, see `cts:and-query`.

- Syntax Summary
- Component Description
- Examples

3.5.3.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<and-query>`
`  anyQueryType`
`  <ordered>bool</ordered>`
`</and-query>`
	`"and-query": {`
	`  "queries": [`
	`  anyQueryType,`
	`  "ordered": boolean`
	`  ]`
	`}`

3.5.3.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	N	One or more sub-queries.
`ordered`	N	Whether or not the sub-query matches must occur in the order of the sub-queries. For example, if the sub-queries are "cat" and "dog", an ordered query will only match fragments where both "cat" and "dog" occur, and where "cat" comes before "dog" in the fragment. Default: false.

3.5.3.3 Examples

The following example searches for documents containing both of the terms “dog” and “cat”, with “dog” occurring before “cat”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <and-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  <term-query>`
`  <text>cat</text>`
`  </term-query>`
`  <ordered>true</ordered>`
`  </and-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "and-query": {`
	`  "queries": [`
	`  {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  },`
	`  {`
	`  "term-query": {`
	`  "text": [ "cat" ]`
	`  }`
	`  },`
	`  { "ordered": "true" }`
	`  ]`
	`  }`
	`  }]`
	`}}`

3.5.4 or-query

Find the union of matches specified by one or more sub-queries. For details, see `cts:or-query`.

- Syntax Summary
- Component Description
- Examples

3.5.4.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<or-query>`
`  anyQueryType`
`</or-query>`
	`"or-query": {`
	`  "queries": [ anyQueryType ]`
	`}`

3.5.4.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	N	One or more sub-queries.

3.5.4.3 Examples

The following example matches documents containing either the phrase “dog bone” or the term “cat”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <or-query>`
`  <term-query>`
`  <text>dog bone</text>`
`  </term-query>`
`  <term-query>`
`  <text>cat</text>`
`  </term-query>`
`  </or-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "or-query": {`
	`  "queries": [`
	`  {`
	`  "term-query": {`
	`  "text": ["dog bone"]`
	`  }`
	`  },`
	`  {`
	`  "term-query": {`
	`  "text": [ "cat" ]`
	`  }`
	`  }`
	`  ]}`
	`  }]`
	`}}`

3.5.5 and-not-query

Find the set difference of the matches specified by two sub-queries. That is, return results that match the positive query, but which do not match the negative query. For details, see `cts:and-not-query`.

- Syntax Summary
- Component Description
- Examples

3.5.5.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<and-not-query>`
`  <positive-query>`
`  *anyQueryType*`
`  </positive-query>`
`  <negative-query>`
`  *anyQueryType*`
`  </negative-query>`
`</and-not-query>`
	`"and-not-query": {`
	`  "positive-query": { *anyQueryType* },`
	`  "negative-query" : { *anyQueryType* }`
	`}`

3.5.5.2 Component Description

Element Name or JSON Key	Req'd?	Description
`positive-query`	Y	A query specifying the results filtered in. All results will match this query.
`negative-query`	Y	A query specifying the results filtered out. None of the results will match this query.

3.5.5.3 Examples

The following example matches occurrences of dog, but only where “cat” does not occur in the same fragment.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <and-not-query>`
`  <positive-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </positive-query>`
`  <negative-query>`
`  <term-query>`
`  <text>cat</text>`
`  </term-query>`
`  </negative-query>`
`  </and-not-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "and-not-query": {`
	`  "positive-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }},`
	`  "negative-query": {`
	`  "term-query": {`
	`  "text": [ "cat" ]`
	`  }}`
	`  }`
	`  }]`
	`}}`

3.5.6 not-query

A query that filters out any results that match its sub-query. For details, see `cts:not-query`.

- Syntax Summary
- Component Description
- Examples

3.5.6.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<not-query>`
`  anyQueryType`
`</not-query>`
	`"not-query": {`
	`  anyQueryType`
	`}`

3.5.6.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	Y	A negative query, specifying the search results to filter out.

3.5.6.3 Examples

The following only matches documents that do not include the term "dog".

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<query>`
`  <not-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </not-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "not-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

3.5.7 not-in-query

A query that returns results matching a positive query only when those matches do not overlap positionally with matches to a negative query. For details, see `cts:not-in-query`.

- Syntax Summary
- Component Description
- Examples

3.5.7.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<not-in-query>`
`  <positive-query>`
`  *anyQueryType*`
`  </positive-query>`
`  <negative-query>`
`  *anyQueryType*`
`  </negative-query>`
`</not-in-query>`
	`"not-in-query": {`
	`  "positive-query": { *anyQueryType* },`
	`  "negative-query" : { *anyQueryType* }`
	`}`

3.5.7.2 Component Description

Element Name or JSON Key	Req'd?	Description
`positive-query`	Y	A positive query, specifying the search results to filter in.
`negative-query`	Y	A negative query, specifying the search results to filter out.

3.5.7.3 Examples

The example below matches fragments that contain at least one occurrence of “dog” outside of the phrase “man bites dog”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <not-in-query>`
`  <positive-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </positive-query>`
`  <negative-query>`
`  <term-query>`
`  <text>man bites dog</text>`
`  </term-query>`
`  </negative-query>`
`  </not-in-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "not-in-query": {`
	`  "positive-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }},`
	`  "negative-query": {`
	`  "term-query": {`
	`  "text": [`
	`  "man bites dog"`
	`  ]`
	`  }}`
	`  }`
	`  }]`
	`}}`

3.5.8 near-query

A query that returns results matching a positive query only when those matches do not overlap positionally with matches to a negative query. For details, see `cts:near-query`.

- Syntax Summary
- Component Description
- Examples

3.5.8.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<near-query>`
`  anyQueryType`
`  <distance>double</distance>`
`  <distance-weight>`
`  double`
`  </distance-weight>`
`  <ordered>boolean</ordered>`
`</near-query>`
	`"near-query": {`
	`  "queries": [`
	`  anyQueryType,`
	`  "distance": "number",`
	`  "distance-weight": "number",`
	`  "ordered" : boolean`
	`  ]`
	`}`

3.5.8.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	Y	One or more queries that must match within the specified proxmity to each other.
`distance`	N	A distance, in number of words, between any two matching queries. Default: 10
`distance-weight`	N	A weight attributed to the distance for this query. Higher weights add to the importance of distance (as opposed to term matches) when the relevance order is calculated. Default: 1.0.
`ordered`	N	Whether or not the sub-query matches must occur in the order of the sub-queries. For example, if the sub-queries are "cat" and "dog", an ordered query will only match fragments where both "cat" and "dog" occur within the required distance and "cat" comes before "dog" in the fragment. Default: false.

3.5.8.3 Examples

The following example matches occurrences of “dog” occuring within in two terms of “cat”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <near-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  <term-query>`
`  <text>cat</text>`
`  </term-query>`
`  <distance>2</distance>`
`  </near-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "near-query": {`
	`  "queries": [`
	`  {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  },`
	`  {`
	`  "term-query": {`
	`  "text": [ "cat" ]`
	`  }`
	`  },`
	`  { "distance": "2" }`
	`  ]`
	`  }`
	`  }]`
	`}}`

3.5.9 properties-query

A query that matches all documents where the sub-query matches against document properties. For details, see `cts:properties-query`.

- Syntax Summary
- Component Description
- Examples

3.5.9.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<properties-query>`
`  anyQueryType`
`</properties-query>`
	`"properties-query": {`
	`  anyQueryType`
	`}`

3.5.9.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	Y	A sub-query to run against document properties.

3.5.9.3 Examples

The following example matches all documents modified since 2012-12-31, assuming you define an element range index on the "last-modified" property and your query includes options defining the "modified" constraint.

Format	Query

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <constraint name="modified">
	  <range type="xs:string">
	  <element
	  ns="http://marklogic.com/xdmp/property"
	  name="last-modified"/>
	  <fragment-scope>properties</fragment-scope>
	  </range>
	  </constraint>
	</options>
	
	<query xmlns="http://marklogic.com/appservices/search">
	  <properties-query>
	  <range-constraint-query>
	  <constraint-name>modified</constraint-name>
	  <value>2012-12-31</value>
	  <range-operator>GT</range-operator>
	  </range-constraint-query>
	  </properties-query>
	</query>

JSON	{
	  "options": {
	  "constraint": [
	  {
	  "name": "modified",
	  "range": {
	  "type": "xs:string",
	  "element": {
	  "ns": "http://marklogic.com/xdmp/property",
	  "name": "last-modified"
	  },
	  "fragment-scope": "properties"
	  }]
	}}
	
	{
	  "query": {
	  "queries": [{
	  "properties-query": {
	  "range-constraint-query": {
	  "value": ["2012-12-31"],
	  "constraint-name": "modified",
	  "range-operator": "GT"
	  }
	  }
	  }]
	}}

3.5.10 directory-query

A query matching documents in the directories with the given URIs. For details, see `cts:directory-query`.

- Syntax Summary
- Component Description
- Examples

3.5.10.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<directory-query>`
`  <uri>`*`directory-uri`*`</uri>`
`  <infinite>`*`boolean`*`</infinite>`
`</directory-query>`
	`"directory-query": {`
	`  "uri": [ `*`directory-uris`*` ],`
	`  "infinite": `*`boolean`*
	`}`

3.5.10.2 Component Description

Element Name or JSON Key	Req'd?	Description
`uri`	Y	One or more directory URIs. A directory URI must end with a forward slash ("/").
`infinite`	N	Whether or not to recurse through all child directories. Default: true.

3.5.10.3 Examples

The following example matches documents in the database directories `/documents/` or `/images/`, but does not look for matches in any sub-directories.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <directory-query>`
`  <uri>/documents/</uri>`
`  <uri>/images/</uri>`
`  <infinite>false</infinite>`
`  </directory-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "directory-query": {`
	`  "uri": [`
	`  "/documents/",`
	`  "/images/"`
	`  ],`
	`  "infinite": false`
	`  }`
	`  }]`
	`}}`

3.5.11 collection-query

A query matching documents in the collections with the given URIs.. For details, see `cts:collection-query`.

- Syntax Summary
- Component Description
- Examples

3.5.11.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<collection-query>`
`  <uri>`*`collection-uri`*`</uri>`
`</collection-query>`
	`"collection-query": {`
	`  uri": [ `*`collection-uris`*` ]`
	`}`

3.5.11.2 Component Description

Element Name or JSON Key	Req'd?	Description
`uri`	Y	One or more collection URIs.

3.5.11.3 Examples

The following example matches documents in the `reports` collection or the `analysis` collection.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <collection-query>`
`  <uri>reports</uri>`
`  <uri>analysis</uri>`
`  </collection-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "collection-query": {`
	`  "uri": ["reports","analysis"]`
	`  }`
	`  }]`
	`}}`

3.5.12 document-query

A query matching documents with the given URIs.. For details, see `cts:document-query`.

- Syntax Summary
- Component Description
- Examples

3.5.12.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<document-query>`
`  <uri>`*`document-uri`*`</uri>`
`</document-query>`
	`"document-query": {`
	`  "uri": [ `*`document-uris`*` ]`
	`}`

3.5.12.2 Component Description

Element Name or JSON Key	Req'd?	Description
`uri`	Y	One or more document URIs.

3.5.12.3 Examples

The following example matches either the document with URI `/documents/reports.xml` or the document with URI `/documents/analysis.xml`.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <document-query>`
`  <uri>/documents/reports.xml</uri>`
`  <uri>/documents/analysis.xml</uri>`
`  </document-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "document-query": {`
	`  "uri": [`
	`  "/documents/reports.xml",`
	`  "/documents/analysis.xml"`
	`  ]`
	`  }`
	`  }]`
	`}}`

3.5.13 document-fragment-query

A query that matches all documents where a sub-query matches any document fragment. For details, see `cts:document-fragment-query`.

- Syntax Summary
- Component Description
- Examples

3.5.13.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<document-fragment-query>`
`  anyQueryType`
`</document-fragment-query>`
	`"document-fragment-query": {`
	`  anyQueryType`
	`}`

3.5.13.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	Y	The query to be matched against any document fragment.

3.5.13.3 Examples

The following example matches any documents that include fragments that contain the term `dog`.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <document-fragment-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </document-fragment-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "document-fragment-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

3.5.14 locks-query

A query that matches all documents where a sub-query matches a `document-locks` fragment. For details, see `cts:locks-query`.

- Syntax Summary
- Component Description
- Examples

3.5.14.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<locks-query>`
`  *anyQueryType*`
`</locks-query>`
	`"locks-query": {`
	`  *anyQueryType*`
	`}`

3.5.14.2 Component Description

Element Name or JSON Key	Req'd?	Description
anyQueryType	Y	The query to be matched against any document fragment.

3.5.14.3 Examples

The following example matches documents with document locks fragments that include the term `write` in `lock:lock-type` element. This example assumes the existence of an element range index on `lock:lock-type` and query options that include a constraint on that element with the name `lock-type`.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <locks-query>`
`  <element-query>`
`  <constraint-name>`
`  lock-type`
`  </constraint-name>`
`  <term-query>`
`  <text>write</text>`
`  </term-query>`
`  </element-query>`
`  </locks-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "locks-query": {`
	`  "element-query": {`
	`  "constraint-name": "lock-type",`
	`  "term-query": {`
	`  "text": [ "write" ]`
	`  }`
	`  }}`
	`  }]`
	`}}`

3.5.15 range-constraint-query

A query that applies a pre-defined range constraint and compares the results to the specified value. For details, see "Constraint Options" on page 25 and the XQuery functions `cts:element-range-query`, `cts-element-attribute-range-query`, `cts:field-range-query`, and `cts:path-range-query`.

A `range-constraint-query` is equivalent to string query expressions of the form `constraint:value` or `constraint LE value`. The named constraint must be backed by a range index.

- Syntax Summary
- Component Description
- Examples

3.5.15.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<range-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <value>`*value-to-match*`</value>`
`  <range-operator>`*operator*`</range-operator>`
`</range-constraint-query>`
	`"range-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "value": [ `*value-to-match*` ],`
	`  "range-operator": "`*operator*`"`
	`}`

3.5.15.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a constraint defined in the global or query-specific query options.
`value`	Y	The value against which to match elements, element attributes, JSON keys, or fields that match the constraint identified by `constraint-name`. This element can occur 0 or more times.
`range-operator`	N	One of `LT`, `LE`, `GT`, `GE`, `EQ`, `NE`. Default: `EQ`. The match relationship that must be satisfied between `constraint-name matches` and `value`.

3.5.15.3 Examples

The following example matches documents containing a `<body-color/>` element with a value of `black`.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="color">`
	`  <range type="xs:string">`
	`  <element ns="" name="body-color"/>`
	`  </range>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <element-query>`
	`  <range-constraint-query>`
	`  <constraint-name>color</constraint-name>`
	`  <value>black</value>`
	`  </range-constraint-query>`
	`  </element-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [`
	`  {`
	`  "name": "color",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "element": {`
	`  "ns": "",`
	`  "name": "body-color"`
	`  }`
	`  }}]}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "element-query": {`
	`  "range-constraint-query": {`
	`  "value": [ "black" ],`
	`  "constraint-name": "color"`
	`  }}`
	`  }]`
	`}}`

3.5.16 value-constraint-query

A query that matches fragments where the value of the text content of an XML element, XML attribute, JSON key-value pair, or field exactly matches the `text` phrase in the query. The match semantics depend on the `text` value, the database configuration, and the options in effect. The element, attribute, key, or field is identified by a value constraint defined in query options. This is similar to a string query term of the form `constraint:value`. For details, see `cts:element-value-query`, `cts:element-attribute-value-query`, `cts:field-value-query`.

- Syntax Summary
- Component Description
- Examples

3.5.16.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<value-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <text>`*name*`</text>`
`  <weight>`*value*`</weight>`
`</value-constraint-query>`
	`"value-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "text": [ `*name*` ],`
	`  "weight": `*number*
	`}`

3.5.16.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a values constraint defined in the global or query-specific query options.
`text`	N	A value to match. Multiple values can be specified. If there is no `text` component, all values matching the constraint are returned.
`weight`	N	A weight for this query. Default: 1.0. Higher weights move search results up in the relevance order. The weight should be less than or equal to the absolute value of 16 (between -16 and 16, inclusive). Weights greater than 16 have the same effect as a weight of 16. Weights less than the absolute value of 0.0625 (between -0.0625 and 0.0625) are rounded to 0, which means that they do not contribute to the score.

3.5.16.3 Examples

The following example matches documents where the field defined with the name "`myFieldName`" has the value "`Jane Doe`". The example assumes the field "`myFieldName`" is defined in the database configuration and that field searches are enabled for the database.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="full-name">`
	`  <value>`
	`  <field name="myFieldName"/>`
	`  </value>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <value-constraint-query>`
	`  <constraint-name>full-name</constraint-name>`
	`  <text>Jane Doe</text>`
	`  </value-constraint-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [`
	`  {`
	`  "name": "full-name",`
	`  "value": {`
	`  "field": {`
	`  "name": "myFieldName"`
	`  }}}]}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "value-constraint-query": {`
	`  "text": [ "Jane Doe" ],`
	`  "constraint-name": "full-name"`
	`  }`
	`  }]`
	`}}`

3.5.17 word-constraint-query

A query that matches fragments containing the specified terms or phrases in the XML element or attribute, JSON key, or field identified by a specified constraint. This is similar to a string query of the form `constraint:value`, where the constraint is a word constraint. For details, see `cts:word-query`.

- Syntax Summary
- Component Description
- Examples

3.5.17.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<word-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <text>`*name*`</text>`
`  <weight>`*value*`</weight>`
`</word-constraint-query>`
	`"word-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "text": [ `*name*` ],`
	`  "weight": `*number*
	`}`

3.5.17.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a word constraint defined in the global or query-specific query options.
`text`	N	Terms or phrases that must occur in documents matching the constraint defined by `constraint-name`. Multiple values can be specified; if any one matches, the document matches the query.
`weight`	N	A weight for this query. Default: 1.0. Higher weights move search results up in the relevance order. The weight should be less than or equal to the absolute value of 16 (between -16 and 16, inclusive). Weights greater than 16 have the same effect as a weight of 16. Weights less than the absolute value of 0.0625 (between -0.0625 and 0.0625) are rounded to 0, which means that they do not contribute to the score.

3.5.17.3 Examples

The following example matches documents containing a `<body-color/>` element that contains the word `black`.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="color">`
	`  <word>`
	`  <element name="body-color"/>`
	`  </word>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <word-constraint-query>`
	`  <constraint-name>color</constraint-name>`
	`  <text>black</text>`
	`  </word-constraint-query>`
	`</query>`

JSON	`{`
	`  "query": {`
	`  "queries": [{`
	`  "word-constraint-query": {`
	`  "text": [ "black" ],`
	`  "constraint-name": "color"`
	`  }`
	`  }]`
	`}}`

3.5.18 collection-constraint-query

A query that applies a pre-defined constraint and compares the results to the specified value. For details, see `cts:collection-query`.

- Syntax Summary
- Component Description
- Examples

3.5.18.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<collection-constraint-query>`
`  <constraint-name>`*`name`*`</constraint-name>`
`  <uri>`*`collection-uri`*`</value>`
`</collection-constraint-query>`
	`"collection-constraint-query": {`
	`  "constraint-name": "`*`name`*`",`
	`  "uri": [ `*`collection-uri`*` ]`
	`}`

3.5.18.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a collection constraint defined in the global or query-specific query options.
`uri`	N	One or more collection URIs to match against.

3.5.18.3 Examples

The following example matches documents in the collection `reports` or the collection `analysis`.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="biz">`
	`  <collection prefix="my-coll-prefix"/>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <collection-constraint-query>`
	`  <constraint-name>biz</constraint-name>`
	`  <uri>reports</uri>`
	`  <uri>analysis</uri>`
	`  </collection-constraint-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [{`
	`  "name": "biz",`
	`  "collection": {`
	`  "prefix": "my-coll-prefix"`
	`  }`
	`  }]`
	`}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "collection-constraint-query": {`
	`  "uri": [ "reports", "analysis" ],`
	`  "constraint-name": "biz"`
	`  }`
	`  }]`
	`}}`

3.5.19 element-constraint-query

A query that matches elements meeting a specified element constraint, with sub-elements and/or attribute that match a specified sub-query(s). The matching element and all of its descendants are considered by the sub-queries. For details, see `cts:element-query`.

- Syntax Summary
- Component Description

- Examples

3.5.19.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<element-constraint-query>`
`  <constraint-name>`*`name`*`</constraint-name>`
`  `*`anyQueryType`*
`</element-constraint-query>`
	`"element-constraint-query": {`
	`  "constraint-name": "`*`name`*`",`
	`  `*`anyQueryType`*
	`}`

3.5.19.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of an constraint defined in the global or query-specific query options.
`anyQueryType`	Y	A query to run against elements matching the constraint identified by `constraint-name`.

3.5.19.3 Examples

The following example matches occurrences of a `<body-color/>` element that contains the term `black`.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="body-color">`
	`  <element name="color" ns="" />`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <element-constraint-query>`
	`  <constraint-name>body-color</constraint-name>`
	`  <term-query>`
	`  <text>black</text>`
	`  </term-query>`
	`  </element-constraint-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [{`
	`  "name": "body-color",`
	`  "element": {`
	`  "name": "color",`
	`  "ns": ""`
	`  }`
	`  }]`
	`}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "element-constraint-query": {`
	`  "constraint-name": "body-color",`
	`  "term-query": {`
	`  "text": [ "black" ]`
	
	`  }`
	`  }]`
	`}}`

3.5.20 properties-constraint-query

A query that matches documents with properties that match the specified property constraint, where the matching properties also match the specified query. For details, see `cts:properties-query`.

- Syntax Summary
- Component Description
- Examples

3.5.20.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<properties-constraint-query>`
`  <constraint-name>`*`name`*`</constraint-name>`
`  `*`anyQueryType`*
`</properties-constraint-query>`
	`"properties-constraint-query": {`
	`  "constraint-name": "`*`name`*`",`
	`  `*`anyQueryType`*
	`}`

3.5.20.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a properties constraint defined in the global or query-specific query options.
`anyQueryType`	Y	A query to run against properties matching the constraint identified by `constraint-name`.

3.5.20.3 Examples

The following example matches documents that have properties fragments containing the term `dog`.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="prop-only">`
	`  <properties />`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <properties-constraint-query>`
	`  <constraint-name>prop-only</constraint-name>`
	`  <term-query>`
	`  <text>dog</text>`
	`  </term-query>`
	`  </properties-constraint-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [{`
	`  "name": "prop-only",`
	`  "properties": null`
	`  }]`
	`}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "properties-constraint-query": {`
	`  "constraint-name": "prop-only",`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

3.5.21 custom-constraint-query

A query constructed by a custom XQuery extension function, using the supplied criteria. For details, see “Creating a Custom Constraint” on page 45.

- Syntax Summary

- Component Description
- Examples

3.5.21.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<custom-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <text>term</text>`
`</custom-constraint-query>`
	`"custom-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "text": [ `*term*` ]`
	`}`

3.5.21.2 Component Description

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a custom constraint defined in the global or query-specific query options.
`text`	N	A query to run against fragments matching the constraint identified by `constraint-name`.

3.5.21.3 Examples

The following example is equivalent to the string query "`part:book`" where `part` is the name of a custom constraint defined in the query options.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="part">`
	`  <custom facet="false">`
	`  <parse apply="part" ns="my-namespace" at="/my-module.xqy"/>`
	`  </custom>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <custom-constraint-query>`
	`  <constraint-name>part</constraint-name>`
	`  <text>book</text>`
	`  </custom-constraint-query>`
	`</query>`

JSON	`{`
	`  "options": {`
	`  "constraint": [{`
	`  "name": "part",`
	`  "custom": {`
	`  "facet": false,`
	`  "parse": {`
	`  "apply": "part",`
	`  "ns": "my-namespace",`
	`  "at": "/my-module.xqy"`
	`  }`
	`  }`
	`  }]`
	`}}`
	
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "custom-constraint-query": {`
	`  "text": [ "book" ],`
	`  "constraint-name": "part"`
	`  }`
	`  }]`
	`}}`

3.5.22 geospatial-constraint-query

A query that returns documents that match the specified geospatial constraint and the matching fragments also match the geospatial queries. For details, see “Geospatial Search Applications” on page 175.

- Syntax Summary
- Component Description
- Examples

3.5.22.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geospatial-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <point>`
`  <latitude>`*float*`</latitude>`
`  <longitude>`*float*`</longitude>`
`  </point>`
`  <box>`
`  <south>`*float*`</south>`
`  <west>`*float*`</west>`
`  <north>`*float*`</north>`
`  <east>`*float*`</east>`
`  </box>`
`  <circle>`
`  <radius>`*float*`</radius>`
`  <point/>`
`  </circle>`
`  <polygon>`
`  <point/>`
`  </polygon>`
`  <text>term</text>`
`</geospatial-constraint-query>`
	`"geospatial-constraint-query": {`
	`  "constraint-name": "name",`
	`  "point": [`
	`  {`
	`  "latitude": `*number*`,`
	`  "longitude": `*number*
	`  }`
	`  ],`
	`  "box": [`
	`  {`
	`  "south": `*number*`,`
	`  "west": `*number*`,`
	`  "north": `*number*`,`
	`  "east": `*number*
	`  }`
	`  ],`
	`  "circle": [`
	`  {`
	`  "radius": `*number*`,`
	point
	`  }`
	`  ],`
	`  "polygon": [ `*point*` ]`
	`}`

3.5.22.2 Component Description

A geospatial constraint query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects.

Element Name or JSON Key	Req'd?	Description
`constraint-name`	Y	The name of a custom constraint defined in the global or query-specific query options.
`point`	N	Zero or more geographic points, each defined by a `latitude` and a `longitude`. The query can contain 0 or more points.
`box`	N	Zero or more rectangular regions, each defined by 4 points: `north`, `south`, `east`, and `west`.
`circle`	N	Zero or more circles, each defined by `radius` and a center `point`.
`polygon`	N	Zero or more polygons, each series of `point`'s.

3.5.22.3 Examples

The following example matches points contained in either of two polygons. The `my-geo-elem-pair` constraint in the query options defines how to construct the points.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-elem-pair">`
	`  <geo-elem-pair>`
	`  <parent ns="ns1" name="elem2"/>`
	`  <lat ns="ns2" name="attr2"/>`
	`  <lon ns="ns3" name="attr3"/>`
	`  </geo-elem-pair>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <geospatial-constraint-query>`
	`  <constraint-name>name</constraint-name>`
	`  <polygon>`
	`  <point> <latitude>1</latitude> <longitude>2</longitude> </point>`
	`  <point> <latitude>3</latitude> <longitude>4</longitude> </point>`
	`  <point> <latitude>5</latitude> <longitude>6</longitude> </point>`
	`  <point> <latitude>7</latitude> <longitude>8</longitude> </point>`
	`  </polygon>`
	`  <polygon>`
	`  <point> <latitude>2</latitude> <longitude>6</longitude> </point>`
	`  <point> <latitude>3</latitude> <longitude>7</longitude> </point>`
	`  <point> <latitude>4</latitude> <longitude>8</longitude> </point>`
	`  <point> <latitude>5</latitude> <longitude>9</longitude> </point>`
	`  </polygon>`
	`  </geospatial-constraint-query>`
	`</query>`

Format	Query

JSON	<pre>{
	  "options": {
	    "constraint": [{
	      "name": "my-geo-elem-pair",
	      "geo-elem-pair": {
	        "parent": { "ns": "ns1", "name": "elem2" },
	        "lat": { "ns": "ns2", "name": "attr2" },
	        "lon": { "ns": "ns3", "name": "attr3" }
	      }
	    }]
	}}
	
	{
	  "query": {
	    "queries": [{
	      "geospatial-constraint-query": {
	        "constraint-name": "name",
	        "polygon": [
	          {
	            "point": [
	              { "latitude": 1, "longitude": 2 },
	              { "latitude": 3, "longitude": 4 },
	              { "latitude": 5, "longitude": 6 },
	              { "latitude": 7, "longitude": 8 }
	            ]
	          },
	          {
	            "point": [
	              { "latitude": 2, "longitude": 6 },
	              { "latitude": 3, "longitude": 7 },
	              { "latitude": 4, "longitude": 8 },
	              { "latitude": 5, "longitude": 9 }
	            ]
	          }
	        ]
	      }
	    }]
	}}</pre>

3.5.23 operator-state

This component of a structured query sets the state of a custom runtime configuration operator defined by your string query grammar. For details, see “Operator Options” on page 35.

- Syntax Summary
- Component Description
- Examples

3.5.23.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<operator-state>`
`  <operator-name>`*name*`</operator-name>`
`  <state-name>`*state*`</state-name>`
`</operator-state>`
	`"operator-state": {`
	`  "operator-name": "`*name*`",`
	`  "state-name": "`*state*`"`
	`}`

3.5.23.2 Component Description

Element Name or JSON Key	Req'd?	Description
`operator-state`	Y	The name of a custom runtime configuration operator defined by the `<operator/>` query option.
`state-name`	Y	The name of a state recognized by this operator.

3.5.23.3 Examples

The following example illustrates use of a custom `sort` operator defined in the query options. The example structured query is equivalent to the string query "`sort:date`". For details, see "Operator Options" on page 35.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <operator name="sort">`
	`  <state name="relevance">`
	`  <sort-order>`
	`  <score/>`
	`  </sort-order>`
	`  </state>`
	`  <state name="date">`
	`  <sort-order direction="descending" type="xs:dateTime">`
	`  <element ns="my-ns" name="date"/>`
	`  </sort-order>`
	`  <sort-order>`
	`  <score/>`
	`  </sort-order>`
	`  </state>`
	` </operator>`
	`</options>`
	
	`<query xmlns:search="http://marklogic.com/appservices/search">`
	`  <operator-state>`
	`  <operator-name>sort</operator-name>`
	`  <state-name>date</state-name>`
	`  </operator-state>`
	`</query>`

Format	Query

JSON	<pre>{
	  "options": {
	    "operator": [{
	      "name": "sort",
	      "state": [
	        {
	          "name": "relevance",
	          "sort-order": [{ "score": null }]
	        },
	        {
	          "name": "date",
	          "sort-order": [
	            {
	              "direction": "descending",
	              "type": "xs:dateTime",
	              "element": { "ns": "my-ns", "name": "date" }
	            },
	            { "score": null }
	          ]
	      }]
	    }]
	}}
	
	{
	  "query": {
	    "queries": [{
	      "operator-state": {
	        "operator-name": "sort",
	        "state-name": "date"
	      }
	    }]
	}}</pre>

4.0 Composing cts:query Expressions

Searches in MarkLogic Server use expressions that have a `cts:query` type. This chapter describes how to create various types of `cts:query` expressions and how you can register some complex expressions to improve performance of future queries that use the registered `cts:query` expressions.

MarkLogic Server includes many Built-In XQuery functions to compose `cts:query` expressions. The signatures and descriptions of the various APIs are described in the *MarkLogic XQuery and XSLT Function Reference*.

This chapter includes the following sections:

- Understanding cts:query
- Combining multiple cts:query Expressions
- Joining Documents and Properties with cts:properties-query or cts:document-fragment-query
- Registering cts:query Expressions to Speed Search Performance
- Adding Relevance Information to cts:query Expressions:
- XML Serializations of cts:query Constructors
- Example: Creating a cts:query Parser

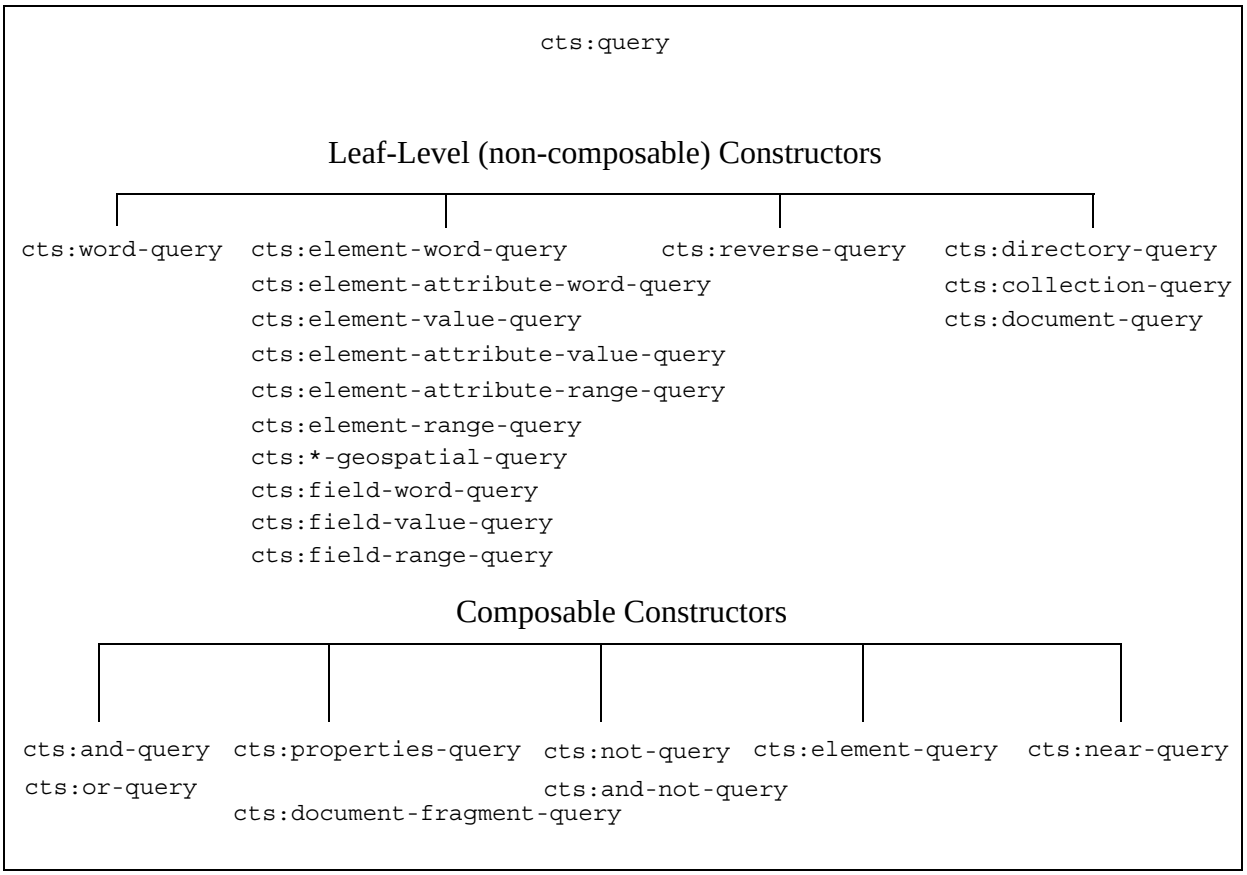
4.1 Understanding cts:query

The second parameter for `cts:search` takes a parameter of `cts:query` type. The contents of the `cts:query` expression determines the conditions in which a search will return a document or node. This section describes `cts:query` and includes the following parts:

- cts:query Hierarchy
- Use to Narrow the Search
- Understanding cts:element-query
- Understanding cts:element-word-query
- Understanding Field Word and Value Query Constructors
- Understanding the Range Query Constructors
- Understanding the Reverse Query Constructor
- Understanding the Geospatial Query Constructors
- Specifying the Language in a cts:query

4.1.1 cts:query Hierarchy

The `cts:query` type forms a hierarchy, allowing you to construct complex `cts:query` expressions by combining multiple expressions together. The hierarchy includes composable and non-composable `cts:query` constructors. A *composable* constructor is one that is used to combine multiple `cts:query` constructors together. A *leaf-level* constructor is one that cannot be used to combine with other `cts:query` constructors (although it can be combined using a composable constructor). The following diagram shows the leaf-level `cts:query` constructors, which are not composable, and the composable `cts:query` constructors, which you can use to combine both leaf-level and other composable `cts:query` constructors. For more details on combining `cts:query` constructors, see the remainder of this chapter.



4.1.2 Use to Narrow the Search

The core search `cts:query` API is `cts:word-query`. The `cts:word-query` function returns true for words or phrases that match its `$text` parameter, thus narrowing the search to fragments containing terms that match the query. If needed, you can use other `cts:query` APIs to combine a `cts:word-query` expression into a more complex expression. Similarly, you can use the other leaf-level cts:query constructors to narrow the results of a search.

4.1.3 Understanding cts:element-query

The `cts:element-query` function searches through a specified element and all of its children. It is used to narrow the field of search to the specified element hierarchy, exploiting the XML structure in the data. Also, it is composable with other `cts:element-query` functions, allowing you to specify complex hierarchical conditions in the `cts:query` expressions.

For example, the following search against a Shakespeare database returns the title of any play that has SCENE elements that have SPEECH elements containing both the words “room” and “castle”:

```
for $x in cts:search(fn:doc(),
   cts:element-query(xs:QName("SCENE"),
       cts:element-query(xs:QName("SPEECH"),
           cts:and-query(("room", "castle")) ) ) )
return
($x//TITLE)[1]
```

This query returns the first `TITLE` element of the play. The `TITLE` element is used for both play and scene titles, and the first one in a play is the title of the play.

When you use `cts:element-query` and you have both the `word positions` and `element word positions` indexes enabled in the Admin Interface, it will speed the performance of many queries that have multiple term queries (for example, `"the long sly fox"`) by eliminating some false positive results.

4.1.4 Understanding cts:element-word-query

While `cts:element-query` searches through an element and all of its children, `cts:element-word-query` searches only the immediate text node children of the specified element. For example, consider the following XML structure:

```
<root>
  <a>hello
    <b>goodbye</b>
  <a>
</root>
```

The following query returns `false`, because `"goodbye"` is not an immediate text node of the element named `a`:

```
cts:element-word-query(xs:QName("a"), "goodbye")
```

4.1.5 Understanding Field Word and Value Query Constructors

The `cts:field-word-query` and `cts:field-value-query` constructors search in fields for either words or values. A field value is defined as all of the text within a field, with a single space between text that comes from different elements. For example, consider the following XML structure:

```
<name>
  <first>Raymond</first>
  <middle>Clevie</middle>
  <last>Carver</last>
</name>
```

If you want to normalize names in the form `firstname lastname`, then you can create a field on this structure. The field might include the element `name` and exclude the element `middle`. The value of this instance of the field would then be `Raymond Carver`, with a space between the text from the two different element values from `first` and `last`. If your document contained other `name` elements with the same structure, their values would be derived similarly. If the field is named `my-field`, then a `cts:field-value-query("my-field", "Raymond Carver")` returns true for documents containing this XML. Similarly, a `cts:field-word-query("my-field", "Raymond Carver")` returns true.

For more information about fields, see Fields Database Settings in the *Administrator's Guide*. For information on lexicons on fields, see "Field Value Lexicons" on page 150.

4.1.6 Understanding the Range Query Constructors

The `cts:element-range-query`, `cts:element-atribute-range-query`, `cts:path-range-query`, and `cts:field-range-query` constructors allow you to specify constraints on a value in a `cts:query` expression. The range query constructors require a range index on the specified element or attribute. For details on range queries, see "Using Range Queries in cts:query Expressions" on page 159.

4.1.7 Understanding the Reverse Query Constructor

The `cts:reverse-query` constructor allows you to match queries stored in a database to nodes that would match those queries. Reverse queries are used as the basis for alert applications. For details, see "Creating Alerting Applications" on page 196.

4.1.8 Understanding the Geospatial Query Constructors

The geospatial query constructors are used to constrain `cts:query` expressions on geospatial data. Geospatial searches are used with documents that have been marked up with latitude and longitude data, and can be used to answer queries like "show me all of the documents that mention places within 100 miles of New York City." For details on gesospatial searches, see "Geospatial Search Applications" on page 175.

4.1.9 Specifying the Language in a cts:query

All leaf-level `cts:query` constructors are language-aware; you can either explicitly specify a language value as an option, or it will default to the database default language. The language option specifies the language in which the query is tokenized and, for stemmed searches, the language of the content to be searched.

To specify the language option in a `cts:query`, use the `lang=`*`language_code`*, where *language_code* is the two or three character ISO 639-1 or ISO 639-2 language code (http://www.loc.gov/standards/iso639-2/php/code_list.php). For example, the following query:

```
let $x :=
<root>
 <el xml:lang="en">hello</el>
 <el xml:lang="fr">hello</el>
</root>
return
$x//el[cts:contains(.,
         cts:word-query("hello", ("stemmed", "lang=fr")))]
```

returns only the French-language node:

```
<el xml:lang="fr">hello</el>
```

Depending on the language of the `cts:query` and on the language of the content, a string will tokenize differently, which will affect the search results. For details on how languages and the `xml:lang` attribute affect tokenization and searches, see “Language Support in MarkLogic Server” on page 292.

4.2 Combining multiple cts:query Expressions

Because `cts:query` expressions are composable, you can combine multiple expressions to form a single expression. There is no limit to how complex you can make a `cts:query` expressions. Any API that has a return type of `cts:*` (for example, `cts:query`, `cts:and-query`, and so on) can be composed with another `cts:query` expression to form another expression. This section has the following parts:

- Using cts:and-query and cts:or-query
- Proximity Queries using cts:near-query
- Using Bounded cts:query Expressions
- Matching Nothing and Matching Everything

4.2.1 Using cts:and-query and cts:or-query

You can construct arbitrarily complex boolean logic by combining `cts:and-query` and `cts:or-query` constructors in a single `cts:query` expression.

For example, the following search with a relatively simple nested `cts:query` expression will return all fragments that contain either the word `alfa` or the word `maserati`, and also contain either the word `saab` or the word `volvo`.

```
cts:search(fn:doc(),
  cts:and-query( ( cts:or-query(("alfa", "maserati")),
                   cts:or-query(("saab", "volvo") )
  ) )
)
```

Additionally, you can use `cts:and-not-query` and `cts:not-query` to add negation to your boolean logic.

4.2.2 Proximity Queries using cts:near-query

You can add tests for proximity to a `cts:query` expression using `cts:near-query`. Proximity queries use the `word positions` index in the database and, if you are using `cts:element-query`, the `element word positions` index. Proximity queries will still work without these indexes, but the indexes will speed performance of queries that use `cts:near-query`.

Proximity queries return `true` if the query matches occur within the specified distance from each other. For more details, see the *MarkLogic XQuery and XSLT Function Reference* for `cts:near-query`.

4.2.3 Using Bounded cts:query Expressions

The following `cts:query` constructors allow you to bound a `cts:query` expression to one or more documents, a directory, or one or more collections.

- `cts:document-query`
- `cts:directory-query`
- `cts:collection-query`

These bounding constructors allow you to narrow a set of search results as part of the second parameter to `cts:search`. Bounding the query in the `cts:query` expression is much more efficient than filtering results in a `where` clause, and is often more convenient than modifying the XPath in the first `cts:search` parameter. To combine a bounded `cts:query` constructor with another constructor, use a `cts:and-query` or a `cts:or-query` constructor.

For example, the following constrains a search to a particular directory, returning the URI of the document(s) that match the `cts:query`.

```
for $x in cts:search(fn:doc(),
   cts:and-query((
     cts:directory-query("/shakespeare/plays/", "infinity"),
         "all's well that"))
)
return xdmp:node-uri($x)
```

This query returns the URI of all documents under the specified directory that satisfy the query `"all's well that"`.

> **Note:** In this query, the query `"all's well that"` is equivalent to a `cts:word-query("all's well that")`.

4.2.4 Matching Nothing and Matching Everything

An empty `cts:word-query` will always match no fragments, and an empty `cts:and-query` will always match all fragments. Therefore the following are true:

```
cts:search(fn:doc(), cts:word-query("") )
=> returns the empty sequence

cts:search(fn:doc(), "" )
=> returns the empty sequence

cts:search(fn:doc(), cts:and-query( () ) )
=> returns every fragment in the database
```

One use for an empty `cts:word-query` is when you have a search box that an end user enters terms to search for. If the user enters nothing and hits the submit button, then the corresponding cts:search will return no hits.

An empty `cts:and-query` that matches everything is sometimes useful when you need a `cts:query` to match everything.

4.3 Joining Documents and Properties with cts:properties-query or cts:document-fragment-query

You can use a `cts:properties-query` to match content in properties document. If you are searching over a document, then a `cts:properties-query` will search in the properties document at the URI of the document. The `cts:properties-query` joins the properties document with its corresponding document. The `cts:properties-query` takes a `cts:query` as a parameter, and that query is used to match against the properties document. A `cts:properties-query` is composable, so you can combine it with other `cts:query` constructors to create arbitrarily complex queries.

Using a `cts:properties-query` in a `cts:search`, you can easily create a query that returns results that join content in a document with content in the corresponding properties document. For example, consider a document that represents a chapter in a book, and the document has properties containing the publisher of the book. you can then write a search that returns documents that match a cts:query where the document has a specific publisher, as in the following example:

```
cts:search(collection(), cts:and-query((
  cts:properties-query(
    cts:element-value-query(xs:QName("publisher"), "My Press") ),
  cts:word-query("a small good thing") )) )
```

This query returns all documents with the phrase `a small good thing` and that have a value of `My Press` in the `publisher` element in their corresponding properties document.

Similarly, you can use `cts:document-fragment-query` to join documents against properties when searching over properties.

4.4 Registering cts:query Expressions to Speed Search Performance

If you use the same complex `cts:query` expressions repeatedly, and if you are using them as an *unfiltered* `cts:query` constructor, you can register the `cts:query` expressions for later use. Registering a `cts:query` expression stores a pre-evaluated version of the expression, making it faster for subsequent queries to use the same expression. Unfiltered constructors return results directly from the indexes and return all candidate fragments for a search, but do not perform post-filtering to validate that each fragment perfectly meets the search criteria. For details on unfiltered searches, see “Using Unfiltered Searches for Fast Pagination” in the *Query Performance and Tuning Guide*.

This section describes registered queries and provides some examples of how to use them. It includes the following topics:

- Registered Query APIs
- Must Be Used Unfiltered
- Registration Does Not Survive System Restart
- Storing Registered Query IDs
- Registered Queries and Relevance Calculations
- Example: Registering and Using a cts:query Expression

4.4.1 Registered Query APIs

To register and reuse unfiltered searches for `cts:query` expressions, use the following XQuery APIs:

- `cts:register`
- `cts:registered-query`
- `cts:deregister`

For the syntax of these functions, see the *MarkLogic XQuery and XSLT Function Reference*.

4.4.2 Must Be Used Unfiltered

You can only use registered queries on unfiltered constructors; using a registered query as a filtered constructor throws the `XDMP-REGFLT` exception. To specify an unfiltered constructor, use the `"unfiltered"` option to `cts:registered-query`. For details about unfiltered searches, see "Using Unfiltered Searches for Fast Pagination" in the *Query Performance and Tuning Guide*.

4.4.3 Registration Does Not Survive System Restart

Registered queries are only stored in the memory cache, and if the cache grows too big, some registered queries might be aged out of the cache. Also, if MarkLogic Server stops or restarts, any queries that were registered are lost and must be re-registered.

If you attempt to call `cts:registered-query` in a `cts:search` and the query is not currently registered, it throws an `XDMP-UNREGISTERED` exception. Because registered queries are not guaranteed to be registered every time they are used, it is good practice to use a `try/catch` around calls to `cts:registered-query`, and re-register the query in the `catch` if the it throws an `XDMP-UNREGISTERED` exception.

For example, the following sample code shows a `cts:registered-query` call used with a try/catch expression in XQuery:

```
(: wrap the registered query in a try/catch :)
try{
xdmp:estimate(cts:search(fn:doc(),
  cts:registered-query(995175721241192518, "unfiltered")))
}
catch ($e)
{
let $registered := 'cts:register(
cts:word-query("hello*world", "wildcarded"))'
return
if ( fn:contains($e/*:code/text(), "XDMP-UNREGISTERED") )
then ( "retry this query with the following registered query ID: ",
       xdmp:eval($registered) )
else ( $e )
}
```

This code is somewhat simplified: it catches the `XDMP-UNREGISTERED` exception and simply reports what the new registered query ID is. In an application that uses registered queries, you probably would want to re-run the query with the new registered ID. Also, this example performs the `try/catch` in XQuery. If you are using XCC to issue queries against MarkLogic Server, you can instead perform the `try/catch` in the middleware Java or .NET layer.

4.4.4 Storing Registered Query IDs

When you register a `cts:query` expression, the `cts:register` function returns an integer, which is the ID for the registered query. After the `cts:register` call returns, there is no way to query the system to find the registered query IDs. Therefore, you might need to store the IDs somewhere. You can either store them in the middleware layer (if you are using XCC to issue queries against MarkLogic Server) or you can store them in a document in MarkLogic Server.

The registered query ID is generated based on a hash of the actual query, so registering the same query multiple times results in the same ID. The registered query ID is valid for all queries against the database across the entire cluster.

4.4.5 Registered Queries and Relevance Calculations

Searches that use registered queries will generate results having different scores from the equivalent searches using a non-registered queries. This is because registered queries are treated as a single term in the relevance calculation. For details on relevance calculations, see “Relevance Scores: Understanding and Customizing” on page 136.

4.4.6 Example: Registering and Using a cts:query Expression

To run a registered query, you first register the query and then run the registered query, specifying it by ID. This section describes some example steps for registering a query and then running the registered query.

1. First register the `cts:query` expression you want to run, as in the following example:

```
cts:register(cts:word-query("hello*world", "wildcarded"))
```

2. The first step returns an integer. Keep track of the integer value (for example, store it in a document).

3. Use the integer value to run a search with the registered query (with the `"unfiltered"` option) as follows:

```
cts:search(fn:doc(),
          cts:registered-query(987654321012345678, "unfiltered") )
```

4.5 Adding Relevance Information to cts:query Expressions:

The leaf-level `cts:query` APIs (`cts:word-query`, `cts:element-word-query`, and so on) have a weight parameter, which allows you to add a multiplication factor to the scores produced by matches from a query. You can use this to increase or decrease the weight factor for a particular query. For details about score, weight, and relevance calculations, see "Relevance Scores: Understanding and Customizing" on page 136.

4.6 XML Serializations of cts:query Constructors

You can create an XML serialization of a `cts:query`. The XML serialization is used by alerting applications that use a `cts:reverse-query` constructor and is also useful to perform various programmatic tasks to a `cts:query`. Alerting applications (see "Creating Alerting Applications" on page 196) find queries that would match nodes, and then perform some action for the query matches. This section describes the serialized XML and includes the following parts:

- Serializing a cts:query to XML
- Function to Construct a cts:query From XML

4.6.1 Serializing a cts:query to XML

A serialized `cts:query` has XML that conforms to the `<marklogic-dir>/Config/cts.xsd` schema, which is in the `http://marklogic.com/cts` namespace, which is bound to the `cts` prefix. You can either construct the XML directly or, if you use any `cts:query` expression within the context of an element, MarkLogic Server will automatically serialize that `cts:query` to XML. Consider the following example:

```
<some-element>{cts:word-query("hello world")}</some-element>
```

When you run the above expression, it serializes to the following XML:

```
<some-element>
  <cts:word-query xmlns:cts="http://marklogic.com/cts">
    <cts:text xml:lang="en">hello world</cts:text>
  </cts:word-query>
</some-element>
```

If you are using an alerting application, you might choose to store this XML in the database so you can match searches that include `cts:reverse-query` constructors. For details on alerts, see "Creating Alerting Applications" on page 196.

4.6.2 Add Arbitrary Annotations With cts:annotate

You can annotate your `cts:query` XML with `cts:annotate` elements. A `cts:annotate` element can be a child of any element in the `cts:query` XML, and it can consist of any valid XML content (for example, a single text node, a single element, multiple elements, complex elements, and so on). MarkLogic Server ignores these annotations when processing the query XML, but such annotations are often useful to the application. For example, you can store information about where the query came from, information about parts of the query to use or not in certain parts of the application, and so on. The following is some sample XML with `cts:annotation` elements:

```
<cts:and-query xmlns:cts="http://marklogic.com/cts">
  <cts:directory-query>
    <cts:annotation>private</cts:annotation>
    <cts:uri>/myprivate-dir/</cts:uri>
  </cts:directory-query>
  <cts:and-query>
    <cts:word-query><cts:text>hello</cts:text></cts:word-query>
    <cts:word-query><cts:text>world</cts:text></cts:word-query>
  </cts:and-query>
  <cts:annotation>
    <useful>something useful to the application here</useful>
  </cts:annotation>
</cts:and-query>
```

For another example that uses `cts:annotate` to store the original query string in a function that generates a `cts:query` from a string, see the last part of the example in "XML Serializations of cts:query Constructors" on page 131.

4.6.3 Function to Construct a cts:query From XML

You can turn an XML serialization of a cts:query back into an un-serialized cts:query with the `cts:query` function. For example, you can turn a serialized `cts:query` back into a `cts:query` as follows:

```
cts:query(
  <cts:word-query xmlns:cts="http://marklogic.com/cts">
    <cts:text>word</cts:text>
  </cts:word-query>
)
(: returns: cts:word-query("word", ("lang=en"), 1) :)
```

4.7 Example: Creating a cts:query Parser

The following sample code shows a simple query string parser that parses double-quote marks to be a phrase, and considers anything else that is separated by one or more spaces to be a single term. If needed, you can use the same design pattern to add other logic to do more complex parsing (for example, OR processing or NOT processing).

```
xquery version "1.0-ml";
declare function local:get-query-tokens($input as xs:string?)
  as element() {
(: This parses double-quotes to be exact matches. :)
<tokens>{
let $newInput := fn:string-join(
(: check if there is more than one double-quotation mark.  If there is,
   tokenize on the double-quotation mark ("), then change the spaces
   in the even tokens to the string "!+!".  This will then allow later
   tokenization on spaces, so you can preserve quoted phrases as phrase
   searches (after re-replacing the "!+!" strings with spaces).  :)
    if ( fn:count(fn:tokenize($input, '"')) > 2 )
    then ( for $i at $count in fn:tokenize($input, '"')
           return
             if ($count mod 2 = 0)
             then fn:replace($i, "\s+", "!+!")
             else $i )
    else ( $input ) , " ")
let $tokenInput := fn:tokenize($newInput, "\s+")

return (
for $x in $tokenInput
where $x ne ""
return
<token>{fn:replace($x, "!\+!", " ")}</token>)
}</tokens>
} ;

let $input := 'this is a "really big" test'
return
local:get-query-tokens($input)
```

This returns the following:

```
<tokens>
  <token>this</token>
  <token>is</token>
  <token>a</token>
  <token>really big</token>
  <token>test</token>
</tokens>
```

Now you can derive a `cts:query` expression from the tokenized XML produced above, which composes all of the terms with a `cts:and-query`, as follows (assuming the `local:get-query-tokens` function above is available to this function):

```
xquery version "1.0-ml";
declare function local:get-query($input as xs:string)
{
let $tokens := local:get-query-tokens($input)
return
 cts:and-query( (cts:and-query(
        for $token in $tokens//token
```

```
        return
        cts:word-query($token/text()) ) ))
} ;

let $input := 'this is a "really big" test'
return
local:get-query($input)
```

This returns the following (spacing and line breaks added for readability):

```
cts:and-query(
  cts:and-query((
    cts:word-query("this", (), 1),
    cts:word-query("is", (), 1),
    cts:word-query("a", (), 1),
    cts:word-query("really big", (), 1),
    cts:word-query("test", (), 1)
    ), ()) ,
  () )
```

You can now take the generated `cts:query` expression and add it to a `cts:search`.

Similarly, you can generate a serialized `cts:query` as follows (assuming the `local:get-query-tokens` function is available):

```
xquery version "1.0-ml";
declare function local:get-query-xml($input as xs:string)
{
let $tokens := local:get-query-tokens($input)
return
 element cts:and-query {
       element cts:and-query {
           for $token in $tokens//token
           return
           element cts:word-query { $token/text() } },
           element cts:annotation {$input} }
} ;

let $input := 'this is a "really big" test'
return
local:get-query-xml($input)
```

This returns the folllowing XML serialization:

```
<cts:and-query xmlns:cts="http://marklogic.com/cts">
  <cts:and-query>
    <cts:word-query>this</cts:word-query>
    <cts:word-query>is</cts:word-query>
    <cts:word-query>a</cts:word-query>
    <cts:word-query>really big</cts:word-query>
    <cts:word-query>test</cts:word-query>
  </cts:and-query>
```

```
  <cts:annotation>this is a "really big" test</cts:annotation>
</cts:and-query>
```

5.0 Relevance Scores: Understanding and Customizing

Search results in MarkLogic Server return in *relevance* order; that is, the result that is most relevant to the `cts:query` expression in the search is the first item in the search return sequence, and the least relevant is the last. There are several tools available to control the relevance score associated with a search result item. This chapter describes the different methods available to calculate relevance, and includes the following sections:

- Understanding How Scores and Relevance are Calculated
- How Fragmentation and Index Options Influence Scores
- Using Weights to Influence Scores
- Proximity Boosting With the distance-weight Option
- Interaction of Score and Quality
- Using cts:score, cts:confidence, and cts:fitness
- Relevance Order in cts:search Versus Document Order in XPath
- Sample cts:search Expressions

5.1 Understanding How Scores and Relevance are Calculated

When you perform a `cts:search` operation, MarkLogic Server produces a result set that includes items matching the `cts:query` expression and, for each matching item, a *score*. The score is a number that is calculated based on statistical information, including the number of documents in a database, the frequency in which the search terms appear in the database, and the frequency in which the search term appears in the document. The relevance of a returned search item is determined based on its score compared with other scores in the result set, where items with higher scores are deemed to be more relevant to the search. By default, search results are returned in relevance order, so changing the scores can change the order in which search results are returned.

As part of a `cts:search` expression, you can specify the following different methods for calculating the score, each of which uses a different formula in its score calculation:

- log(tf)*idf Calculation
- log(tf) Calculation
- Simple Term Match Calculation
- Random Score Calculation
- Term Frequency Normalization

5.1.1 log(tf)*idf Calculation

The `logtfidf` method of relevance calculation is the default relevance calculation, and it is the option `score-logtfidf` of `cts:search`. The `logtfidf` method takes into account term frequency (how often a term occurs in a single fragment) and document frequency (in how many documents does the term occur) when calculating the score. Most search engines use a relevance formula that is derived by some computation that takes into account term frequency and document frequency.

The `logtfidf` method (the default scoring method) uses the following formula to calculate relevance:

```
log(term frequency) * (inverse document frequency)
```

The `term frequency` is a normalized number representing how many terms are in a document. The term frequency is normalized to take into account the size of the document, so that a word that occurs 10 times in a 100 word document will get a higher score than a word that occurs 100 times in a 1,000 word document.

The `inverse document frequency` is defined as:

```
log(1/df)
```

where `df` (document frequency) is the number of documents in which the term occurs.

For most search-engine style relevance calculations, the `score-logtfidf` method provides the most meaningful relevance scores. Inverse document frequency (IDF) provides a measurement of how "information rich" a document is. For example, a search for "the" or "dog" would probably put more emphasis on the occurences of the term "dog" than of the term "the".

5.1.2 log(tf) Calculation

The option `score-logtf` for `cts:search` computes scores using the `logtf` method, which does not take into account how many documents have the term. The `logtf` method uses the following formula to calculate scores:

```
log(term frequency)
```

where the `term frequency` is a normalized number representing how many terms are in a document. The term frequency is normalized to take into account the size of the document, so that a word that occurs 10 times in a 100 word document will get a higher score than a word that occurs 100 times in a 1,000 word document.

When you use the `logtf` method, scores are based entirely on how many times a document matches the search term, and does not take into account the "information richness" of the search terms.

5.1.3 Simple Term Match Calculation

The option `score-simple` on `cts:search` performs a simple term-match calculation to compute the scores. The `score-simple` method gives a score of 8*weight for each matching term in the `cts:query` expression, and then scales the score up by multiplying by 256. It does not matter how many times a given term matches (that is, the term frequency does not matter); each match contributes 8*weight to the score. For example, the following query (assume the default weight of 1) would give a score of 8*256=2048 for any fragment with one or more matches for “hello”, a score of 16*256=4096 for any fragment that also has one or more matches for “goodbye”, or a score of zero for fragments that have no matches for either term:

```
cts:or-query(("hello", "goodbye"))
```

Use this option if you want the scores to only reflect whether a document matches terms in the query, and you do not want the score to be relative to frequency or “information-richness” of the term.

5.1.4 Random Score Calculation

The option `score-random` on `cts:search` computes a randomly-generated score for each search match. You can use this to randomly choose fragments matching a query. If you perform the same search multiple times using the `score-random` option, you will get different ordering each time (because the scores are randomly generated at runtime for each search).

5.1.5 Term Frequency Normalization

The scoring methods that take into account term frequency (`score-logtfidf` and `score-logtf`) will, by default, normalize the term frequency (how many search term matches there are for a document) based on the size of the document. The idea of this normalization is to take into account how frequent a term occurs in the document, relative to the other documents in the database. You can think of this is the density of terms in a document, as opposed to simply the frequency of the terms. The term frequency normalization makes a document that has, for example, 10 occurrences of the word `"dog"` in a 10,000,000 word document have a lower relevance than a document that has 10 occurrences of the word `"dog"` in a 100 words document. With the default term frequency normalization of `scaled-log`, the smaller document would have a higher score (and therefore be more relevant to the search), because it has a greater “term density” of the word `"dog"`. For most search applications, this behavior is desirable.

If you would like to change that behavior, you can set the `tf normalization` option on the database configuration to lessen or eliminate the effects of the size of the matching document in the score calculation, which in turn would strengthen the effect of its term frequency (the number of matches in that document). The `unscaled-log` option does no scaling based on document size, and the `scaled-log` option (the default) does the maximum scaling of the document based on document size. Additionally, there are four intermediate settings, `weakest-scaled-log`, `weakly-scaled-log`, `moderately-scaled-log`, and `strongly-scaled-log`, which have increasing degrees of scaling in between none and the most scaling. If you change this setting in the database and `reindexer enable` is set to `true`, then the database will begin reindexing.

5.2 How Fragmentation and Index Options Influence Scores

Scores are calculated based on index data, and therefore based on unfiltered searches. That has several implications to scores:

- Scores are fragment-based, so term frequency and document frequency are calculated based on term frequency per fragment and fragment frequency respectively.
- Scores are based on unfiltered searches, so they include false-positive results.

Because scores are based on fragments and unfiltered searches, index options will affect scores, and in some case will make the scores more "accurate"; that is, base the scores on searches that return fewer false-positive results. For example, if you have `word positions` enabled in the database configuration, searches for three or more term phrases will have fewer false-positive matches, thereby improving the accuracy of the scores.

For details on unfiltered searches and how you can tell if there are false-positive matches, see "Using Unfiltered Searches for Fast Pagination" in the *Query Performance and Tuning Guide*.

5.3 Using Weights to Influence Scores

Use a weight in a query sub-expression to either boost or lower the sub-expression contribution to the relevance score.

For example, you can specify weights for leaf-level `cts:query` constructors, such as `cts:word-query` and `cts:element-value-query`; for details, see *XQuery and XSLT Reference Guide*. You can also specify weights in the equivalent Search API abstractions, such as the structured query constructs `value-query` and `word-constraint-query`, or when defining a word or value constraint in query options.

The default weight is 1.0. Use the following guidelines for choosing custom weights:

- To boost the score contribution, set the weight higher than 1.0.
- To lower the score contribution, set the weight between 0 and 1.0.
- To contribute nothing to the score, set the weight to 0.
- To make the score contribution negative, set the weight to a negative number.

Scores are normalized, so a weight is not an absolute multiplier on the score. Instead, weights indicate how much terms from a given query sub-expression are weighted in comparison to other sub-expressions in the same expression. A weight of 2.0 doubles the contribution to the score for terms that match that query. Similarly, a weight of 0.5 halves the contribution to the score for terms that match that query. In some cases, the score reaches a maximum, so a weight of 2.0 and a weight of 20,000 can yield the same contribution to the score.

Adding weights is particularly useful if you have several components in a query expression, and you want matches for some parts of the expression to be weighted more heavily than other parts. For an example of this, see “Increase the Score for some Terms, Decrease for Others” on page 145.

5.4 Proximity Boosting With the distance-weight Option

If you have the `word positions` indexing option enabled in your database, you can use the `distance-weight` option to the leaf-level `cts:query` constructors, and then all of the terms passed into that `cts:query` constructors will consider the proximity of the terms to each other for the purposes of scoring. This proximity boosting will make documents with matches close together have higher scores. Because search results are sorted by score, it will have the effect of making documents having the search terms close together have higher relevance ranking. This section provides some examples that use the `distance-weight` option along with explanations of the examples, and includes the following parts:

- Example of Simple Proximity Boosting
- Using Proximity Boosting With cts:and-query Semantics
- Using cts:near-query to Achieve Proximity Boosting

5.4.1 Example of Simple Proximity Boosting

The distance weight is only applied to the matches for `cts:query` constructors in which the `distance-weight` occurs. For example, consider the following `cts:query` constructor:

```
cts:word-query(("cat", "dog")), "distance-weight=3")
```

If one document has an instance of `"cat"` very near `"dog"`, and another document has the same number of `"cat"` and `"dog"` terms, but they are not very near, then the one with the `"cat"` near `"dog"` will have a higher score.

For example, consider the following:

```
xquery version "1.0-ml";
(: make sure word positions are enabled in the database :)
(:
   create 3 documents, then run two searches, one with
   distance-weight and one without, printing out the scores
:)
xdmp:document-insert("/2.xml",
  <p>The cat is pretty near a dog.</p>) ;

xdmp:document-insert("/1.xml",
  <p>The cat dog is very near.</p>) ;

xdmp:document-insert("/3.xml",
  <p>The cat is not very near the very large dog.</p>) ;

for $x in (cts:search(fn:doc(), cts:word-query(("cat", "dog") ,
```

```
                                    "distance-weight=3" ) ),
            cts:search(fn:doc(), cts:word-query(("cat", "dog") ) ) )
return
element hit{attribute uri {xdmp:node-uri($x)},
            attribute score {cts:score($x)},
            attribute text{fn:string($x/p)}}
```

This returns the following results:

```
<hit uri="/1.xml" score="146" text="The cat dog is very near."/>
<hit uri="/2.xml" score="140" text="The cat is pretty near a dog."/>
<hit uri="/3.xml" score="135"
     text="The cat is not very near the very large dog."/>
<hit uri="/3.xml" score="72"
     text="The cat is not very near the very large dog."/>
<hit uri="/2.xml" score="72" text="The cat is pretty near a dog."/>
<hit uri="/1.xml" score="72" text="The cat dog is very near."/>
```

Notice that the first three hits use the `distance-weight`, and the ones with the terms closer together have higher scores, and thus rank higher in the search. The last three hits have the same score because they all have the same number of each term in the `cts:query` and there is no proximity taken into account in the scores.

5.4.2 Using Proximity Boosting With cts:and-query Semantics

Because the `distance-weight` option applies to the terms in individual `cts:query` constructors, the terms are combined as an or-query (that is, any term match is a match for the query). Therefore, the example above would also return results for documents that contain `"cat"` and not `"dog"` and vice versa. If you want to have and-query semantics (that is, all terms must match for the query to match) and also have proximity boosting, you will have to construct a `cts:query` that does an and of all of the terms in addition to the `cts:query` with the `distance-weight` option.

For example:

```
xquery version "1.0-ml";
cts:search(fn:doc(), cts:and-query((
                       cts:word-query("cat"),
                       cts:word-query("dog"),
                       cts:word-query(("cat", "dog") ,
                               "distance-weight=3" ) )) )
```

The difference between this query and the previous one is that the previous one would return a document that contained `"cat"` but not `"dog"` (or vice versa), and this one will only return documents containing both `"cat"` and `"dog"`.

If you have a large corpus of documents and you expect to have many matches for your searches, then you might find you do not need to use the `cts:and-query` approach. The reason a large corpus has an effect is because document frequency is taken into account in the relevance calculation, as described in "Understanding How Scores and Relevance are Calculated" on page 136. You might find that the most relevant documents still float to the top of your search even without the `cts:and-query`. What you do will depend on your application requirements, your preferences, and your data.

5.4.3 Using cts:near-query to Achieve Proximity Boosting

Another technique that makes results with closer proximity have higher scores is to use `cts:near-query`. Searches that use the `cts:near-query` constructor will take proximity into account when calculating scores, as long as the `word positions` index option is enabled in the database. Additionally, you can use the `distance-weight` parameter to further boost the effect of proximity on scoring.

Because `cts:near-query` takes a `distance` argument, you have to think about how near you want results to be in order for them to match. With the `distance` parameter to `cts:near-query`, there is a tradeoff between the size of the `distance` and performance. The higher the number for the `distance`, the more work MarkLogic Server does to resolve the query. For many queries, this amount of work might be very small, but for some complex queries it can be noticeable.

To construct a query that uses `cts:near-query` for proximity boosting, pass the `cts:query` for your search as the first parameter to a `cts:near-query`, and optionally add a `distance-weight` parameter to further boost the proximity. The `cts:near-query` matches will always take distance into account, but setting a `distance-weight` will further boost the proximity weight. For example, consider how the following query, which uses the same data as the above examples, produces similar results:

```
xquery version "1.0-ml";
cts:search(fn:doc(),
   cts:near-query(
     cts:and-query((
        cts:word-query("cat"),
        cts:word-query("dog") )),
     1000, (), 3) )
```

This query uses a `distance` of 1,000, therefore documents that have `"cat"` and `"dog"` that are more than 1,000 words apart are not included in its result. The size you use is dependent on your data and the performance characteristics of your searches. If you were more concerned about missing document where the matches are more than 1,000 words away, then you should raise that number; if you are seeing performance issues and want faster performance, and you are OK with missing results that are above the distance threshold (which are probably not relevant anyway), then you should make the number smaller. For databases with a large amount of documents, keep in mind

that not returning the documents with words that are far apart from each other will probably result in very similar search results, especially for the most relevant hits (because the results with the matches far apart have low relevance scores compared to the ones that have matches close together).

5.5 Interaction of Score and Quality

Each document contains a quality value, and is set either at load time or with `xdmp:document-set-quality`. You can use the optional `$QualityWeight` parameter to `cts:search` to force document quality to have an impact on scores. The scores are then determined by the following formula:

```
Score = Score + (QualityWeight * Quality)
```

The default of `QualityWeight` is 1.0 and the default quality on a document is 0, so by default, documents without any quality set have no quality impact on score. Documents that do have quality set, however, will have impact on the scores by default (because the default `QualityWeight` is 1, effectively boosting the score by the document quality).

If you want quality to have a smaller impact on the score, set the `QualityWeight` between 0 and 1.0. If you want the quality to have no impact on the score, set the `QualityWeight` to 0. If you want the quality to have a larger impact on raising the score, set the `QualityWeight` to a number greater than 1.0. If you want the quality to have a negative effect on scores, set the `QualityWeight` to a negative number or set document quality to a negative number.

> **Note:** If you set document quality to a negative number and if you set `QualityWeight` to a negative number, it will boost the score with a positive number.

5.6 Using cts:score, cts:confidence, and cts:fitness

You can get the score for a result node by calling `cts:score` on that node. The score is a number, where higher numbers indicate higher relevance for that particular result set.

Similarly, you can get the confidence by calling `cts:confidence` on a result node. The confidence is a number (of type `xs:float`) between 0.0 and 1.0. The confidence number does not include any quality settings that might be on the document. Confidence scores are calculated by first bounding the scores between 0 and 1.0, and then taking the square root of the bounded number.

As an alternate to `cts:confidence`, you can get the fitness by calling `cts:fitness` on a result node. The fitness is a number (of type `xs:float`) between 0.0 and 1.0. The fitness number does not include any quality settings that might be on the document, and it does not use document frequency in the calculation. Therefore, `cts:fitness` returns a number indicating how well the returned node satisfies the query issued, which is subtly different from relevance, because it does not take into account other documents in the database.

5.7 Relevance Order in cts:search Versus Document Order in XPath

When understanding the order an expression returns in, there are two main rules to consider:

- `cts:search` expressions always return in relevance order (the most relevant to the least relevant).
- XPath expressions always return in document order.

A subtlety to note about these rules is that if a `cts:search` expression is followed by some XPath steps, it turns the expression into an XPath expression and the results are therefore returned in document order. For example, consider the following query:

```
cts:search(fn:doc(), "my search phrase")
```

This returns a relevance-ordered sequence of document nodes that contain the specified phrase. You can get the scores of each node by using `cts:score`. Things will change if you then add an XPath step to the expression as follows:

```
cts:search(fn:doc(), "my search phrase")//TITLE
```

This will now return a *document-ordered* sequence of `TITLE` elements. Also, in order to compute the answer to this query, MarkLogic Server must first perform the search, and then reorder the search in document order to resolve the XPath expression. If you need to perform this type of query, it is usually more efficient (and often *much* more efficient) to use `cts:contains` in an XPath predicate as follows:

```
fn:doc()[cts:contains(., "my search phrase")]//TITLE
```

Note: In most cases, this form of the query (all XPath expression) will be much more efficient than the previous form (with the XPath step after the `cts:search` expression). There might be some cases, however, where it might be less efficient, especially if the query is highly selective (does not match many fragments).

When you write queries as XPath expressions, MarkLogic Server does not compute scores, so if you need scores, you will need to use a `cts:search` expression. Also, if you need a query like the above examples but need the results in relevance order, then you can put the search in a `FLWOR` expression as follows:

```
for $x in cts:search(fn:doc(), "my search phrase")
return
$x//TITLE
```

This is more efficient than the `cts:search` with an XPath step following it, and returns relevance-ranked and scored results.

5.8 Sample cts:search Expressions

This section lists several `cts:search` expressions that include weight and/or quality parameters. It includes the following examples:

- Magnify the Score Boost for Documents With Quality
- Increase the Score for some Terms, Decrease for Others

5.8.1 Magnify the Score Boost for Documents With Quality

The following search will make any documents that have a quality set (set either at load time or with `xdmp:document-set-quality`) give much higher scores than documents with no quality set.

```
cts:search(fn:doc(), cts:word-query("my phrase"), (), 3.0)
```

Note: For any documents that have a quality set to a negative number less than -1.0, this search will have the effect of lowering the score drastically for matches on those documents.

5.8.2 Increase the Score for some Terms, Decrease for Others

The following search will boost the scores for documents that satisfy one query while decreasing the scores for documents that satisfy another query.

```
cts:search(fn:doc(), cts:and-query((
  cts:word-query("alfa", (), 2.0), cts:word-query("lada", (), 0.5)
  )) )
```

This search will boost the scores for documents that contain the word `alfa` while lowering the scores for document that contain the word `lada`. For documents that contain both terms, the component of the score from the word `alfa` is boosted while the component of the score from the word `lada` is lowered.

6.0 Browsing With Lexicons

MarkLogic Server allows you to create *lexicons*, which are lists of unique words or values, either throughout an entire database (words only) or within named elements or attributes (words or values). Also, you can define lexicons that allow quick access to the document and collection URIs in the database, and you can create word lexicons on named fields. This chapter describes the lexicons you can create in MarkLogic Server and describes how to use the API to browse through them. This chapter includes the following sections:

- About Lexicons
- Creating Lexicons
- Word Lexicons
- Element/Element-Attribute/Path Value Lexicons
- Field Value Lexicons
- Value Co-Occurrences Lexicons
- Geospatial Lexicons
- Range Lexicons
- URI and Collection Lexicons
- Performing Lexicon-Based Queries

6.1 About Lexicons

A *word lexicon* stores all of the unique, case-sensitive, diacritic-sensitive words, either in a database, in an element defined by a QName, or in an attribute defined by a QName. A *value lexicon* stores all of the unique values for an element or an attribute defined by a QName (that is, the entire and exact contents of the specified element or attribute). A *value co-occurrences lexicon* stores all of the pairs of values that appear in the same fragment. A *geospatial lexicon* returns geospatial values from the geospatial index. A *range lexicon* stores buckets of values that occur within a specified range of values. A *URI lexicon* stores the URIs of the documents in a database, and a *collection lexicon* stores the URIs of all collections in a database.

All lexicons determine their order and uniqueness based on the collation specified (for `xs:string` types), and you can create multiple lexicons on the same object with different collations. For information on collations, see “Collations” on page 304. You can also create value lexicons on non-string values.

All of these types of lexicons have the following characteristics:

- Lexicon terms and values are case-sensitive.
- Lexicon terms and values are unstemmed.
- Lexicon terms and values are diacritic-sensitive.
- Lexicon terms and values do not have any relevance information associated with them.
- Uniqueness in lexicons is based on the specified collation of the lexicon.
- Lexicon terms in word lexicons do not include any punctuation. For example, the term `case-sensitive` in a database will be two terms in the lexicon: `case` and `sensitive`.
- Lexicon values in value lexicons do include punctuation.
- In order to perform lexicon-based queries, the appropriate lexicon must be created. If the lexicon has not been created, the lexicon query will throw an exception.
- Lexicons are used with the Search API to create constraints. Lexicons based on range indexes are used to create value constraints, which are used for facets. For details on the Search API, constraints, and facets, see "Search API: Understanding and Using" on page 16.

Even though the lexicons store terms case-sensitive, unstemmed, and diacritic-sensitive, you can still do case-insensitive and diacritic-insensitive lexicon-based queries by specifying the appropriate option(s). For details on the syntax, see the *MarkLogic XQuery and XSLT Function Reference*.

6.2 Creating Lexicons

You must create the appropriate lexicon before you can run lexicon-based queries. You create lexicons in the Admin Interface. For detailed information on creating lexicons, see the "Text Indexing" and "Element/Attribute Range Indexes and Lexicons" chapters of the *Administrator's Guide*. For all of the lexicons, you must complete at least one of the following before you can successfully run lexicon-based queries:

- Create/enable the lexicon before you load data into the database, or
- Reindex the database after creating/enabling the lexicon, or
- Reload the data after creating/enabling the lexicon.

The following is a brief summary of how to create each of the various types of lexicons:

- To create a word lexicon for the entire database, enable the `word lexicon` setting on the Admin Interface Database Configuration page (Databases > *db_name*) and specify a collation for the lexicon (for example, `http://marklogic.com/collation/` for the UCA Root Collation).

- To create an element word lexicon, specify the element namespace URI, localname, and collation on the Admin Interface Element Word Lexicon Configuration page (Databases > *db_name* > Element Word Lexicons).
- To create an element attribute word lexicon, specify the element and attribute namespace URIs, localnames, and collation on the Admin Interface Element Attribute Word Lexicon Configuration page (Databases > *db_name* > Attribute Word Lexicons).
- To create an element value lexicon, specify the element namespace URI and localname, the collation (for `xs:string`), and the type (for example, `xs:string`) on the Admin Interface Range Element Index Configuration page (Databases > *db_name* > Element Indexes).
- To create an element attribute value lexicon, specify the element and attribute namespace URIs and localnames, the collation (for `xs:string`), and the type (for example, `xs:string`) on the Admin Interface Range Element-Attribute Index Configuration page (Databases > *db_name* > Attribute Indexes).
- To create a field value lexicon, first create a field in the Admin Interface (Databases > *db_name* > Fields). Then create the field value lexicon by specifying the type (for example, `xs:string`) and the field name on the Admin Interface Field Range Index Configuration page (Databases > *db_name* > Field Range Indexes).

  Note: If your system is set to reindex/refragment, newly created lexicons will not be available until reindexing is completed.

6.3 Word Lexicons

There are several types of word lexicons:

- Word Lexicon for the Entire Database
- Element/Element-Attribute Word Lexicons
- Field Word Lexicons

6.3.1 Word Lexicon for the Entire Database

A word lexicon covers the entire database, and holds all of the unique terms in the database, with uniqueness determined by the specified collation. You enable the word lexicon in the database page of the Admin Interface by enabling the `word lexicon` database setting. If the database already has content loaded, you must reindex the database before you can perform any lexicon queries. The following are the APIs for the word lexicon:

- `cts:words`
- `cts:word-match`

6.3.2 Element/Element-Attribute Word Lexicons

An element word lexicon or an element-attribute word lexicon contains all of the unique terms in the specified element or attribute, with uniqueness determined by the specified collation. The element word lexicons only contain words that exist in immediate text node children of the specified element as well as any text node children of elements defined in the Admin Interface as element-word-query-throughs or phrase-throughs; it does not include words from any other children of the specified element. You create element and element-attribute word lexicons in the Admin Interface with the `Element Range Indexes` and `Attribute Range Indexes` links under the database in which you want to create the lexicons. The following are the APIs for the element and element-attribute word lexicons:

- `cts:element-words`
- `cts:element-word-match`
- `cts:element-attribute-words`
- `cts:element-attribute-word-match`

6.3.3 Field Word Lexicons

A field is a named object that you create at the database level, and it defines a set of elements which can be accessed together through the field. You can create word lexicons on fields, which list all of the unique words that are included in the field. You can create field word lexicons in the configuration page for each field. Like all other lexicons, field word lexicons are unique to a collation, and you can, if you need to, create multiple lexicons in different collations. For details on fields, see Fields Database Settings in the *Administrator's Guide*. The following are the APIs for the field word lexicons:

- `cts:field-words`
- `cts:field-word-match`

6.4 Element/Element-Attribute/Path Value Lexicons

An element value lexicon, element-attribute value lexicon, or a path value lexicon contains all of the unique values in the specified element or attribute. The values are the entire and exact contents of the specified element or attribute. You create element and element-attribute value lexicons in the Admin Interface by creating a range index of a particular type (for example, `xs:string`) for the element or attribute to which you want the value lexicon. The following are the APIs for the element, element-attribute, and path value lexicons:

- `cts:element-values`
- `cts:element-value-match`
- `cts:element-attribute-values`
- `cts:element-attribute-value-match`
- `cts:values`
- `cts:value-match`

The `cts:element-values` and `cts:element-value-match` functions are used to return values from element value lexicons implemented using element range indexes. The `cts:element-attribute-values` and `cts:element-attribute-value-match` functions are used to return values from attribute value lexicons implemented using attribute range indexes. The `cts:values` and `cts:value-match` functions are used to return values from path value lexicons implemented using path range indexes. A path value lexicon can be either an element or an attribute.

> **Note:** You can only create element value lexicons on simple elements (that is, the elements cannot have any element children).

When you have a value lexicon on an element or an attribute, you can also use the `cts:frequency` API to get fast and accurate counts of how many times the value occurs. You can either get counts of the number of fragments that have at least one instance of the value (using the default `fragment-frequency` option to the value lexicon APIs) or you can get total counts of values in each item (using the `item-frequency` option). For details and examples, see the documentation for `cts:frequency` and for the value lexicon APIs in the *MarkLogic XQuery and XSLT Function Reference*.

6.5 Field Value Lexicons

A field value lexicon contains all of the unique values for the specified field. You create field value lexicons in the Admin Interface by creating a range index of a particular type (for example, `xs:string`) for the field to which you want a field lexicon. The following are the APIs for field value lexicons:

- `cts:field-values`
- `cts:field-value-match`

When you have a value lexicon on a field, you can also use the `cts:frequency` API to get fast and accurate counts of how many times the value occurs. You can either get counts of the number of fragments that have at least one instance of the value (using the default `fragment-frequency` option to the value lexicon APIs) or you can get total counts of values in each item (using the `item-frequency` option). For details and examples, see the documentation for `cts:frequency` and for the value lexicon APIs in the *MarkLogic XQuery and XSLT Function Reference*.

Field value lexicons are useful in cases where something you want to treat as a discreet value does not occur in a single element or attribute. For example, consider the following XML structure:

```
<name>
  <first>Raymond</first>
  <middle>Clevie</middle>
  <last>Carver</last>
</name>
```

If you want to normalize names in the form `firstname lastname`, then you can create a field on this structure. The field might include the element `name` and exclude the element `middle`. The value of this instance of the field would then be `Raymond Carver`. If your document contained other name elements with the same structure, their values would be derived similarly. The range index for the field stores each unique instance of the field value.

For details on fields, see Fields Database Settings in the *Administrator's Guide*.

6.6 Value Co-Occurrences Lexicons

Value co-occurrence lexicons find pairs of element or attribute values that occur in the same fragment. If you have positions enabled in your range indexes, you can also specify a maximum word distance (`proximity=`*`N`* option) that the values must be from each other in order to match as a co-occurring pair. The following APIs support these lexicons:

- `cts:element-value-co-occurrences`
- `cts:element-attribute-value-co-occurrences`
- `cts:value-co-occurrences`
- `cts:field-value-co-occurrences`

These APIs return XML structures containing the pairs of co-occurring values. You can use `cts:frequency` on the output of these functions to find the frequency (the counts) of each co-occurrence.

Additionally, you can get co-occurrences from geospatial lexicons, as described in "Geospatial Lexicons" on page 153.

Note: Because the URI and collection lexicons are implemented as range indexes, you can specify a special QName for the document URI or collection URI lexicons to get the list of values with their URI or collections. The QNames are in the `http://marklogic.com/xdmp` namespace and the URI index has the local name `document` and the collection index has the local name `collection`, both using the `http://marklogic.com/collation/codepoint` collation. You can then use these QNames (for example, `xdmp:document` and `xdmp:collection`, as `xdmp` is bound to that namespace by default in the `1.0-ml` dialect) in `cts:element-value-co-occurrences` as one of the element QNames to find element value/document URI pairs or element value/collection URI pairs. Make sure to also specify the codepoint collation option for these QNames (for example, `"collation-2=http://marklogic.com/collation/codepoint"` if you are specifying one of these QNames as the second argument to `cts:element-value-co-occurences`).

Consider the following example (note that this example uses entity enrichment, which is a separately licensed option and you need an entity enrichment license to run this example):

```
xquery version "1.0-ml";
(:
   Before running this, create two string
   element range indexes: one for the e:person element
   and one for the e:location element, where e is bound
   to the namespace http://marklogic.com/entity.
:)
import module namespace entity="http://marklogic.com/entity" at
       "/MarkLogic/entity.xqy";
let $x := <text>George Washington was the first President
                of the United States. Martha Washington was
                his wife.  They lived at Mount Vernon.</text>
return
xdmp:document-insert("/george.xml", entity:enrich($x))
```

The `/george.xml` document created looks like the following:

```
<text>
  <e:person xmlns:e="http://marklogic.com/entity">George
  Washington</e:person> was the first President of the
  <e:gpe xmlns:e="http://marklogic.com/entity">United States</e:gpe>.
  <e:person xmlns:e="http://marklogic.com/entity">Martha
  Washington</e:person> was his wife.  They lived at
  <e:location xmlns:e="http://marklogic.com/entity">Mount
  Vernon</e:location>.
</text>
```

Now you can run the following co-occurrence query to find all co-occurring people and locations:

```
xquery version "1.0-ml";

declare namespace e="http://marklogic.com/entity";
cts:element-value-co-occurrences(xs:QName("e:person"),
     xs:QName("e:location"))
```

This produces the following output:

```
<cts:co-occurrence xmlns:cts="http://marklogic.com/cts"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <cts:value xsi:type="xs:string">George Washington</cts:value>
  <cts:value xsi:type="xs:string">Mount Vernon</cts:value>
</cts:co-occurrence>
<cts:co-occurrence xmlns:cts="http://marklogic.com/cts"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <cts:value xsi:type="xs:string">Martha Washington</cts:value>
  <cts:value xsi:type="xs:string">Mount Vernon</cts:value>
</cts:co-occurrence>
```

If you wanted to get the frequency of how many of each co-occurring pair exist, either in each item or in each fragment (depending on whether you use the `item-frequency` or the default `fragment-frequency` option), use `cts:frequency` on the lexicon lookup as follows:

```
xquery version "1.0-ml";
declare namespace e="http://marklogic.com/entity";
for $x in cts:element-value-co-occurrences(xs:QName("e:person"),
               xs:QName("e:location"))
return cts:frequency($x)
(:
   returns a frequency of 1 for each pair if /george.xml
   is the only document in the database
:)
```

6.7 Geospatial Lexicons

The following APIs support the geospatial lexicons:

- `cts:element-attribute-pair-geospatial-boxes`
- `cts:element-attribute-pair-geospatial-value-match`
- `cts:element-attribute-pair-geospatial-values`
- `cts:element-attribute-value-geospatial-co-occurrences`
- `cts:element-child-geospatial-boxes`
- `cts:element-child-geospatial-value-match`
- `cts:element-child-geospatial-values`
- `cts:element-geospatial-boxes`
- `cts:element-geospatial-value-match`
- `cts:element-geospatial-values`
- `cts:element-pair-geospatial-boxes`
- `cts:element-pair-geospatial-value-match`
- `cts:element-pair-geospatial-values`
- `cts:element-value-geospatial-co-occurrences`

You must create the appropriate geospatial index to use its corresponding geospatial lexicon. For example, to use `cts:element-geospatial-values`, you must first create a geospatial element index. Use the Admin Interface (Databases > *database_name* > Geospatial Indexes) or the Admin API to create geospatial indexes for a database.

The `*-boxes` APIs return XML elements that show buckets of ranges, each bucket containing one or more `cts:box` values.

6.8 Range Lexicons

The range lexicons return values divided into buckets. The ranges are ranges of values of the type of the lexicon. A range index is required on the element(s) or attribute(s) specified in the range lexicon. The following APIs support these lexicons:

- `cts:element-attribute-value-ranges`
- `cts:element-value-ranges`
- `cts:value-ranges`
- `cts:field-value-ranges`

Additionally, there are the following geospatial box lexicons to find ranges of geospatial values divided into buckets:

- `cts:element-attribute-pair-geospatial-boxes`
- `cts:element-child-geospatial-boxes`
- `cts:element-geospatial-boxes`
- `cts:element-pair-geospatial-boxes`

The range lexicons return a sequence of XML nodes, one node for each bucket. You can use `cts:frequency` on the result set to determine the number of items (or fragments) in the buckets. The `"empties"` option specifies that an XML node is returned for buckets that have no values (that is, for buckets with a frequency of zero). By default, empty buckets are not included in the result set. For details about all of the options to the range lexicons, see the *MarkLogic XQuery and XSLT Function Reference*.

6.9 URI and Collection Lexicons

The URI and Collection lexicons respectively list all of the document URIs and all of the collection URIs in a database. To enable or disable these lexicons, use the Database Configuration page in the Admin Interface. Use these lexicons to quickly search through all of the URIs in a database. The following APIs support these lexicons:

- `cts:collection-match`
- `cts:collections`
- `cts:uri-match`
- `cts:uris`

6.10 Performing Lexicon-Based Queries

Lexicon queries return a sequence of words (or values in the case of value lexicons) from the appropriate lexicon. For string values, the words or values are returned in collation order, and the terms are case- and diacritic-sensitive. For other data types, the values are returned in order, where values that are "greater than" return before values that are "less than". This section lists the lexicon APIs and provides some examples and explanation of how to perform lexicon-based queries. It includes the following parts:

- Lexicon APIs
- Constraining Lexicon Searches to a cts:query Expression
- Using the Match Lexicon APIs
- Determining the Number of Fragments Containing a Lexicon Term

6.10.1 Lexicon APIs

Use the following Search Built-in XQuery APIs to perform lexicon-based queries:

- `cts:words`
- `cts:word-match`
- `cts:element-words`
- `cts:element-word-match`
- `cts:element-attribute-words`
- `cts:element-attribute-word-match`
- `cts:element-values`
- `cts:element-value-match`
- `cts:element-attribute-values`
- `cts:element-attribute-value-match`
- `cts:values`
- `cts:value-match`
- `cts:field-values`
- `cts:field-value-match`
- `cts:collection-match`
- `cts:collections`
- `cts:uri-match`
- `cts:uris`

In order to perform lexicon-based queries, the appropriate lexicon must be created. If the lexicon has not been created, the lexicon query will throw an exception.

The `cts:*-words` APIs return all of the words in the lexicon (or all of the words from a starting point if the optional `$start` parameter is used). The `cts:*-match` APIs return only words in the lexicon that match the wildcard pattern.

For details about the individual functions, see the Search APIs in the *MarkLogic XQuery and XSLT Function Reference*.

6.10.2 Constraining Lexicon Searches to a cts:query Expression

You can use the `$query` option of the lexicon APIs to constrain your lexicon lookups to fragments matching a particular `cts:query` expression. When you specify the `$query` option, the lexicon search returns all of the terms (or values for lexicon value queries) in the fragments that match the specified `cts:query` expression.

For example, the following is a query against a database of all of Shakespeare's plays fragmented at the SCENE level:

```
cts:words("et", (), "et tu")[1 to 10]

=> et ete even ever every eyes fais faith fall familiar
```

This query returns the first 10 words from the lexicon of words, starting with the word `et`, for all of the fragments that match the following query:

```
cts:word-query("et tu")
```

In the case of the Shakespeare database, there are 2 scenes that match this query, one from *The Tragedy of Julius Caesar* and one from *The Life of Henry the Fifth*. Note that this is a different set of words than if you omitted the `$query` parameter from the search. The following shows the query without the `$query` parameter. The results represent the 10 words in the entire word lexicon for all of the Shakespeare plays, starting with the word `et`:

```
cts:words("et")

=> et etc etceteras ete eternal eternally eterne eternity
   eternized etes
```

Note that when you constrain a lexicon lookup to a `cts:query` expression, it returns the lexicon items for any fragment in which the `cts:query` expression returns `true`. No filtering is done to the `cts:query` expression to validate that the match actually occurs in the fragment. In some cases, depending on the index options you have set, it can return `true` in cases where there is no actual match. For example, if you do not have `fast element word searches` enabled in the database configuration, it is possible for a `cts:element-word-query` to match a fragment because both the word and the element exist in the fragment, but not in the same element. The filtering stage of `cts:search` resolves these discrepancies, but they are not resolved in lexicon APIs that use the `$query` option. For details about how this works, see Understanding the Search Process and Understanding Unfiltered Searches sections in the *Query Performance and Tuning Guide*.

6.10.3 Using the Match Lexicon APIs

Each type of lexicon (word, element word, element-attribute word, element value, and element-attribute value) has a function (`cts:*-match`) which allows you to use a wildcard pattern to constrain the lexicon entries returned; the `cts:*-match` APIs return only words or values in the lexicon that match the wildcard pattern. The following query finds all of the words in the lexicon that start with `zou`:

```
cts:word-match("zou*")

=> Zounds zounds
```

It returns both the uppercase and lowercase words that match because search defaults to case-insensitive when all of the letters in the base of the wildcard pattern are lowercase. If you want to match the pattern case-sensitive, diacritic-sensitive, or with some other option, add the appropriate option to the query. For example:

```
cts:word-match("zou*", "case-sensitive")

=> zounds
```

For details on the query options, see the *MarkLogic XQuery and XSLT Function Reference*. For details on wildcard searches, see “Understanding and Using Wildcard Searches” on page 231.

6.10.4 Determining the Number of Fragments Containing a Lexicon Term

The lexicon contains the unique terms in a database. To minimize redundant disk I/Os when you are performing estimates following a query-constrained word lexicon lookup, and therefore for this type of query to be resolved as efficiently as possible, the `cts:word-query` should have the following characteristics:

- Specify the `unstemmed`, `case-sensitive`, and `diacritic-sensitive` options.
- Specify a `weight` of 0.

These characteristics ensure that the word being estimated is exactly the same as the word returned from the lexicon.

For example, if you want to figure out how many fragments contain a lexicon term, you can perform a query like the following:

```
<words>{
for $word in cts:words("aardvark", (),
     cts:directory-query("/", "infinity"))[1 to 1000]
  let $count := xdmp:estimate(cts:search(fn:doc(),
        cts:word-query($word,("unstemmed","case-sensitive",
                              "diacritic-sensitive"),0)))
return <word text="{$word}" count="{$count}"/> }
</words>
```

This query returns one `word` element per lexicon term, along with the matching term and counts of the number of fragments that have the term, under the specified directory (`/`), starting with the term `aardvark`. Sample output from this query follows:

```
<words>
  <word text="aardvark" count="10"/>
  <word text="aardvarks" count="10"/>
  <word text="aardwolf" count="5"/>
...
</words>
```

7.0 Using Range Queries in cts:query Expressions

MarkLogic Server allows you to access range indexes in a `cts:query` expression to constrain a search by a range of values in an element or attribute. This chapter describes some details about these range queries and includes the following sections:

- Overview of Range Queries
- Range Query cts:query Constructors
- Examples of Range Queries

7.1 Overview of Range Queries

This section provides an overview of what range queries are and why you might want to use them, and includes the following sections:

- Uses for Range Queries
- Requirements for Using Range Queries
- Performance and Coding Advantages of Range Queries

7.1.1 Uses for Range Queries

Range queries are designed to constrain searches on ranges of a value. For example, if you want to find all articles that were published in 2005, and if your content has an element (or an attribute or a property) named `PUBLISHDATE` with type `xs:date`, you can create a range index on the element `PUBLISHDATE`, then specify in a search that you want all articles with a `PUBLISHDATE` greater than December 31, 2004 and less than January 1, 2006. Because that element has a range index, MarkLogic Server can resolve the query extremely efficiently.

Because you can create range indexes on a wide variety of XML datatypes, there is a lot of flexibility in the types of content with which you can use range queries to constrain searches. In general, if you need to constrain on a value, it is possible to create a range index and use range queries to express the ranges in which you want to constrain the results.

7.1.2 Requirements for Using Range Queries

Keep in mind the following requirements for using range queries in your `cts:search` operations:

- Range queries require a range index to be defined on the element or attribute in which you want to constrain the results.
- The range index must be in the same collation as the one specified in the range query.
- If no collation is specified in the range query, then the query takes on the collation of the query (for example, if a collation is specified in the XQuery prolog, that is used). For details on collation defaults, see "How Collation Defaults are Determined" on page 309.

Because range queries require range indexes, keep in mind that range indexes take up space, add to memory usage on the machine(s) in which MarkLogic Server runs, and increase loading/reindexing time. As such, they are not exactly "free", although, particularly if you have a relatively small number of them, they will not use a huge amount of resources. The amount of resources used depends a lot on the content; how many documents have the elements and/or attributes specified, how often do those elements/attributes appear in the content, how large is the content set, and so on. As with many performance improvements, there are trade-offs to analyze, and the best way to analyze the impact is to experiment and see if the cost is worth the performance improvement. For details about range indexes and procedures for creating them, see the Range Indexes and Lexicons chapter in the *Administrator's Guide*.

7.1.3 Performance and Coding Advantages of Range Queries

Most of what you can express using range queries you can also express using predicates in XPath expressions. There are two big advantages of using range queries over XPath predicates:

- Performance
- Ease of coding

Using range queries in `cts:query` expressions can produce faster performance than using XPath predicates. Range indexes are in-memory structures, and because range indexes are required for range queries, they are usually very fast. There is no requirement for the range index when specifying an XPath predicate, and it is therefore possible to specify a predicate that might need to scan a large number of fragments, which could take considerable time. Additionally, because range queries are `cts:query` objects, you can use registered queries to pre-compile them, adding more performance advantages.

There are also coding advantages to range queries over XPath predicates. Because range queries are leaf-level `cts:query` constructors, they can be combined with other constructors (including other range query constructors) to form complex expressions. It is fairly easy to write XQuery code that takes user input from a form (from drop-down lists, text boxes, radio buttons, and so on) and use that user input to generate extremely complex `cts:query` expressions. It is very difficult to do that with XPath expressions. For details on `cts:query` expressions, see "Composing cts:query Expressions" on page 121.

7.2 Range Query cts:query Constructors

The following XQuery APIs are included in the range query constructors:

- `cts:element-attribute-range-query`
- `cts:element-range-query`
- `cts:path-range-query`
- corresponding accessor functions

Each API takes QNames, the type of operator (for example, >=, <=, and so on), values, and a collation as inputs. For details of these APIs and for their signatures, see the *MarkLogic XQuery and XSLT Function Reference*.

> **Note:** For release 3.2, range queries do not contribute to the score, regardless of the weight specified in the `cts:query` constructor.

7.3 Examples of Range Queries

The following are some examples that use range query constructors.

Consider a document with a URI `/dates.xml` with the following structure:

```
<root>
  <entry>
    <date>2007-01-01</date>
    <info>Some information.</info>
  </entry>
  <entry>
    <date>2006-06-23</date>
    <info>Some other information.</info>
  </entry>
  <entry>
    <date>1971-12-23</date>
    <info>Some different information.</info>
  </entry>
</root>
```

Assume you have defined an element range index of type `xs:date` on the QName `date` (note that you must either load the document after defining the range index or complete a reindex of the database after defining the range index).

You can now issue queries using the `cts:element-range-query` constructor. The following query searches the `entry` element of the document `/dates.xml` for entries that occurred on or before January 1, 2000.

```
cts:search(doc("/dates.xml")/root/entry,
  cts:element-range-query(xs:QName("date"), "<=",
      xs:date("2000-01-01") ) )
```

This query returns the following node, because it is the only one that satisfies the range query:

```
<entry>
    <date>1971-12-23</date>
    <info>Some different information.</info>
</entry>
```

The following query uses a `cts:and-query` to combine two date ranges, dates after January 1, 2006 and dates before January 1, 2008.

```
cts:search(doc("/dates.xml")/root/entry,
  cts:and-query((
   cts:element-range-query(xs:QName("date"), ">",
      xs:date("2006-01-01") ),
   cts:element-range-query(xs:QName("date"), "<",
      xs:date("2008-01-01") ) )) )
```

This query returns the following two nodes:

```
<entry>
    <date>2007-01-01</date>
    <info>Some information.</info>
</entry>

<entry>
    <date>2006-06-23</date>
    <info>Some other information.</info>
</entry>
```

8.0 Using Aggregate Functions

This chapter describes how to use builtin aggregate functions and aggregate user-defined functions (UDFs) to analyze values in lexicons and range indexes.

This chapter contains the following sections:

- Introduction to Aggregate Functions
- Using Builtin Aggregate Functions
- Using Aggregate User-Defined Functions

8.1 Introduction to Aggregate Functions

An aggregate function performs an operation over the values in one or more range indexes. For example, computing a sum or count over an element, attribute, or field range index. Aggregate functions are best used for analytics that produce a small number of results, such as computing a single numeric value across a set of range index values.

Aggregate functions use In-Database MapReduce, which greatly improves performance because:

- Analysis is parallelized across the hosts in a cluster, as well as across the database forests on each host.
- Analysis is performed close to the data.

MarkLogic Server provides builtin aggregate functions for common mathematical and statistical operations. You can also implement your own aggregate functions, using the Aggregate UDF interface. For details, see Implementing an Aggregate User-Defined Function in the *Application Developer's Guide*.

8.2 Using Builtin Aggregate Functions

MarkLogic Server provides the following builtin aggregate functions, accessible through the XQuery, REST, and Java APIs.

XQuery Function	REST and Java Aggregate Name
`cts:avg-aggregate`	`avg`
`cts:correlation`	`correlation`
`cts:count-aggregate`	`count`
`cts:covariance`	`covariance`

XQuery Function	REST and Java Aggregate Name
`cts:covariance-p`	`covariance-population`
`cts:max`	`max`
`cts:median`	`median`
`cts:min`	`min`
`cts:stddev`	`stddev`
`cts:stddev-p`	`stddev-population`
`cts:sum-aggregate`	`sum`
`cts:variance`	`variance`
`cts:variance-p`	`variance-population`

The table below summarizes how to call an aggregate function directly using the XQuery, REST and Java APIs:

Interface	Mechanism	Example

XQuery	Call the builtin function directly from your XQuery code.	`cts:sum-aggregate(`
		`  cts:element-reference(`
		`  xs:QName("Amount")`
		`  )`
		`)`

REST	Send a GET `/version/values/{name}` request, naming the function in the `aggregates` request parameter. For details, see Analyzing Lexicons and Range Indexes With Aggregate Functions in the *REST Application Developer's Guide*.	`GET` `/v1/values/amount?options=my-index-defns&aggregate=sum`

Java	Specify the aggregate name using `ValuesDefinition.setAggregate` and pass the `ValuesDefinition` to `QueryManager.values` or `QueryManager.tuples`. For details, see *Java Application Developer's Guide*.	`QueryManager qm = ...;`
		`ValuesDefinition vdef =`
		`  qm.newValuesDefinition(...);`
		`vdef.setAggregate("sum");`
		
		`TuplesHandle t =`
		`  qm.tuples(vdef,`
		`  new TuplesHandle());`

You can also specify an aggregate function in a `<values/>` or `<tuples/>` element of query options. For example:

```
<options xmlns="http://marklogic.com/appservices/search">
  <values name="my-values">
    <aggregate apply="sum" />
    ...
  </values>
</options>
```

For more details, see “Search API: Understanding and Using” on page 16, the *Java Application Developer’s Guide*, or the *REST Application Developer’s Guide*.

8.3 Using Aggregate User-Defined Functions

You can create an aggregate user-defined function (UDF) to analyze the values in one or more range indexes. An aggregate UDF must be installed before you can use it. For information on creating and installing aggregate UDFs, see User-Defined Functions in the *Application Developer’s Guide*.

Aggregate UDFs are best for analyses that compute a small number of results over the values in one or more range indexes, rather than analyses that produce results in proportion to the number of range index values or the number of documents processed.

UDFs are identified by a relative path and a function name. The path is the path under which the plugin is installed. The path is `scope/plugin-id`, where `scope` is the scope passed to `plugin:install-from-zip` when the plugin is installed, and `plugin-id` is the ID specified in `<id/>` in the plugin manifest. For details, see Installing a Native Plugin in the *Application Developer’s Guide*.

The following example uses an aggregate UDF called “myAvg” that is provided by the plugin installed with the path `native/sampleplugin`:

```
cts:aggregate("native/sampleplugin", "myAvg", ...)
```

The table below summarizes how to invoke aggregate UDFs in XQuery, Java, and RESTful applications.

Note: You can only pass extra parameters to an aggregate UDF from XQuery.

Interface	Mechanism	Example

XQuery	Call `cts:aggregate`, supplying the path to the native plugin that implements the aggregate and the aggregate name. Pass aggregate-specific parameters through the 4th argument.	`cts:aggregate(`
		`  **"native/samplePlugin",`**
		`  **"myAvg",`**
		`  cts:element-reference(`
		`  xs:QName("Amount")),`
		`  (plugin-arg1, plugin-arg2)`
		`)`

REST	Send a GET request to the `/values/{name}` service, supplying the path to the native plugin in `aggregatePath` and the function name in `aggregate`. For details, Analyzing Lexicons and Range Indexes With Aggregate Functions in the *REST Application Developer's Guide*.	`GET`
		`/v1/values/amount?options=myoptions&`
		`aggregatePath=native/samplePlugin&aggregate=myAvg`

Java	Set the aggregate name and path on a `ValuesDefinition`, then pass the `ValuesDefinition` to `QueryManager.values` or `QueryManager.tuples`. For details, see the *Java Application Developer's Guide*.	`QueryManager qm = ...;`
		`ValuesDefinition vdef =`
		`  qm.newValuesDefinition(...);`
		`vdef.setAggregate("myAvg");`
		`vdef.setAggregatePath(`
		`  "native/samplePlugin");`
		
		`TuplesHandle t =`
		`  qm.values(vdef,`
		`  new ValuesHandle());`

You can also specify an aggregate UDF in a `<values/>` or `<tuples/>` element of query options. For example:

```
xquery version "1.0-ml";
import module namespace search =
"http://marklogic.com/appservices/search"
    at "/MarkLogic/appservices/search/search.xqy";

<options xmlns="http://marklogic.com/appservices/search">
  <values name="my-values">
    <aggregate apply="myAvg" udf="native/samplePlugin" />
    ...
  </values>
</options
```

For more details, see "Search API: Understanding and Using" on page 16, the *Java Application Developer's Guide*, or the *REST Application Developer's Guide*.

9.0 Highlighting Search Term Matches

This chapter describes ways you can use `cts:highlight` to wrap terms that match a search query with any markup. It includes the following sections:

- Overview of cts:highlight
- General Search and Replace Function
- Built-In Variables For cts:highlight
- Using cts:highlight to Create Snippets
- cts:walk Versus cts:highlight
- Common Usage Notes

For the syntax of `cts:highlight`, see the *MarkLogic XQuery and XSLT Function Reference*.

9.1 Overview of cts:highlight

When you execute a search in MarkLogic Server, it returns a set of nodes, where each node contains text that matches the search query. A common application requirement is to display the results with the matching terms highlighted, perhaps in bold or in a different color. You can satisfy these highlighting requirements with the `cts:highlight` function, which is designed with the following main goals:

- Make the task of highlighting search hits easy.
- Make queries that do text highlighting perform well.
- Make it possible to do more complex actions than simple text highlighting.

Even though it is designed to make it easy to highlight search term hits, the `cts:highlight` function is implemented as a general purpose function. The function substitutes search hits with an XQuery expression specified in the third argument. Because you can substitute the search term hits with any XQuery expression, you can perform all kinds of search and replace actions on terms that match a query. These search and replace operations will perform well, too, because `cts:highlight` is built-in to MarkLogic Server.

9.1.1 All Matching Terms, Including Stemmed, and Capitalized

When you use the standard XQuery string functions such as `fn:replace` and `fn:contains` to find matches, you must specify the exact string you want to match. If you are trying to highlight matches from a `cts:search` query, exact string matches will not find all of the hits that match the query. A `cts:highlight` query match, however, is anything that matches the `cts:query` specified as the second argument of `cts:highlight`.

If you have stemmed searches enabled, matches can be more than exact text matches. For example, `run`, `running`, and `ran` all match a query for `run`. For details on stemming, see "Understanding and Using Stemmed Searches" on page 212.

Similarly, query matches can have different capitalization than the exact word for which you actually searched. Additionally, wildcard matches (if wildcard indexes are enabled) will match a whole range of queries. Queries that use `cts:highlight` will find all of these matches and replace them with whatever the specified expression evaluates to.

9.2 General Search and Replace Function

Although it is designed to make highlighting easy, `cts:highlight` can be used for much more general search and replace operations. For example, if you wanted to replace every instance of the term `content database` with `contentbase`, you could issue a query similar to the following:

```
for $x in cts:search(//mynode, "content database")
return
cts:highlight($x, "content database", "contentbase")
```

This query happens to use the same search query in the `cts:search` as it does in the `cts:highlight`, but that is not required (although it is typical of text highlighting requirements). For example, the following query finds all of the nodes that contain the word `foo`, and then replaces the word `bar` in those nodes with the word `baz`:

```
for $x in cts:search(fn:doc(), "foo")
return
cts:highlight($x, "bar", "baz")
```

Because you can use any XQuery expression as the replace expression, you can perform some very complex search and replace operations with a relatively small amount of code.

9.3 Built-In Variables For cts:highlight

The `cts:highlight` function has three built-in variables which you can use in the replace expression. The expression is evaluated once for each query match, so each variable is bound to a sequence of query matches, and the value of the variables is the value of the query match for each iteration. This section describes the three variables and explains how to use them in the following subsections:

- Using the $cts:text Variable to Access the Matched Text
- Using the $cts:node Variable to Access the Context of the Match
- Using the $cts:queries Variable to Feed Logic Based on the Query
- Using $cts:start to Capture the String-Length Position
- Using $cts:action to Stop Highlighting

9.3.1 Using the $cts:text Variable to Access the Matched Text

The `$cts:text` variable holds the strings representing of the query match. For example, assume you have the following document with the URI `test.xml` in a database in which stemming is enabled:

```
<root>
  <p>I like to run to the market.</p>
  <p>She is running to catch the train.</p>
  <p>He runs all the time.</p>
</root>
```

You can highlight text from a query matching the word `run` as follows:

```
for $x in cts:search(doc("test.xml")/root/p, "run")
return
cts:highlight($x, "run", <b>{$cts:text}</b>)
```

The expression `<b>{$cts:text}</b>` is evaluated once for each query match, and it replaces the query match with whatever it evaluates to. Because `run`, `running`, and `ran` all match the `cts:query` for `run`, the results highlight each of those words and are as follows:

```
<p>I like to <b>run</b> to the market.</p>
<p>She is <b>running</b> to catch the train.</p>
<p>He <b>runs</b> all the time.</p>
```

9.3.2 Using the $cts:node Variable to Access the Context of the Match

The `$cts:node` variable provides access to the text node in which the match occurs. By having access to the node, you can create expressions that do things in the context of that node. For example, if you know your XML has a structure with a hierarchy of `book`, `chapter`, `section`, and `paragraph` elements, you can write code in the highlight expression to display the section in which each hit occurs. The following code snippet shows an XPath statement that returns the first element named `chapter` above the text node in which the highlighted term occurs:

```
$cts:node/ancestor::chapter[1]
```

You can then use this information to do things like add a link to display that chapter, search for some other terms within that chapter, or whatever you might need to do with the information. Once again, because `cts:highlight` evaluates an arbitrary XQuery expression for each search query hit, the variations of what you can do with it are virtually unlimited.

The following example shows how to use the `$cts:node` variable in a test to print the highlighted term in blue if its immediate parent is a `p` element, otherwise to print the highlightled term in red:

```
let $doc := <root>
                <p>This is blue.</p>
                <p><i>This is red italic.</i></p>
             </root>
return
cts:highlight($doc, cts:or-query(("blue", "red")),
 (if ( $cts:node/parent::p )
  then ( <font color="blue">{$cts:text}</font> )
  else ( <font color="red">{$cts:text}</font> ) )
            )
```

This query returns the following results:

```
<root>
 <p>This is <font color="blue">blue</font>.</p>
 <p><i>This is <font color="red">red</font>italic.</i></p>
</root>
```

9.3.3 Using the $cts:queries Variable to Feed Logic Based on the Query

The `$cts:queries` variable provides access to the `cts:query` that satisfies the query match. You can use that information to drive some logic about how you might highlight different queries in different ways.

For example, assume you have the following document with the URI `hellogoodbye.xml` in your database:

```
<root>
  <a>It starts with hello and ends with goodbye.</a>
</root>
```

You can then run the following query to use some simple logic which displays queries for `hello` in blue and queries for `goodbye` in red:

```
cts:highlight(doc("hellogoodbye.xml"),
              cts:and-query((cts:word-query("hello"),
                             cts:word-query("goodbye"))),
if ( cts:word-query-text($cts:queries) eq "hello" )
then ( <font color="blue">{$cts:text}</font> )
else ( <font color="red">{$cts:text}</font> ) )

returns:

<root>
  <a>It starts with <font color="blue">hello</font>
  and ends with <font color="red">goodbye</font>.</a>
</root>
```

9.3.4 Using $cts:start to Capture the String-Length Position

The `$cts:start` variable returns the starting position of the matching text (`$cts:text`), based on the string-length of the text node being processed (`$cts:node`).

9.3.5 Using $cts:action to Stop Highlighting

Use `xdmp:set` to change the value of `$cts:action` and specify what action should occur after processing a match. You can use this variable to control highlighting, typically based on some condition (such as how many matches have already occurred) that you have coded into your application). ou can specify for highlighting to `continue` (the default), to `skip` highlighting the remainder of the matches in the current text node, or to `break`, stopping highlighting for the rest of the input.

9.4 Using cts:highlight to Create Snippets

When you are performing searches, you often want to highlight the result of the search, showing only the part of the document in which the search match occurs. These portions of the document where the search matches are often called snippets. This section shows a simple example that describes the basic design pattern for using `cts:highlight` to create snippets. The example shown here is trivial in that it only prints out the parent element for the search hit, but it shows the pattern you can use to create useful snippets. A typical snippet might show the matched results in bold and show a few words before and after the results.

The basic design pattern to create snippets is to first run a `cts:search` to find your results, then, for search each match, run `cts:highlight` on the match to mark it up. Finally, you run the highlighted match through a recursive transformation or through some other processing to write out the portion of the document you are interested in. For details about recursive transformations, see Transforming XML Structures With a Recursive typeswitch Expression in the *Application Developer's Guide*.

The following example creates a very simple snippet for a search in the Shakespeare database. It simply returns the parent element for the text node in which the search matches. It uses `cts:highlight` to create a temporary element (named `HIGHLIGHTME`) around the element containing the search match, and then uses that temporary element name to find the matching element in the transformation.

```
xquery version "1.0-ml";
declare function local:truncate($x as item()) as item()*
{
  typeswitch ($x)
  case element(HIGHLIGHTME) return $x/node()
  case element(TITLE) return if ($x/../../PLAY) then $x else ()
  default return for $z in $x/node() return local:truncate($z)
};

let $query := "to be or not to be"
```

```
for $x in cts:search(doc(), $query)
return
local:truncate(cts:highlight($x, $query,
  <HIGHLIGHTME>{$cts:node/parent::element()}</HIGHLIGHTME>))
(:
   returns:
   <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
   <LINE>To be, or not to be: that is the question:</LINE>
:)
```

This example simply returns the elements in which the match occurs (in this case, only one element matches the query) and the `TITLE` element that contains the title of the play. You can add any logic you want to create a snippet that is right for your application. For example, you might want to also print out the name of the act and the scene title for each search result, or you might want to calculate the line number for each result. Because you have the whole document available to you in the transformation, it is easy to do many interesting things with the content.

Note: The use of a recursive typeswitch makes sense assuming you are doing something interesting with various parts of the node returned from the search (for example, printing out the play title, act number, and scene name). If you only want to return the element in which the search match occurs, you can do something simpler. For example, you can use XPath on the highlighted expression to simplify this design pattern as follows:

```
let $query := "to be or not to be"
for $x in cts:search(doc(), $query)
return
cts:highlight($x, $query, <HIGHLIGHTME>{
    $cts:node/parent::element()}</HIGHLIGHTME>)//HIGHLIGHTME/node()
```

9.5 cts:walk Versus cts:highlight

The function `cts:walk` is similar to `cts:highlight`, but instead of returning a copy of the node passed in with the specified changes, it returns only the expression evaluations for the text node matches specified in the `cts:walk` call. Because `cts:walk` does not construct a copy of the node, it is faster than `cts:highlight`. In cases where you only need to return the expression evaluations, `cts:walk` will be more efficient than `cts:highlight`.

9.6 Common Usage Notes

This section shows some common usage patterns to be aware with when using `cts:highlight`. The following topics are included:

- Input Must Be a Single Node
- Using xdmp:set Side Effects With cts:highlight
- No Highlighting with cts:similar-query or cts:element-attribute-*-query

9.6.1 Input Must Be a Single Node

The input to `cts:highlight` must be a single node. That means that if you want to highlight query hits from a `cts:search` operation that returns multiple nodes, you must bind the results of the `cts:search` to a variable (in a `for` loop, for example), as in the following example:

```
for $x in cts:search(fn:doc(), "MarkLogic")
return
cts:highlight($x, "MarkLogic", <b>{$cts:text}</b>)
```

This query returns all of the documents in the database that contain `MarkLogic`, with `b` tags surrounding each query match.

> **Note:** The input node to `cts:highlight` must be a document node or an element node; it cannot be a text node.

9.6.2 Using xdmp:set Side Effects With cts:highlight

If you want to keep the state of the highlighted terms so you can handle some instances differently than others, you can define a variable and then use the `xdmp:set` function to change the value of the variable as the highlighted terms are processed. Some common uses for this functionality are:

- Highlight only the first instance of a term.
- Highlight the first term in a different color then the rest of the terms.
- Keep a count on the number of terms matching the query.

The ability to change the state (also known as side effects) opens the door for infinite possibilities of what to do with matching terms.

The following example shows a query that highlights the first query match with a bold tag and returns only the matching text for the rest of the matches.

Assume you have following document with the URI `/docs/test.xml` in your database:

```
<html>
  <p>hello hello hello hello</p>
</html>
```

You can then run the following query to highlight just the first match:

```
let $count := 0
return
  cts:highlight(doc("/docs/test.xml"), "hello",
   (: Increment the count for each query match :)
    (xdmp:set($count, $count + 1 ),
     if ( $count = 1 )
     then ( <b>{$cts:text}</b> )
     else ( $cts:text ) )
```

```
        )

Returns:

<html>
  <p><b>hello</b> hello hello hello</p>
</html>
```

Because the expression is evaluated once for each query match, the `xdmp:set` call changes the state for each query match, having the side effect of the conditions being evaluated differently for each query match.

9.6.3 No Highlighting with cts:similar-query or cts:element-attribute-*-query

You cannot use `cts:highlight` to highlight results from queries containing `cts:similar-query` or any of the `cts:element-attribute-*-query` functions. Using `cts:highlight` with these queries will return the nodes without any highlighting.

10.0 Geospatial Search Applications

This chapter describes how to use the geospatial functions and describes the type of applications that might use these functions, and includes the following sections:

- Overview of Geospatial Data in MarkLogic Server
- Understanding Geospatial Coordinates and Regions
- Geospatial Indexes
- Using the API
- Simple Geospatial Search Example

10.1 Overview of Geospatial Data in MarkLogic Server

In its most basic form, geospatial data is a set of latitude and longitude coordinates. Geospatial data in MarkLogic Server is marked up in XML elements and/or attributes. There are a variety of ways you can represent geospatial data as XML, and MarkLogic Server supports several different representations. This section provides an overview of how geospatial data and queries work in MarkLogic Server, and includes the following parts:

- Terminology
- Coordinate System
- Types of Geospatial Queries
- XQuery Primitive Types And Constructors for Geospatial Queries
- Well-Known Text (WKT) Markup Language

10.1.1 Terminology

The following terms are used to describe the geospatial features in MarkLogic Server:

- coordinate system

  A geospatial *coordinate system* is a set of mappings that map places on Earth to a set of numbers. The vertical axis is represented by a latitude coordinate, and the horizontal axis is represented by a longitude coordinate, and together they make up a coordinate system that is used to map places on the Earth. For more details, see “Latitude and Longitude Coordinates in MarkLogic Server” on page 179.

- point

  A geospatial *point* is the spot in the geospatial coordinate system representing the intersection of a given latitude and longitude. For more details, see “Points in MarkLogic Server” on page 179.

- proximity

  The *proximity* of search results is how close the results are to each other in a document. Proximity can apply to any type of search terms, included geospatial search terms. For example, you might want to find a search term *dog* that occurs within 10 words of a point in a given zip code.

- distance

  The *distance* between two geospatial objects refers to the geographical closeness of those geospatial objects.

10.1.2 Coordinate System

MarkLogic Server supports two types of coordinate systems for geospatial data:

- WGS84
- Raw

By default, MarkLogic Server uses the World Geodetic System (WGS84) as the basis for geocoding. WGS84 sets out a coordinate system that assumes a single map projection of the earth. WGS84 is widely used for mapping locations on the earth, and is used by a wide range of services, including many satellite services (notably: Global Positioning System—GPS) and Google Maps. There are other geocoding systems, some of which have advantages or disadvantages over WGS84 (for example, some are more accurate in a given region, some are less popular); MarkLogic Server uses WGS84, which is a widely accepted standard for global point representation. For details on WGS84, see http://en.wikipedia.org/wiki/World_Geodetic_System.

You can use the raw coordinate system when you want your points mapped onto a flat plane instead of onto the geometry of the earth.

10.1.3 Types of Geospatial Queries

The following types of geospatial queries are supported in MarkLogic Server:

- point query—matches a single point
- box query—any point within a rectangular box
- radius query—any point within a specified distance around a point
- polygon query—any point within a specified *n*-sided polygon

Geospatial `cts:query` constructors are composable just like any other `cts:query` constructors. For details on composing `cts:query` constructors, see “Composing cts:query Expressions” on page 121.

In addition to geospatial query constructors, there are built-in functions to perform operations (such as calculating distance) on geospatial data, as enumerated in “Geospatial Operations” on page 188.

Note: Using the geospatial query constructors requires a valid geospatial license key; without a valid license key, searches that include geospatial queries will throw an exception.

10.1.4 XQuery Primitive Types And Constructors for Geospatial Queries

To support geospatial queries, MarkLogic Server has the following XQuery primitive types:

- `cts:box`
- `cts:circle`
- `cts:complex-polygon`
- `cts:linestring`
- `cts:point`
- `cts:polygon`

You use these primitive types in geospatial `cts:query` constructors (for example, `cts:element-geospatial-query`, `cts:element-attribute-pair-geospatial-query`, `cts:element-pair-geospatial-query`, and so on.). Each of the `cts:box`, `cts:circle`, `cts:complex-polygon`, `cts:linestring`, `cts:point`, and `cts:polygon` XQuery primitive types is an instance of the `cts:region` base type. These types define regions, and then the query returns true if the regions contain matching data in the context of a search.

Additionally, there are constructors for each primitive type which attempt to take data and construct it into the corresponding type. If the data is not constructible, then an exception is thrown. MarkLogic Server parses the data to try and extract points to construct into the type. For example, the following constructs the string into a `cts:polygon` which includes the points separated by a space:

```
cts:polygon("38,-10 40,-10 39, -15")
```

The following constructs these coordinates (represented as numbers) into a `cts:point`:

```
cts:point(38.7, -10.3)
```

10.1.5 Well-Known Text (WKT) Markup Language

MarkLogic supports well-know text (WKT) markup language for representing geospatial data. You can use the following WKT objects in MarkLogic: `POINT`, `POLYGON`, `LINESTRING`, `TRIANGLE`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`, and `GEOMETRYCOLLECTION`. To use WKT in a geospatial query, you use the `cts:parse-wkt` function to convert WKT into a sequence of `cts:region` items, and then you can use the `cts:region` items in any of the geospatial `cts:query` constructors.

For example, the following code returns a `cts:complex-polygon` that has an outer boundary and an inner boundary:

```
cts:parse-wkt("
 POLYGON(
  (0 0, 0 10, 10 10, 10 0, 0 0),
  (0 5, 0 7, 5 7, 5 5, 0 5) )" )
```

The following table shows how the WKT types map to the MarkLogic geospatial types.

WKT Type	MarkLogic Geopsatial Type
POINT	cts:point
POINT EMPTY	cts:point (flagged as empty)
POLYGON	cts:complex-polygon | cts:polygon
POLYGON EMPTY	cts:complex-polygon (flagged as empty)
LINESTRING	cts:linestring
LINESTRING EMPTY	cts:linestring (flagged as empty)
TRIANGLE	cts:polygon
TRIANGLE EMPTY	cts:complex-polygon (flagged as empty)
MULTIPOINT	cts:point*
MULTIPOINT EMPTY	()
MULTILINESTRING	cts:linestring*
MULTILINESTRING EMPTY	()
MULTIPOLYGON	(cts:polygon | cts:complex-polygon)*
MULTIPOLYGON EMPTY	()
GEOMETRYCOLLECTION	cts:region*
GEOMETRYCOLLECTION EMPTY	()
others	throws XDMP-BADWKT

Similarly, you can convert a `cts:region` to WKT with the `cts:to-wkt` function. For example, the following returns a WKT `POINT`:

```
cts:to-wkt(cts:point(1, 2))
```

You cannot convert a `cts:circle` or a `cts:box` to WKT. For more details on WKT, see http://en.wikipedia.org/wiki/Well-known_text.

10.2 Understanding Geospatial Coordinates and Regions

This section describes the rules for geospatial coordinates and the various regions (`cts:box`, `cts:circle`, `cts:complex-polygon`, `cts:linestring`, `cts:point`, and `cts:polygon`), and includes the following parts:

- Understanding the Basics of Coordinates and Points
- Understanding Geospatial Boxes
- Understanding Geospatial Polygons: Polygons, Complex Polygons, and Linestrings
- Understanding Geospatial Circles
- Linestrings

10.2.1 Understanding the Basics of Coordinates and Points

To understand how geospatial regions are defined in MarkLogic Server, you should first understand the basics of coordinates and of points. This section describes the following:

- Latitude and Longitude Coordinates in MarkLogic Server
- Points in MarkLogic Server

10.2.1.1 Latitude and Longitude Coordinates in MarkLogic Server

Latitudes have north/south coordinates. They start at 0 degrees for the equator and head north to 90 degrees for the north pole and south to -90 degrees for the south pole. If you specify a coordinate that is greater than 90 degrees or less than -90 degrees, the value is truncated to either 90 or -90, respectively.

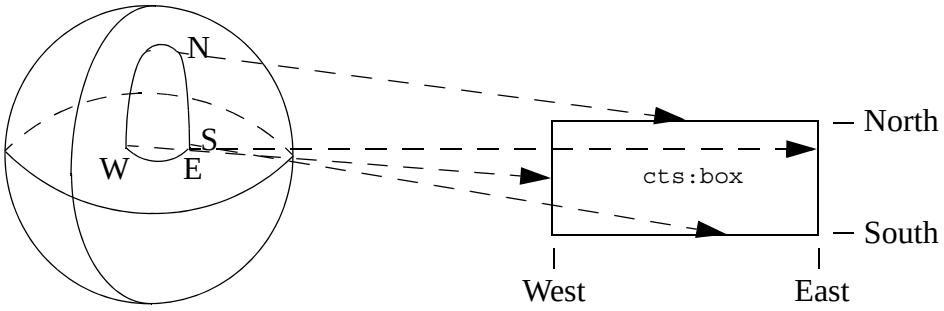
Longitudes have east/west coordinates. They start at 0 degrees at the Prime Meridian and head east around the Earth to 180 degrees, and head west around the earth (from the Prime Meridian) to -180 degrees. If you travel 360 degrees, it brings you back to the Prime Meridian. If you go west from the Prime Meridian, the numbers go negative. For example, New York City is west of the Prime Meridian, and its longitude is -73.99 degrees. Adding or subtracting any multiple of 360 to a longitude coordinate gives an equivalent coordinate.

10.2.1.2 Points in MarkLogic Server

A point is simply a pair of latitude and longitude coordinates. Where the points intersect is a place on the Earth. For example, the coordinates of San Francisco, California are a the pair that includes the latitude of 37.655983 and the longitude of -122.425525. The `cts:point` type is used to define a point in MarkLogic Server. Use the `cts:point` constructor to construct a point from a set of coordinates. Additionally, points are used to define the other regions in MarkLogic Server (`cts:box`, `cts:polygon`, and `cts:circle`).

10.2.2 Understanding Geospatial Boxes

Geospatial boxes allow you to make a region defined by four coordinates. The four coordinates define a *geospatial box* which, when projected onto a flat plane, forms a rectangular box. A point is said to be in that geospatial box if it is inside the boundaries of the box. The four coordinates that define a box represent the southern, western, northern, and eastern boundaries of the box. The box is two-dimensional, and is created by taking a projection from the three-dimensional Earth onto a flat surface. On the surface of the Earth, the edges of the box are arcs, but when those arcs are projected into a plane, they become two-dimensional latitude and longitude lines, and the space defined by those lines forms a rectangle (represented by a `cts:box`), as shown in the following figure.



Projecting coordinates from the curved earth into a flat box

The following are the assumptions and restrictions associated with geospatial boxes:

- The four points on a box are south, west, north, and east, in that order.
- Assuming a projection from the Earth onto a two-dimensional plane, boxes are determined by going from the south western limit to south eastern limit (even if it passes the date line), then north to the north eastern limit (border on the poles), then west to the north western limit, then back south to the south western limit where you started.
- When determining the west/east boundary of the box, you always start at the western longitude and head east toward the eastern longitude. This means that if your western point is east of the date line, and your eastern point is west of the date line, then you will head east around the Earth until you get back to the eastern point.
- Similarly, when determining the south/north sides of the box, you always start at the southern latitude and head north to the northern latitude. You cannot cross the pole, however, as it does not make sense to have the northern point south of the southern point. If you do cross a pole, a search that uses that box will throw an `XDMP-BADBOX` runtime error (because you cannot go north from the north pole). Note that the error will happen at search time, not at box creation time.

- If the eastern coordinate is equal to the western coordinate, then only that longitude is considered. Similarly, if the northern coordinate is equal to the southern coordinate, only that latitude is considered. The consequence of these facts are the following:
    - If the western and eastern coordinates are the same, the box is a vertical line between the southern and northern coordinates passing through that longitude coordinate.
    - If the southern and northern coordinates are the same, the box is a horizontal line between the western and eastern coordinates passing through that latitude coordinate.
    - If the western and eastern coordinates are the same, and if the southern and northern coordinates are the same, then the box is a point specified by those coordinates.
- The boundaries on the box are either in or out of the box, depending on query options (there are various boundary options on the geospatial `cts:query` constructors to control this behavior).

10.2.3 Understanding Geospatial Polygons: Polygons, Complex Polygons, and Linestrings

Geospatial polygons allow you to make a region with n-sided boundaries for your geospatial queries. These boundaries can represent any area on Earth (with the exceptions described below). For example, you might create a polygon to represent a country or a geographical region. There are three ways to construct these types of geospatial regions in MarkLogic: polygons, complex polygons, and linestrings. This section describes some of the charateristics of polygons, and includes the following parts:

- Overview of Polygons
- Polygons
- Complex Polygons
- Linestrings

10.2.3.1 Overview of Polygons

Polygons offer a large degree of flexibility compared with circles or boxes. In exchange for the flexibility, geospatial polygons are not quite as fast and not quite as accurate as geospatial boxes. The efficiency of the polygons is proportional to the number of sides to the polygon. For example, a typical 10-sided polygon will likely perform faster than a typical 1000-sided polygon. The speed is dependent on many factors, including where the polygon is, the nature of your geospatial data, and so on.

The following are the assumptions and restrictions associated with geospatial polygons:

- Assumes the Earth is a sphere, divided by great circle arcs running through the center of the earth, one great circle divided the longitude (running through the Greenwich Meridian, sometimes called the Prime Meridian) and the other dividing the latitude (at the equator).
- Each side of the polygons are semi-spherical projections from the endpoints onto the spherical Earth surface. Therefore, the lines are not all in a single plane, but instead follow the curve of the Earth (approximated to be a sphere).
- A polygon cannot include both poles. Therefore, it cannot have both poles as a boundary (regardless of whether the boundaries are included), which means it cannot encompass the full 180 degrees of latitude.
- A polygon edge must be less than 180 degrees; that is, two adjacent points of a polygon must wrap around less than half of the earth's longitude or latitude. If you need a polygon to wrap around more than 180 degrees, you can still do it, but you must use more than two points. Therefore, adjacent vertices cannot be separated by more than 180 degrees of longitude. As a result, a polygon cannot include the pole, except along one of its edges. Also as a result, if two points that make up a polygon edge are greater than 180 degrees apart, MarkLogic Server will always choose the direction that is less than 180 degrees.
- Geospatial queries are constrained to elements and attributes named in the `cts:query` constructors. To cross multiple formats in a single query, use `cts:or-query`.
- Some searches will throw a runtime exception if a polygon is not valid for the coordinate system (the coordinate system is specified at search time, not at `cts:polygon` creation time).
- The boundaries on the polygon are either in or out of the polygon, depending on query options (there are various boundary options on the geospatial `cts:query` constructors to control this behavior).
- Because of the spherical Earth assumption, and because points are represented by floats, results are not exact; polygons are not as accurate as the other methods because they use a sphere as a model of the Earth. While it may not be that intuitive, floats are used to represent points on the Earth because it turns out that there is no benefit in the accuracy if you use doubles (the Earth is just not that big).

10.2.3.2 Polygons

You can construct a `cts:polygon` by specifying the points that make up the vertices of the polygon. All points that are bounded by the resulting region are defined to be contained within the region.

10.2.3.3 Complex Polygons

You can construct a `cts:complex-polygon` by constructing a polygon within zero or more other polygons, and the resulting complex polygon is the part within the outer polygon but not within the inner polygon(s). You can also cast a `cts:complex-polygon` with no holes (that is, with no inner polygons) to a `cts:polygon`. If you specify multiple inner polygons, none of them should overlap each other.

10.2.3.4 Linestrings

A linestring is a sequence of connected joined arcs that do not necessarily form a closed loop the way a polygon forms a closed loop (although it is permissible for a linestring to form a closed loop). The "lines" are actually arcs because they are projected onto the earth's surface. A linestring supports equality and inequality: two linestrings are equal if all of their verticies are equal (or if they are both empty). It is possible to cast a `cts:linestring` to a `cts:polygon`, which results in a "flat" polygon that traces the same set of linestrings back to close the polygon.

10.2.4 Understanding Geospatial Circles

Geospatial circles allow you to define a region with boundaries defined by a point with a radius specified in miles. The point and radius define a circle, and anything inside the circle is within the boundaries of the `cts:circle`, and the boundaries of the circle are either in or out, depending on query options (there are various boundary options on the geospatial `cts:query` constructors to control this behavior).

10.3 Geospatial Indexes

Because you store geospatial data in XML markup within a document, you can query the content constraining on the geospatial XML markup. You can create geospatial indexes to speed up geospatial queries and to enable geospatial lexicon queries, allowing you to take full advantage of having the geospatial data marked up in your content. This section describes the different kinds of geospatial indexes and includes the following parts:

- Different Kinds of Geospatial Indexes
- Geospatial Index Positions
- Geospatial Lexicons

10.3.1 Different Kinds of Geospatial Indexes

Use the Admin Interface to create any of these indexes, under Database > *database_name* > Geospatial Indexes. The following sections describe how the geospatial data is structured for each of the four types of geospatial indexes, and also describes the geospatial positions option, which is available for each index.

- Geospatial Element Indexes
- Geospatial Element Child Indexes
- Geospatial Element Pair Indexes
- Geospatial Attribute Pair Indexes
- Geospatial Path Range Indexes

Note: You must have a valid geospatial license key to create or use any geospatial indexes.

10.3.1.1 Geospatial Element Indexes

With a geospatial element index, the geospatial data is represented by whitespace or punctuation (except `+`, `-`, or `.`) separated element content:

```
<element-name>37.52  -122.25</element-name>
```

For `point` format, the first entry represents the latitude coordinate, and the second entry represents the longitude coordinate. For `long-lat-point` format, the first entry represents the longitude coordinate and the second entry represents the latitude coordinate. You can also have other entries, but they are ignored (for example, KML has an additional altitude coordinate, which can be present but is ignored).

10.3.1.2 Geospatial Element Child Indexes

With a geospatial element child index, the geospatial data comes from whitespace or punctuation (except `+`, `-`, or `.`) separated element content, but only for elements that are a specific child of a specific element.

```
<element-name1>
  <element-name2>37.52  -122.25</element-name2>
</element-name1>
```

For `point` format, the first entry represents the latitude coordinate, and the second entry represents the longitude coordinate. For `long-lat-point` format, the first entry represents the longitude coordinate and the second entry represents the latitude coordinate.

10.3.1.3 Geospatial Element Pair Indexes

With a geospatial element pair index, the geospatial data comes from a specific pair of elements that are a child of another specific element.

```
<element-name>
  <latitude>37.52</latitude>
  <longitude>-122.25</longitude>
</element-name1>
```

10.3.1.4 Geospatial Attribute Pair Indexes

With a geospatial attribute pair index, the geospatial data comes from a pair of specific attributes of a specific element.

```
<element-name latitude="37.52" longitude="-122.25"/>
```

10.3.1.5 Geospatial Path Range Indexes

With a geospatial path range index, the geospatial data is expressed in the same manner as a geospatial element index and the element or attribute index is defined by a path expression.

For example, the data:

```
<a:data>
  <a:geo>37.52  -122.25</a:geo>
</a:data>
```

is indexed using the following path expression:

```
/a:data/a:geo
```

You might also express the geospatial data as an attribute. For example:

```
<a:data>
  <a:geo data="37.52  -122.25"/>
</a:data>
```

is indexed using the following path expression:

```
/a:data/a:geo/@data
```

Note: Once you have created a geospatial path range index using the Admin Interface, you cannot change the path expression. Instead, you must remove the existing geospatial path range index and create a new one with the updated path expression.

10.3.2 Geospatial Index Positions

Each geospatial index has a positions option. The positions option speeds up queries that constrain the distance in the document between geospatial data in the document (using `cts:near-query`, for example). Additionally, when element positions are enabled in the database, it improves index resolution (more accurate estimates) for element queries that involve geospatial queries (with a geospatial index with positions enabled for the geospatial data).

10.3.3 Geospatial Lexicons

Geospatial indexes enable geospatial lexicon lookups. The lexicon lookups enable very fast retrieval of geospatial values. For details on geospatial lexicons, see “Geospatial Lexicons” on page 153.

10.4 Using the API

This section provides an overview of the Geospatial API, and includes the folloiwng parts:

- Basic Procedure for Performing a Geospatial Query
- Geospatial cts:query Constructors
- Geospatial Value Constructors for Regions
- Geospatial Format Conversion Functions
- Geospatial Operations

10.4.1 Basic Procedure for Performing a Geospatial Query

Using the geospatial API is just like using any `cts:query` constructors, where you use the `cts:query` as the second parameter (or a building block of the second parameter) of `cts:search`. The basic procedure involves the following steps:

1. Load geospatial data into a database.
2. Create geospatial indexes (optional, speeds performance).
3. Construct primitive types to use in geospatial `cts:query` constructors.
4. Construct the geospatial queries using the geospatial primitive types.
5. Use the geospatial queries in a `cts:search` operation.

You can also use the geospatial queries in `cts:contains` operations, regardless of whether the geospatial data is in the database.

10.4.2 Geospatial cts:query Constructors

The following geospatial `cts:query` constructors are available:

- `cts:element-attribute-pair-geospatial-query`
- `cts:element-pair-geospatial-query`
- `cts:element-geospatial-query`
- `cts:element-child-geospatial-query`

10.4.3 Geospatial Value Constructors for Regions

The following APIs are used to construct geospatial regions. Use these functions with the geospatial `cts:query` constructors above to construct `cts:queries`.

- `cts:box`
- `cts:circle`
- `cts:complex-polygon`
- `cts:linestring`
- `cts:point`
- `cts:polygon`

For details on these functions, see the *MarkLogic XQuery and XSLT Function Reference*. These functions are complementary to the type constructors with the same names, which are described in “XQuery Primitive Types And Constructors for Geospatial Queries” on page 177.

10.4.4 Geospatial Format Conversion Functions

There are XQuery library modules to translate Metacarta, GML, KML, and GeoRSS formats to `cts:box`, `cts:circle`, `cts:point`, and `cts:polygon` formats. The functions in these libraries are designed to take geospatial data in these formats and construct `cts:region` primitive types to pass into the geospatial `cts:query` constructors and construct appropriate queries. For the signatures of these functions, see the XQuery Library Module section of the *MarkLogic XQuery and XSLT Function Reference*.

10.4.5 Geospatial Operations

The following APIs are used to perform various operations and calculations on geospatial data:

- `cts:box-intersects`
- `cts:circle-intersects`
- `cts:polygon-intersects`
- `cts:complex-polygon-intersects`
- `cts:polygon-contains`
- `cts:complex-polygon-contains`
- `cts:distance`
- `cts:shortest-distance`
- `cts:destination`

For their signatures and for more details on these functions, see the XQuery Library Module section of the *MarkLogic XQuery and XSLT Function Reference*.

10.5 Simple Geospatial Search Example

This section provides an example showing a `cts:search` that uses a geospatial query.

Assume a document with the URI `/geo/zip_labels.xml` with the following form:

```
<labels>
  <label id="1">96044</label>
  ...
  <label id="589">95616</label>
  <label id="712">95616</label>
  <label id="715">95616</label>
  ...
</labels>
```

Assume you have polygon data in a document with the URI `/geo/zip.xml` with the following form:

```
<polygon id="712">
       0.383337584506173E+02,       -0.121659014798844E+03
       0.383133840000000E+02,       -0.121656011000000E+03
       0.383135090000000E+02,       -0.121666647000000E+03
       0.383135090000000E+02,       -0.121666647000000E+03
       0.383135120000000E+02,       -0.121666875000000E+03
       0.383349030000000E+02,       -0.121667035000000E+03
       0.383353510000000E+02,       -0.121657355000000E+03
       0.383496550000000E+02,       -0.121656811000000E+03
       0.383495590000000E+02,       -0.121646955000000E+03
       0.383494950000000E+02,       -0.121645323000000E+03
       0.383473190000000E+02,       -0.121645691000000E+03
       0.383370790000000E+02,       -0.121650187000000E+03
       0.383133840000000E+02,       -0.121656011000000E+03
</polygon>
```

You can then take the contents of the polygon element and cast it to a `cts:polygon` using the `cts:polygon` constructor. For example, the following returns a `cts:polygon` for the above data:

```
cts:polygon(fn:data(fn:doc("/geo/zip.xml")//polygon[@id eq "712"]))
```

Further assume you have content of the following form:

```
<feature id="1703188" class="School">
  <name>Ralph Waldo Emerson Junior High School</name>
  <state id="06">CA</state>
  <county id="113">Yolo</county>
  <lat dms="383306N">38.5515731</lat>
  <long dms="1214639W">-121.7774624</long>
  <elevation>17</elevation>
  <map>Merritt</map>
</feature>
```

Now consider the following XQuery:

```
let $searchterms := ("school", "junior")
let $zip := "95616"
let $ziplabel :=
fn:doc("/geo/zip_labels.xml")//label[contains(.,$zip)]
let $polygons :=
   for $p in fn:doc("/geo/zip.xml")//polygon[@id=$ziplabel/@id]
   return cts:polygon(fn:data($p))
let $query :=
   cts:and-query((
       for $term in $searchterms return cts:word-query($term),
       cts:element-pair-geospatial-query(xs:QName("feature"),
             xs:QName("lat"), xs:QName("long"), $polygons) ))
return  (
<h2>{fn:concat("Places with the term '",
               fn:string-join($searchterms, "' and the term '"),
               "' in the zipcode ", fn:data($zip), ":")}</h2>,
  <ol>{for $feature in cts:search(//feature, $query)
  order by $feature/name
  return (
  <li><h3>{fn:data($feature/name)," "}
  <code>({fn:data($feature/lat)},{fn:data($feature/long)})</code></h3>
  <p>{fn:data($feature/@class)} in {fn:data($feature/county)},
  {fn:data($feature/state)} from map {fn:data($feature/map)}</p></li> )
  }</ol> )
```

This returns results similar to the following (shown rendered in a browser):

Places with the term 'school' and the term 'junior' in the zipcode 95616:

1. **Emerson Junior High School** (38.5476843,-121.7452392)

   School in Yolo, CA from map Davis

2. **Oliver Wendell Holmes Junior High School** (38.556573,-121.7363502)

   School in Yolo, CA from map Davis

3. **Ralph Waldo Emerson Junior High School** (38.5515731,-121.7774624)

   School in Yolo, CA from map Merritt

11.0 Marking Up Documents With Entity Enrichment

This chapter describes how to use entity enrichment in MarkLogic Server to add XML markup for entities such as people and places around text. It contains the following sections:

- Overview of Entity Enrichment
- Built-In Entity Enrichment
- Entity Enrichment Pipelines

11.1 Overview of Entity Enrichment

With XML, you can add element tags around text. If you add element tags around text that has a particular meaning, then you can then search for those tags to find occurrences of that meaningful thing. Common things to mark up with element tags are people and places, but there are many more things that are useful to mark up. Many industries have domain-specific things that are meaningful to mark up. For example, medical researchers might find it useful to mark up perscription drugs with a tag such as `RX`. The class of things to mark up are sometimes called *entities*, and the process of making XML more useful by finding these entities and then marking it up with meaningful tags (and searchable) is called *entity enrichment*.

MarkLogic Server includes built-in entity capabilities (licensed separately), and is also capable of integrating with third-party entity enrichment services. The Content Processing Framework makes it easy to call out to third-party tools, and there are some samples included to demonstrate this process, as described in “Sample Pipelines Using Third-Party Technologies” on page 195.

11.2 Built-In Entity Enrichment

The built-in entity enrichment in MarkLogic Server requires an entity enrichment licence key and requires a separate set of libraries to install. This section describes the installation procedure and provides some details on the core entity enrichment capabilites. This section includes the following parts:

- Installing the Entity Enrichment Libraries
- Details of MarkLogic Entity Enrichment

11.2.1 Installing the Entity Enrichment Libraries

Before you can use entity enrichment, you must first install the MarkLogicBasis package for your platform as follows:

1. Ensure that you have a license key that includes entity enrichment. If you navigate to the Host Status page in the Admin Interface (Hosts > hostname > Status), it should list `entity enrichment` under Options. If you do not have the proper license, contact MarkLogic Support or your account manager.

2. Obtain the entity enrichment download from MarkLogic Support and save the file on the machine in which MarkLogic runs.

3. Install entity enrichment as follows:

Platform	Perform the following.
Windows (64-bit)	1. Locate the `MarkLogicBasis-6.0-1-x86_64.msi` file you downloaded and double-click it to run the installer. 2. Accept the installer defaults. 3. Restart MarkLogic Server (for example, from the Admin Interface).

Linux (64-bit)	1. Navigate to the directory in which you saved the MarkLogicBasis installation file. For example:
	`% cd /space/tmp`
	2. As the root user, run the folowing:
	`% rpm -i MarkLogicBasis-6.0-1-x86_64.rpm`
	3. Restart MarkLogic Server (for example, from the Admin Interface or by running `/etc/init.d/Marklogic restart` as the root user).

Solaris (64-bit SPARC and x64)	1. Download the package to `/var/spool/pkg`.
	2. Unpack the compressed tar file in `/var/spool/pkg` with the following shell commands:
	`% cd /var/spool/pkg`
	`% uncompress MARKlogicBasis-6.0-1-sparc.tar.Z`
	`% tar xf MARKlogicBasis-6.0-1-sparc.tar`
	`% rm MARKlogicBasis-6.0-1-sparc.tar`
	Note: The package name will end in `amd64.tar.Z` for x64 (64-bit AMD Opteron and Intel 64-bit EM64T). If you are installing a release other than 6.0-1, the package name will match the latest service release.
	3. As the `root` user, install the package with the following command:
	`# pkgadd MARKlogicBasis`

Mac OS X	Not available.

The entity enrichment package is installed in the `<marklogic-dir>/MarkLogicBasis` directory.

When MarkLogic Server restarts, you will see messages in the log file indicating that the entity enrichment libraries are loaded. The log messages are similar to the following:

```
2011-09-22 19:29:43.147 Info: Checking for Basis at
/opt/MarkLogicBasis/rlp/lib/amd64-glibc25-gcc41/
2011-09-22 19:29:43.147 Info: Checking for Basis plugin at
lib/BasisEntityExtractor.so
2011-09-22 19:29:43.160 Info: Loaded Basis entity extraction
```

11.2.2 Details of MarkLogic Entity Enrichment

You can enrich documents with entity markup in MarkLogic Server using the following APIs:

- `cts:entity-highlight`
- `entity:enrich`

The `cts:entity-highlight` API calls out to the built-in entity enrichment capabilities in MarkLogic Server (entity enrichment license key required for each language). The `cts:entity-highlight` API is similar to `cts:highlight` (see "Highlighting Search Term Matches" on page 167), except it finds entities and allows you to replace their content with other content. The `entity:enrich` API is implemented in an XQuery library module, and it uses `cts:entity-highlight` to mark up entities according to the `<marklogic-dir>/Config/entity.xsd` schema.

> **Note:** A valid entity enrichment license key is required to use `cts:entity-highlight`; without a valid license key, it throws an exception. If you have a valid license for entity enrichment, you can entity enrich text in English and in any other languages for which you have a valid license key. For languages in which you do not have a valid license key, `cts:entity-highlight` finds no entities for text in that language. Additionally, you must have the entity enrichment package installed as described in "Installing the Entity Enrichment Libraries" on page 191.

The entity enrichment finds entities for a wide variety of things, including people and places. For a description of all of the entity types, see the cts:entity-highlight documentation in *MarkLogic XQuery and XSLT Function Reference*. The following shows a simple example of using the built-in entity enrichment:

```
xquery version "1.0-ml";

import module namespace entity="http://marklogic.com/entity"
    at "/MarkLogic/entity.xqy";

let $myxml := <node>George Washington never visited Norway.</node>
return
entity:enrich($myxml)
```

This returns the following:

```
<node>
  <entity:person xmlns:entity="http://marklogic.com/entity">George
     Washington</entity:person> never visited
  <entity:gpe xmlns:entity="http://marklogic.com/entity">Norway.
  </entity:gpe>
</node>
```

11.3 Entity Enrichment Pipelines

MarkLogic Server includes Content Processing Framework (CPF) applications to perform entity enrichment on your XML. You can use the built-in capabilities, you can use the CPF applications for third-party entity extraction technologies, or you can create custom applications with your own technology or some other third-party technology. This section includes the following parts:

- MarkLogic Server Entity Enrichment Pipeline
- Sample Pipelines Using Third-Party Technologies
- Custom Entity Enrichment Pipelines

These CPF applications require you to install content processing on your database. For details on CPF, including information about domains and pipelines, see the *Content Processing Framework Guide* guide.

11.3.1 MarkLogic Server Entity Enrichment Pipeline

The MarkLogic Server entity enrichment pipeline uses the built-in entity enrichment capabilites in MarkLogic Server to add XML tags to text. When you set up the entity enrichment pipeline, all documents added to the domain and all documents modified in the domain will be enriched using the built-in entity enrichment capabilities (entity enrichment license key required).

To configure the MarkLogic Server entity enrichment CPF application on your system, perform the following steps:

1. Install content processing on the database in which you want to enrich your content.

2. Set up your domain appropriately for your content.

3. Attach the Status Change Handling and Entity Enrichment pipelines to your domain.

Any documents inserted or updated under the domain to which these pipelines are attached will now use the MarkLogic Server built-in entity enrichment, and the appropriate markup will be added to the documents.

11.3.2 Sample Pipelines Using Third-Party Technologies

There are sample pipelines and CPF applications which connect to third-party entity enrichment tools. The sample pipelines are installed in the `<marklogic-dir>/Installer/samples` directory. There are sample pipelines for the following entity enrichment tools:

- TEMIS Luxid®
- Calais OpenCalais
- SRA NetOwl
- Janya
- Data Harmony

MarkLogic Server connects to these tools via a web service. Sample code is provided on an as-is basis; the sample code is not intended for production applications and is not supported. For details, including setup instructions, see the `README.txt` file and the `samples-license.txt` file in the `<marklogic-dir>/Installer/samples` directory.

11.3.3 Custom Entity Enrichment Pipelines

You can create custom CPF applications to enrich your documents using other third-party enrichment applications. To create a custom CPF application you will need the third party application, a way to connect to it (via a web service, for example), and you will need to write XQuery code and a pipeline file similar to the ones used for the sample applications described in the previous section.

12.0 Creating Alerting Applications

This chapter describes how to create alerting applications in MarkLogic Server as well as describes the components of alerting applications, and includes the following sections:

- Overview of Alerting Applications in MarkLogic Server
- cts:reverse-query Constructor
- XML Serialization of cts:query Constructors
- Security Considerations of Alerting Applications
- Indexes for Reverse Queries
- Alerting API
- Alerting Sample Application

12.1 Overview of Alerting Applications in MarkLogic Server

An *alerting application* is used to notify users when new content is available that matches a predefined (and usually stored) query. MarkLogic Server includes several infrastructure components that you can use to create alerting applications that have very flexible features and perform and scale to very large numbers of stored queries.

A sample alerting application, which uses the Alerting API, is available as an open source project on github (https://github.com/marklogic/alerting). The sample application has all of the low-level components needed in many enterprise-class alerting applications, but it is packaged in a sample application with a user interface designed to demonstrate the functionality of an alert application; your own applications would likely have a very different and more powerful user interface. Also, the sample application is for demonstration purposes only, and is not designed to be put into production; see the `samples-license.txt` file in the `<marklogic-dir>/Samples` directory for more information. If you do not care about understanding the low-level components of an alerting application, you can skip to the sections of this chapter about the Alerting API and the sample application, “Alerting API” on page 199 and “Alerting Sample Application” on page 206.

The heart of the components for alerting applications is the ability to create *reverse queries*. A reverse query (`cts:reverse-query`) is a `cts:query` that returns true if the node supplied to the reverse query would match a query if that query were run in a search. For more details about `cts:reverse-query`, see “cts:reverse-query Constructor” on page 197.

Alerting applications use reverse queries and several other components, including serialized `cts:query` constructors, reverse query indexes, MarkLogic Server security components, the Alert API, Content Processing Framework domains and pipelines, and triggers.

> **Note:** Alerting applications require a valid alerting license key. The license key is required to use the reverse index and to use the Alerting API.

12.2 cts:reverse-query Constructor

The `cts:reverse-query` constructor is used in a `cts:query` expression. It returns true for `cts:query` nodes that match an input. For example, consider the following:

```
let $node := <a>hello there</a>
let $query := <xml-element>{cts:word-query("hello")}</xml-element>
return
cts:contains($query, cts:reverse-query($node))
(: returns true :)
```

This query returns `true` because the `cts:query` in `$query` would match `$node`. In concept, the `cts:reverse-query` constructor is the opposite of the other `cts:query` constructors; while the other `cts:query` constructors match documents to queries, the `cts:reverse-query` constructor matches queries to documents. This functionality is the heart of an alerting application, as it allows you to efficiently run searches that return all queries that, if they were run, would match a given node.

The `cts:reverse-query` constructor is fully composable; you can combine the `cts:reverse-query` constructor with other constructors, just like you can any other `cts:query` constructor. The Alerting API abstracts the `cts:reverse-query` constructor from the developer, as it generates any needed reverse queries. For details about how `cts:query` constructors work, see “Composing cts:query Expressions” on page 121.

12.3 XML Serialization of cts:query Constructors

A `cts:query` expression is used in a search to specify what to search for. A `cts:query` expression can be very simple or it can be arbitrarily complex. In order to store `cts:query` expressions, MarkLogic Server has an XML representation of a `cts:query`. Alerting applications store the serialized XML representation or `cts:query` expressions and index them with the reverse index. This provides fast and scalable answers to searches that ask “what queries match this document.” Storing the XML representation of a `cts:query` in a database is one of the components of an alerting application. The Alerting API abstracts the XML serialization from the developer. For more details about serializing a `cts:query` to XML, see the XML Serializations of cts:query Constructors section of the chapter “Composing cts:query Expressions” on page 121.

12.4 Security Considerations of Alerting Applications

Alerting applications typically allow individual users to create their own criteria for being alerted, and therefore there are some inherent security requirements in alerting applications. For example, you don’t want everyone to be alerted when a particular user’s alerting criteria is met, you only want that particular user alerted. This section describes some of the security considerations and includes the following parts:

- Alert Users, Alert Administrators, and Controlling Access
- Predefined Roles for Alerting Applications

12.4.1 Alert Users, Alert Administrators, and Controlling Access

Because there is both a need to manage an alerting application and a need for users of the alerting application to have some ability to perform actions on the database, alerting applications need to manage security. Users of an alerting application need to run some queries that they might not be privileged to run. For example, they need to look at configuration information in a controlled way. To manage this, alerting applications can use amps to allow users to perform operations for which they do not have privileges by providing the needed privileges *only* in the context of the alerting application. For details about amps and the MarkLogic Server security model, see the *Understanding and Using Security Guide* guide.

The Alerting API, along with the built-in roles `alert-admin` and `alert-user`, abstracts all of the complex security logic so you can create a applications that properly deal with security, but without having to manage the security yourself.

12.4.2 Predefined Roles for Alerting Applications

There are two pre-defined roles designed for use in alerting applications that are built using the Alerting API, as well as some internal roles that the Alerting API uses:

- Alert-Admin Role
- Alert-User Role
- Roles For Internal Use Only

12.4.2.1 Alert-Admin Role

The `alert-admin` role is designed to give administrators of an alerting applications all of the privileges that are needed to create configurations (alert configs) with the Alerting API. It has a significant amount of privileges, including the ability to run code as any user that has a rule, so only trusted users (users who are assumed to be non-hostile, appropriately trained, and follow proper administrative procedures) should be granted the `alert-admin` role. Assign the `alert-admin` role to administrators of your alerting application.

12.4.2.2 Alert-User Role

The `alert-user` role is a minimally privileged role. It is used in the Alerting API to allow regular alert users (as opposed to `alert-admin` users) to be able to execute code in the Alerting API. Some of that code needs to read and update documents used by the alerting application (configuration files, rules, and so on), and this role provides a mechanism for the Alerting API to give the access needed (and no more access) to users of an alerting application.

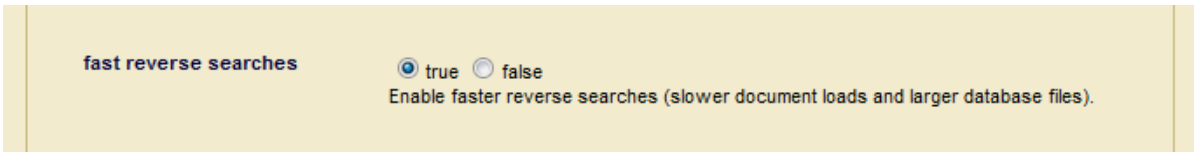
The `alert-user` role only has privileges that are used by the Alerting API; it does not provide execute privileges to any functions outside the scope of the Alerting API. The Alerting API uses the `alert-user` role as a mechanism to amp more privileged operations in a controlled way. It is therefore reasonably safe to assign this role to any user whom you trust to use your alerting application.

12.4.2.3 Roles For Internal Use Only

There are also two other roles used by the Alerting API which you should not explicitly grant to any user or role: `alert-internal` and `alert-execution`. These roles are used to amp special privileges within the context of certain functions of the Alerting API, and giving these roles to any users would give them privileges on the system that you might not want them to have; do not grant these roles to any users.

12.5 Indexes for Reverse Queries

You enable or disable the reverse query index in the database configuration by setting the `fast reverse searches` index setting to `true`:



The `fast reverse searches` index speeds up searches that use `cts:reverse-query`. For alerting applications to scale to large numbers of rules, you should enable fast reverse searches.

> **Note:** Alerting applications require a valid alerting license key. The license key is required to use the reverse index and to use the Alerting API.

12.6 Alerting API

The Alerting API is designed to help you build a robust alerting application. The API handles the details for security in the application, as well as provides mechanisms to set up all of the components of an alerting application. It is designed to make it easy to use triggers and CPF to keep the state of documents being alerted. This section describes the Alerting API and includes the following parts:

- Alerting API Concepts
- Using the Alerting API
- Using CPF With an Alerting Application

The Alerting API is implemented as an XQuery library module. For the individual function signatures and descriptions, see the *MarkLogic XQuery and XSLT Function Reference*.

12.6.1 Alerting API Concepts

There are three main concepts to understand when using the Alerting API:

- Alert Config
- Actions to Execute When an Alert Fires
- Rules For Firing Alerts

12.6.1.1 Alert Config

The alert *config* is the XML representation of an alerting configuration for an alerting application. Typically, an alerting application needs only one alert config, although you can have many if you need them. The Alerting API defines an XML representation of an alert config, and that XML representation is returned from the `alert:make-config` function. You then persist the config in the database using the `alert:config-insert` function. The Alerting API also has setter and getter functions to manipulate an alert config. The alert config is designed to be created and updated by an administrator of the alerting application, and therefore users who manipulate the alert config must have the `alert-admin` role.

12.6.1.2 Actions to Execute When an Alert Fires

An *action* is some XQuery code to execute when an alert occurs. An action could be to update a document in the database, to send an email, or whatever makes sense for your application. The action is an XQuery main module, and the Alerting API defines an XML representation of an action, and that XML representation is returned from the `alert:make-action` function. The action XML representation points to the XQuery main module that performs the action. You then persist this XML representation of an alert action in the database using the `alert:action-insert` function. The Alerting API also has setter and getter functions to manipulate an alert action. Alert actions are designed to be created and updated by an administrator of the alerting application, and therefore users who manipulate alert actions must have the `alert-admin` role.

Alert actions are invoked or spawned with `alert:invoke-matching-actions` or `alert:spawn-matching-actions`, and the actions can accept the following external variables:

```
declare namespace alert = "http://marklogic.com/xdmp/alert";

declare variable $alert:config-uri as xs:string external;
declare variable $alert:doc as node() external;
declare variable $alert:rule as element(alert:rule) external;
declare variable $alert:action as element(alert:action) external;
```

These external variables are available to the action if it needs to use them. To use the variables, the above variable declarations must be in the prolog of the action module that is invoked or spawned.

12.6.1.3 Rules For Firing Alerts

A *rule* is the criteria for which a user is alerted combined with a reference to an action to perform if that criteria is met. For example, if you are interested in any new or changed content that matches a search for `jelly beans`, you can define a rule that fires an alert when a new or changed document comes in that has the term `jelly beans` in it. This might translate into the following `cts:query`:

```
cts:word-query("jelly beans")
```

The rule also has an action associated with it, which will be performed if the document matches the query. Alerting applications are designed to support very large numbers of rules with fast, scalable performance. The amount of work for each rule also depends on what the action is for each rule. For example, if you have an application that has an action to send an email for each matching rule, you must consider the impact of sending all of those emails if you have large numbers of matching rules.

The Alerting API defines an XML representation of a rule, and that XML representation is returned from the `alert:make-rule` function. You then persist the rule in the database using the `alert:rule-insert` function. Rules are designed to be created and updated by regular users of the alerting application. The Alerting API also has setter and getter functions to manipulate an alert rule. Because those regular users who create rules must have the needed privileges and permissions to perform certain tasks (such as reading and updating certain documents), a minimal set of privileges are required to insert a rule. Therefore users who create rules in an alerting application must have the `alert-user` role, which has a minimum set of privileges.

12.6.2 Using the Alerting API

Once you understand the concepts described in the previous section, using the Alerting API is straight-forward. This section describes the following details of using the Alerting API:

- Set Up the Configuration (User With alert-admin Role)
- Set Up Actions (User With alert-admin Role)
- Create Rules (Users With alert-user Role)
- Run the Rules Against Content

12.6.2.1 Set Up the Configuration (User With alert-admin Role)

The first step in using the Alerting API is to create an alert config. For details about an alert config, see “Alert Config” on page 200. You should create the alert config as an alerting application administrator (a user with the `alert-admin` role or the `admin` role). The following sample code demonstrates how to create an alert config:

```
(: run this a user with the alert-admin role :)
xquery version "1.0-ml";
import module namespace alert = "http://marklogic.com/xdmp/alert"
  at "/MarkLogic/alert.xqy";

let $config := alert:make-config(
      "my-alert-config-uri",
      "My Alerting App",
      "Alerting config for my app",
        <alert:options/> )
return
alert:config-insert($config)
```

12.6.2.2 Set Up Actions (User With alert-admin Role)

An alerting application administrator must also set up actions to be performed when an alert occurs. An action is an XQuery main module and can be arbitrarily simple or arbitrarily complex. For details about alert actions, see “Actions to Execute When an Alert Fires” on page 200.

In practice, setting up the actions will require a good understanding of what you are trying to accomplish in an alerting application. The following is an extremely simple action that sends a log message to the `ErrorLog.txt` file. To create this action, create an XQuery document under your App Server root with the following content:

```
xdmp:log(fn:concat(xdmp:get-current-user(), " was alerted"))
```

For example, if the App Server in which your alerting application is running has a root of `/space/alert`, put this XQuery document in `/space/alert/log.xqy`.

For another example of an alert action, see the `<marklogic-dir>/Modules/alert/log.xqy` XQuery file. Alert actions can perform any action you can write in XQuery.

You also need to set up the action with the Alerting API, as in the following example (run this as a user with the `alert-admin` role):

```
xquery version "1.0-ml";
import module namespace alert = "http://marklogic.com/xdmp/alert"
  at "/MarkLogic/alert.xqy";

let $action := alert:make-action(
    "xdmp:log",
    "log to ErrorLog.txt",
    xdmp:modules-database(),
    xdmp:modules-root(),
    "/alert-action.xqy",
    <alert:options>put anything here</alert:options> )
return
alert:action-insert("my-alert-config-uri", $action)
```

12.6.2.3 Create Rules (Users With alert-user Role)

You should set up the alerting application so that regular users of the application can create rules. You might have a form, for example, to assist users in creating the rules. The following is the basic code needed to set up a rule. Note that your code will probably be considerably more complex, as this code has no user interface. This code simply creates a rule with the action described above and associates a `cts:query` with the rule. Run this code as a user with the `alert-user` role.

```
xquery version "1.0-ml";
import module namespace alert = "http://marklogic.com/xdmp/alert"
  at "/MarkLogic/alert.xqy";

let $rule := alert:make-rule(
    "simple",
    "hello world rule",
    0, (: equivalent to xdmp:user(xdmp:get-current-user()) :)
    cts:word-query("hello world"),
    "xdmp:log",
    <alert:options/> )
return
alert:rule-insert("my-alert-config-uri", $rule)
```

Note: If your action performs any privileged activities, including reading or creating documents in the database, you will need to add the appropriate execute privileges and URI privileges to users running the application.

12.6.2.4 Run the Rules Against Content

To make the application fire alerts (that is, execute the actions for rules), you must run the rules against some content. You can do this in several ways, including setting up triggers with the Alerting API (`alert:create-triggers`), using CPF and the Alerting pipeline, or creating your own mechanism to run the rules against content.

To run the rules manually, you can use the `alert:spawn-matching-actions` or `alert:invoke-matching-actions` APIs. These are useful to run alerts in any context, either within an application or as an easy way to test your rules. The `alert:spawn-matching-actions` is good when you have many alerts that might fire at once, because it will spawn the actions to the task server to execute asynchronously. The `alert:invoke-matching-actions` API runs the action immediately, so be careful using this if there can be large numbers of matching actions, as they will all be run in the same context. You can run these APIs as any user, and whether or not they produce an action will depend upon what each rule's owner has permissions to see. The following is a very simple example that fires the previously created alert:

```
xquery version "1.0-ml";
import module namespace alert = "http://marklogic.com/xdmp/alert"
  at "/MarkLogic/alert.xqy";

alert:invoke-matching-actions("my-alert-config-uri",
      <doc>hello world</doc>, <options/>)
```

If you created the config, action, and rule as described in the previous sections, this logs the following to your `ErrorLog.txt` file when running the code as a user named `some-user` who has the `alert-user` role (assuming this user created the rule):

```
some-user was alerted
```

Note: If you have very large numbers of alerts, and if the actions for your rules are resource-intensive, invoking or spawning matching actions can produce a significant amount of query activity for your system. This is OK, as that is the purpose of an alerting application, but you should plan your resources accordingly.

12.6.3 Using CPF With an Alerting Application

It is a natural fit to use alerting applications built using the Alerting API with the Content Processing Framework (CPF). CPF is designed to keep state for documents, so it is easy to use CPF to keep track of when a document in a particular scope is created or updated, and then perform some action on that document. For alerting applications, that action involves running a reverse query on the changed documents and then firing alerts for any matching rules (the Alerting API abstracts the reverse query from the developer).

To simplify using CPF with alerting applications, there is a pre-built pipeline for alerting. The pipeline is designed to be used with an alerting application built with the Alerting API. This Alerting CPF application will run alerts on all new and changed content within the scope of the CPF domain to which the Alerting pipeline is attached.

If you use the Alerting pipelines with any of the other pipelines included with MarkLogic Server (for example, the conversion pipelines and/or the modular documents pipelines), the Alerting pipeline is defined to have a priority such that it runs after all of the other pipelines have completed their processing. This way, alerts happen on the final view of content that runs through a pipeline process. If you have any custom pipelines that you use with the Alerting pipeline, consider adding priorities to those pipelines so the alerting occurs in the order in which you are expecting.

To set up a CPF application that uses alerting, perform the following steps:

1. Enable the reverse index for your database, as described in “Indexes for Reverse Queries” on page 199.

2. Set up the alert config and alert actions as a user with the `alert-admin` role, as described in “Set Up the Configuration (User With alert-admin Role)” on page 202 and “Set Up Actions (User With alert-admin Role)” on page 202.

3. Set up an application to have users (with the `alert-user` role) define rules, as described in “Create Rules (Users With alert-user Role)” on page 203.

4. Install Content Processing in your database, if it is not already installed (Databases > *database_name* > Content Processing > Install tab).

5. Set up the `domain scope` for a domain.

6. Attach the Alerting pipeline and the Status Change Handling pipeline to the domain. You can also attach any other pipelines you need to the domain (for example, the various conversion pipelines).

7. Use either the `alert:config-set-cpf-domain-names` or `alert:config-set-cpf-domain-ids` function to notify the alerting configuration of the domain, so that the alerting action can determine what alerting configuration it is to use.

   For example, if your CPF domain's name is Default Documents, you could do the following.

```
alert:config-insert(
   alert:config-set-cpf-domain-names(
      alert:config-get($config-uri),
      ("Default Documents")))
```

Note: An alerting configuration can be used with multiple CPF domains, in which case you set a sequence of multiple domain names or IDs.

Any new or updated content within the domain scope will cause all matching rules to fire their corresponding action. If you will have many alerts that are spawned to the task server, make sure the task server is configured appropriately for your machine. For example, if you are running on a machine that has 16 cores, you might want to raise the `threads` setting for the task server to a higher number then the default of 4. What you set the `threads` setting depends on what other work is going on your machine.

For details about CPF, see the *Content Processing Framework Guide* guide.

12.7 Alerting Sample Application

A sample alerting application is available on developer.marklogic.com/code. The sample application uses the Alerting API, and has all of the low-level components needed in many enterprise-class alerting applications, but it is packaged in a sample application with a user interface designed to demonstrate the functionality of an alert application; your own applications would likely have a very different and more powerful user interface. This sample code is provided on an as-is basis; the sample code is not intended for production applications and is not supported. For details, including setup instructions, see the code on developer.marklogic.com/code.

13.0 Using fn:count vs. xdmp:estimate

This chapter descibes some of the differences between the `fn:count` and `xdmp:estimate` functions, and includes the following sections:

- fn:count is Accurate, xdmp:estimate is Fast
- The xdmp:estimate Built-In Function
- Using cts:remainder to Estimate the Size of a Search
- When to Use xdmp:estimate

13.1 fn:count is Accurate, xdmp:estimate is Fast

The XQuery language provides general support for counting the number of items in a sequence through the use of the `fn:count` function. However, the general-purpose nature of `fn:count` makes it difficult to optimize. Sequences to be counted can include arbitrarily complex combinations of sequences stored in the database, constructed dynamically, filtered after retrieval or construction, etc. In most cases, MarkLogic Server must process the sequence in order to count it. This can have significant I/O requirements that would impact performance.

MarkLogic Server provides the `xdmp:estimate` XQuery built-in as an efficient way to approximate `fn:count`. Unlike `fn:count`, which frequently must process its answer by inspecting the data directly (hence the heavy I/O loads), `xdmp:estimate` computes its answer directly from indexes. In certain situations, the index-derived value will be identical to the value returned by `fn:count`. In others, the values differ to a varying degree depending on the specified sequence and the data. In instances where `xdmp:estimate` is not able to return a fast estimate, it will throw an error. Hence, you can depend on `xdmp:estimate` to be fast, just as you can depend on `fn:count` to be accurate.

Effectively, `xdmp:estimate` puts the decision to optimize counting through use of the indexes in the hands of the developer.

13.2 The xdmp:estimate Built-In Function

`xdmp:estimate` accepts searchable XPath expressions as its parameter and returns an approximation of the number of items in the sequence:

```
xdmp:estimate(/book)
xdmp:estimate(//titlepage[cts:contains(., "primer")])
xdmp:estimate(cts:search(//titlepage, cts:word-query("primer")))
xdmp:estimate(/object[.//id = "57483"])
```

`xdmp:estimate` does not always return the same value as `fn:count`. The `fn:count` function returns the exact number of items in the sequence that is provided as a parameter. In contrast, `xdmp:estimate` provides an answer based on the following rules:

1. If the parameter passed to `xdmp:estimate` is a searchable XPath expression, `xdmp:estimate` returns the number of fragments that it will select from the database for post-filtering. This number is computed directly from the indexes at extremely high performance. It may, however, differ from the actual `fn:count` of the sequence specified if either (a) there are multiple matching items within a single fragment or (b) there are fragments provisionally selected by the indexes that do not actually contain a matching item.

2. If the parameter passed to `xdmp:estimate` is not a searchable XPath expression (that is, it is not an XPath rooted at a `doc`, `collection()`, or `input()` function, or a / or // step), `xdmp:estimate` will throw an error.

`xdmp:estimate` is defined in this way to ensure a sharp contrast against the `fn:count` function. `xdmp:estimate` will always execute quickly. `fn:count` will always return the "correct" answer. Over time, as MarkLogic improves the server's underlying optimization capability, there will be an increasing number of scenarios in which `fn:count` is both correct and fast. But for the moment, we put the decision about which approach to take in the developer's hands.

13.3 Using cts:remainder to Estimate the Size of a Search

For times when you need both an estimate of the search size and need to actually run the search as part of the same query statement, you can use the `cts:remainder` function to estimate the size of the search. Running `cts:remainder` on a search and the search is more efficient that running `xdmp:estimate` on a search and the search. If you just need the estimate, but not the actual search results, then `xdmp:estimate` is more efficient.

`cts:remainder` returns the number of nodes remaining from a particular node of a search result set. When you run it on the first node, it returns the same result as `xdmp:estimate` on the search. `cts:remainder` also has the flexibility to return the estimated results of a search starting with any item in the search (for example, how many results remain after the 500th search item), and it does this in an efficient way.

Like `xdmp:estimate`, `cts:remainder` uses the indexes to find the *approximate* results based on unfiltered results. For an explanation of unfiltered results, see "Using Unfiltered Searches for Fast Pagination" in the *Query Performance and Tuning Guide*. For the syntax and examples of `cts:remainder`, see the *MarkLogic XQuery and XSLT Function Reference*.

13.4 When to Use xdmp:estimate

MarkLogic Server uses its indexes to *approximate* the identification of XML fragments that may contain constructs that match~~es~~ the specified XPath. This set of fragments is then filtered to determine the exact nodes to return for further processing.

For searchable XPath expressions, `xdmp:estimate` returns the number of fragments selected in the first approximation step described above. Because this operation is carried out directly from indexes, the operation is virtually instantaneous. However, there are two scenarios in which this approximation will not match the results that would be returned by `fn:count`:

1. If a fragment contains more than one matching item for the XPath specified, `xdmp:estimate` will *undercount* these items as a single item whereas `fn:count` would count them individually.
2. In addition, it is possible to *overcount*. Index optimization sometimes must over-select in order to ensure that no matching item is missed. During general query processing, these over-selected fragments are discarded in the second-stage filtering process. But `xdmp:estimate` will count these fragments as matching items whereas `fn:count` would exclude them.

Consider the sample query outlined below. The first step in the optimization algorithm outlined above is illustrated by the `xdmp:query-trace` output shown after the query:

Query:

```
/MedlineCitationSet/MedlineCitation//Author[LastName="Smith"])
```

Query trace output:

```
2004-04-06 17:49:39 Info: eval line 5: Analyzing path:
fn:doc()/child::MedlineCitationSet/child::MedlineCitation/
descendant::Author[child::LastName = "Smith"]
2004-04-06 17:49:39 Info: eval line 5: Step 1 is searchable: fn:doc()
2004-04-06 17:49:39 Info: eval line 4: Step 2 axis does not use
indexes:child
2004-04-06 17:49:39 Info: eval line 4: Step 2 test is searchable:
MedlineCitationSet
2004-04-06 17:49:39 Info: eval line 5: Step 2 is searchable:
child::MedlineCitationSet
2004-04-06 17:49:39 Info: eval line 4: Step 3 axis does not use
indexes:child
2004-04-06 17:49:39 Info: eval line 4: Step 3 test is searchable:
MedlineCitation
2004-04-06 17:49:39 Info: eval line 5: Step 3 is searchable:
child::MedlineCitation
2004-04-06 17:49:39 Info: eval line 5: Step 4 axis does not use
indexes:descendant
2004-04-06 17:49:39 Info: eval line 5: Step 4 test is searchable:
Author
2004-04-06 17:49:39 Info: eval line 5: Step 4 predicate 1 is
searchable:
child::LastName = "Smith"
2004-04-06 17:49:39 Info: eval line 5: Step 4 is searchable:
descendant::Author[child::LastName = "Smith"]
2004-04-06 17:49:39 Info: eval line 5: Path is searchable.
2004-04-06 17:49:39 Info: eval line 5: Gathering constraints.
```

```
2004-04-06 17:49:39 Info: eval line 4: Step 2 test contributed 1
constraint: MedlineCitationSet
2004-04-06 17:49:39 Info: eval line 4: Step 3 test contributed 2
constraints: MedlineCitation
2004-04-06 17:49:39 Info: eval line 5: Step 4 test contributed 1
constraint: Author
2004-04-06 17:49:39 Info: eval line 4: Comparison contributed hash
value constraint: LastName = "Smith"
2004-04-06 17:49:39 Info: eval line 5: Step 4 predicate 1 contributed 1
constraint: child::LastName = "Smith"
2004-04-06 17:49:39 Info: eval line 5: Executing search.
2004-04-06 17:49:39 Info: eval line 5: Selected 263 fragments to filter
```

In this scenario, applying `fn:count` to the XPath provided would tell us that there are 271 authors with a last name of "Smith" in the database. Using `xdmp:estimate` yields an answer of 263. In this example, `xdmp:estimate` undercounted because there are fragments with multiple authors named "Smith" in the database, and xdmp:estimate only counts the number of fragments.

Understanding when these situations will occur with a given database and dataset requires an in-depth understanding of the optimizer. Given that the optimizer evolves with every release of the server, this is a daunting task.

The following three sets of guidelines will help you know when and how to use `xdmp:estimate`:

- When Estimates Are Good Enough
- When XPaths Meet The Right Criteria
- When Empirical Tests Demonstrate Correctness

13.4.1 When Estimates Are Good Enough

In some situations, an estimate of the correct answer is good enough. Many search engines use this approach today, only estimating the total number of "hits" when displaying the first twenty results to the user. In scenarios in which the exact count is not important, it makes sense to use `xdmp:estimate`.

13.4.2 When XPaths Meet The Right Criteria

If you need to get the precise answer rather than just an approximation, there are some simple criteria to keep in mind if you want to use `xdmp:estimate` for its performance benefits:

1. Counting nodes that are either fragment or document roots will always return the correct result.

   Examples:

   ```
   xdmp:estimate(/node-name) is equivalent to count(/node-name)
   ```

```
xdmp:estimate(//MedlineCitation) is equivalent to count(//MedlineCitation)
```

if `MedlineCitation` is a fragment-root. For example, this constraint is how the sample Medline application is configured in the sample code on `http://support.marklogic.com`.

2. If a single fragment can contain more than one element that matches a predicate, you have the potential for undercounting. Assume that the sample data below resides in a single fragment:

```
<authors>
  <author>
    <last-name>Smith</last-name>
    <first-name>Alison</first-name>
  </author>
  <author>
    <last-name>Smith</last-name>
    <first-name>James</first-name>
  </author>
  <author>
    <last-name>Peterson</last-name>
    <first-name>David</first-name>
  </author>
</authors>
```

   In this case, an XPath which specifies `fn:doc()//author[last-name = "Smith"]` will *undercount*, counting only one item for the two matches in the above sample data.

3. If the XPath contains multiple predicates, you have the potential of overcounting. Using the sample data above, an XPath which specifies `fn:doc()//author[last-name = "Smith"][first-name = "David"]` will not have any matches. However, since the above fragment contains author elements that satisfy the predicates `[last-name = "Smith"]` and `[first-name = "David"]` individually, it will be selected for post-filtering. In this case, `xdmp:estimate` will consider the above fragment a match and overcount.

13.4.3 When Empirical Tests Demonstrate Correctness

As a last step, you can use two techniques to understand the value that will be returned by `xdmp:estimate`:

1. At development time, use `xdmp:estimate` and `fn:count` to count the same sequence and see if the results are different for datasets which exhibit all the structural variation you expect in your production dataset.

2. Turn on `xdmp:query-trace`, evaluate the XPath sequence that you wish to use with `xdmp:estimate`, and inspect the query-trace output in the log file. This output will tell you how much of the XPath was searchable, how many fragments were selected (this is the answer that `xdmp:estimate` will provide), and how many ultimately matched (this is the answer that `fn:count` will provide).

14.0 Understanding and Using Stemmed Searches

This chapter describes how to use the stemmed search functionality in MarkLogic Server. The following sections are included:

- Stemming in MarkLogic Server
- Enabling Stemming
- Stemmed Searches Versus Word Searches
- Using cts:highlight or cts:contains to Find if a Word Matches a Query
- Interaction With Wildcard Searches

14.1 Stemming in MarkLogic Server

MarkLogic Server supports stemming in English and other languages. For a list of languages in which stemming is supported, see “Supported Languages” on page 301.

If stemmed searches are enabled in the database configuration, MarkLogic Server automatically searches for words that come from the same *stem* of the word specified in the query, not just the exact string specified in the query. A stemmed search for a word finds the exact same terms as well as terms that derive from the same meaning and part of speech as the search term. The stem of a word is not based on spelling. For example, `card` and `cardiac` have different stems even though the spelling of `cardiac` begins with `card`. On the other hand, `running` and `ran` have the same stem (`run`) even though their spellings are quite different. If you want to search for a word based on partial pattern matching (like the `card` and `cardiac` example above), use wildcard searches as described in “Understanding and Using Wildcard Searches” on page 231.

Stemming enables the search to return highly relevant matches that would otherwise be missed. For example, the terms `run`, `running`, and `ran` all have the same stem. Therefore, a stemmed search for the term `run` returns results that match elements containing the terms `running`, `ran`, and `runs`, as well as results for the specified term `run`.

The stemming supported in MarkLogic Server does not cross different parts of speech. For example, `conserve` (verb) and `conservation` (noun) are not considered to have the same stem because they have different parts of speech. Consequently, if you search for `conserve` with stemmed searches enabled, the results will include documents containing `conserve` and `conserves`, but not documents with `conservation` (unless `conserve` or `conserves` also appears).

Stemming is language-specific, that is, each word is treated to be in the specified language. The language can be specified with an `xml:lang` attribute or by several other methods, and a term in one language will not match a stemmed search for the same term in another language. For details on how languages affect queries, see “Querying Documents By Languages” on page 298.

Contains SAP BusinessObjects Text Analysis XI from SAP AG. Copyright © 1996-2011. All rights reserved.

14.2 Enabling Stemming

To use stemming in your searches, stemming must be enabled in your database configuration. All new databases created in MarkLogic Server have stemming enabled by default. Stemmed searches require indexes which are created at load, update, or reindex time. If you enable stemming in an existing database that did not previously have stemming enabled, you must either reload or reindex the database to ensure that you get stemmed results from searches. You should plan on allocating an additional amount of disk space about twice the size of the source content if you enable stemming.

There are three types of stemming available in MarkLogic Server: basic, advanced, and decompounding. The following table describes the stemming options available on the database configuration page of the Admin Interface.

Stemming Option	Description
OFF	No words are indexed for stemming.
Basic	This is the default. Each word is indexed to a single stem.
Advanced	Each word is indexed to one or more stems. Some words can have two or more meanings, and can therefore have multiple stems. For example, the word `further` stems to `further` (as in *he attended the party to further his career*) and it stems to `far` (as in *she was further along in her studies than he*).
Decompounding	All stems for each word are indexed, and smaller component words of large compound words are also indexed. Mostly used in languages such as German that use compound words.

If stemming is enabled for the database, you can further control the use of stemming at the query level. Stemming can be used with any of the MarkLogic `cts:query` constructor functions, such as `cts:word-query`, `cts:element-word-query`, and `cts:element-value-query`. Stemming options, `"stemmed"` or `"unstemmed"`, can be specified in the options parameter to the `cts:query` constructor. For more details on these functions, see the *MarkLogic XQuery and XSLT Function Reference* (http://docs.marklogic.com).

Query terms that contain a wildcard will not be stemmed. If you leave the stemming option unspecified, the system will perform a stemmed search for any word that does not contain a wildcard. Therefore, as long as stemming is enabled in the database, you do not have to enable stemming explicitly in a query.

If stemming is turned off in the database, and stemming is explicitly specified in the query, the query will throw an error.

14.3 Stemmed Searches Versus Word Searches

The stemmed search indexes and word search (unstemmed) indexes have overlapping functionality, and there is a good chance you can get the results you want with only the stemmed search indexes enabled (that is, leaving the word search indexes turned off).

Stemmed searches return relevance-ranked results for the words you search for *as well as for* words with the same stem as the words you search for. Therefore, you will get the same results as with a word search plus the results for items containing words with the same stem. In most search applications, this is the desirable behavior.

The only time you need to also have word search indexes enabled is when your application requires an exact word search to *only* return the exact match results (that is, to *not* return results based on stemming).

Additionally, the stemmed search indexes take up less disk space than the word search (unstemmed) indexes. You can therefore save some disk space and decrease load time when you use the default settings of stemmed search enabled and word search turned off in the database configuration. Every index has a cost in terms of disk space used and increased load times. You have to decide based on your application requirements if the cost of creating extra indexes is worthwhile for your application, and whether you can fulfill the same requirements without some of the indexes.

If you do need to perform word (unstemmed) searches when you only have stemmed search indexes enabled (that is, when word searches are turned off in the database configuration), you must do so by first doing a stemmed search and then filtering the results with an unstemmed `cts:query`, as described in “Unstemmed Searches” on page 299.

14.4 Using cts:highlight or cts:contains to Find if a Word Matches a Query

Because stemming enables query matches for terms that do not have the same spelling, it can sometimes be difficult to find the words that actually caused the query to match. You can use `cts:highlight` to test and/or highlight the words that actually matched the query. For details on `cts:highlight`, see the *MarkLogic XQuery and XSLT Function Reference* and "Highlighting Search Term Matches" on page 167.

You can also use `cts:contains` to test if a word matches the query. The `cts:contains` function returns `true` if there is a match, `false` if there is no match. For example, you can use the following function to test if a word has the same stem as another word.

```
xquery version "1.0-ml";
declare function local:same-stem(
  $word1 as xs:string, $word2 as xs:string)
  as xs:boolean
{
  cts:contains(text{$word1},$word2)
};

(: The following returns true because
   running has the same stem as run :)
local:same-stem("run", "running")
```

14.5 Interaction With Wildcard Searches

For information about how stemmed searches and Wildcard searches interact, see "Interaction with Other Search Features" on page 235.

15.0 Custom Dictionaries for Tokenizing and Stemming

Custom dictionaries are used to customize the way words are stemmed and tokenized for each language. This chapter describes custom dictionaries and contains the following sections:

- Custom Dictionaries in MarkLogic Server
- Dictionary and Entry Schemas
- Custom Dictionary Functions
- Usage Examples

15.1 Custom Dictionaries in MarkLogic Server

If you want to change the default stemming or tokenizing behavior for any MarkLogic supported and licensed language, you can create or modify a custom dictionary. The custom dictionary entries override the built-in stemming data or add new stemming data for MarkLogic to use. For example, assume “meeting” is mapped to stem “meet” in the built-in dictionary. But, due to calendar entries, you don't want “meeting” to match “meet”. You can prevent this by adding a custom dictionary entry for the word “meeting” with “meeting” as a stem.

Custom dictionaries have three uses:

- Japanese (`ja`), Simplified Chinese (`zh`), and Traditional Chinese (`zh_hant`) all use a linguistic tokenizer to divide text into tokens (words and punctuation) since these languages do not have spaces separating words. You can change the tokenizer’s behavior by adding entries to a language’s custom dictionary.
- For all languages which tokenize based on spaces and punctuation, as well as Japanese, their dictionaries map inflections of words to their dictionary form. For example, English’s dictionary maps “views”, ‘viewed”, and “viewing” all back to their common stem, “view”. You can change, delete, or add dictionary entries to modify what words are mapped to which stems. For more information on stemming and tokenizing, see “Understanding and Using Stemmed Searches” on page 212.
- Handling spelling variation and technical vocabulary, for words like “aluminum” and “aluminium”. Due to a dictionary entry, these two spellings are effectively the same for anything in the server based on stemming.

There is only one custom dictionary for each MarkLogic licensed and supported language. To get the custom dictionary for a language, first get a list of all of the licensed languages for your system. By iterating over this list, you can get any of your custom dictionaries. Custom dictionaries are stored in the data directory, so they survive MarkLogic server upgrades.

The custom dictionaries format and API make use of namespaces and are validated during dictionary writes to avoid runtime errors. However, MarkLogic does not check for duplicate entries which, while unnecessary, do not cause errors. Nor does it check for non-Latin characters in an English custom dictionary, although entries with such characters are invalid and not processed. Fixing errors in a custom dictionary does not require a restart.

Caution: Reindexing is required if documents are affected by custom dictionary changes. To determine what, if any, documents are affected before changing the dictionary, do a word search (not a stemmed search) for any words that your changes will add or delete. For example, if you are adding “viewer” to stem to “view” and deleting the “viewing” to “view” stem, you would want to search for both “viewer” and “viewing”, as these are the words which are affected by changing the dictionary. “view” itself is not affected. Documents containing those words will need reindexing after the dictionary change.

15.2 Dictionary and Entry Schemas

Custom dictionaries are stored as XML documents and use a dedicated namespace and a common entry format. Here is an example custom dictionary with a single entry:

```
<dictionary xmlns=”http://marklogic.com/xdmp/custom-dictionary”
            xml:lang=”en”>
  <entry>
    <word>servlets</word>
    <stem>servlet</stem>
    <pos>Nn</pos>
  </entry>
</dictionary>
```

Note that the root `<dictionary>` element has an `xml:lang` attribute specifying the dictionary’s associated language, in this case English.

A dictionary can have any number of `<entry>` elements, which each contain three sub-elements:

- `<word>`: A string, it contains a word to be stemmed.
- `<stem>`: A string, the stem for the `<word>` string.

- `<pos>`: For languages without space-separated words, such as Chinese and Japanese, it specifies a *part of speech*. For other languages, it is ignored. If omitted, `Nn-Prop` (proper noun) is the default value. The common values are:

Value	Part of Speech
Adj	Adjective
Adv	Adverb
Interj	Interjection
Nn	Noun
NN-Prop	Proper Noun (default)
Verb	Verb

15.3 Custom Dictionary Functions

To use the `custom-dictionary.xqy` module in your own XQuery modules, include the following line in your XQuery prolog:

```
import module namespace cdict =
   "http://marklogic.com/xdmp/custom-dictionary" at
   "/MarkLogic/custom-dictionary.xqy";
```

This imports the module and binds the module namespace to the custom dictionary module prefix `cdict`.

There are four custom dictionary related functions, as shown below. Note that you cannot edit a custom dictionary in place. Instead, you have to first get the dictionary. Then add to, edit, or delete its entries. Finally, overwrite the old version by writing the new version out.

- Get All Licensed Languages (`cdict:get-languages`)
- Get A Custom Dictionary (`cdict:dictionary-read`)
- Add/Write A Custom Dictionary (`cdict:dictionary-write`)
- Delete A Custom Dictionary (`cdict:dictionary-delete`)

For more details on these functions, see *MarkLogic XQuery and XSLT Function Reference*.

15.3.1 Get All Licensed Languages

```
cdict:get-languages()
  as xs:string*
```

Requires the `http://marklogic.com/xdmp/privileges/custom-dictionary-user` privilege. Returns the 2-letter ISO language codes for all your server's licensed languages. A list of codes and their associated languages is at http://www.loc.gov/standards/iso639-2/php/code_list.php Note that MarkLogic only uses the 2-letter ISO 639-1 codes, including `zh`'s `zh_Hant` variant.

15.3.2 Get A Custom Dictionary

```
cdict:dictionary-read(
  $lang as xs:string
) as element(cdict:dictionary)
```

Requires the `http://marklogic.com/xdmp/privileges/custom-dictionary-user` privilege. `$lang` is an ISO language code. If `$lang` matches a licensed language with a custom dictionary, the local host returns that custom dictionary (which is the same across the cluster). The dictionary's `xml:lang` attribute is also returned, indicating its associated language. If `$lang` is not a licensed language, it raises an `XDMP-LANG` error.

15.3.3 Add/Write A Custom Dictionary

```
cdict:dictionary-write(
  $lang as xs:string,
  $dict as element(cdict:dictionary)
) as empty-sequence()
```

Requires the `http://marklogic.com/xdmp/privileges/custom-dictionary-admin` privilege. `$lang` is an ISO language code. `$dict` is the custom dictionary If `$lang` matches a licensed language and `$dict` validates, the cluster installs `$dict` and returns host IDs from where the dictionary is saved. If `$lang` is not a licensed language, it raises an `XDMP-LANG` error. If validation fails, it raises validation errors. `dictionary-write` ignores the `xml:lang` attribute.

15.3.4 Delete A Custom Dictionary

```
cdict:dictionary-delete(
  $lang as xs:string
) as empty-sequence()
```

Requires the `http://marklogic.com/xdmp/privileges/custom-dictionary-admin` privilege. `$lang` is an ISO language code. If `$lang` matches a licensed language with a custom dictionary, the dictionary is deleted and it returns the host IDs from which the dictionary was deleted. Raises an `XDMP-LANG` error if `$lang` is not a licensed language..

15.4 Usage Examples

The following code shows how to find a custom dictionary, get it, add and edit entries, write out the modified dictionary, and finally how to delete it. After each code sample is an example response.

First, get the sequence of supported and licensed languages:

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                 at "/MarkLogic/custom-dictionary.xqy";

cdict:get-languages()

==> ("en", "ja", "zh", "zh_Hant")
```

Next, get the dictionary contents for a particular language, in this case English (`en`):

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";

cdict:dictionary-read("en")

=> <cdict:dictionary
        xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary"
        xml:lang="en">
     <cdict:entry>
        <cdict:word>Furbies</cdict:word>
        <cdict:stem>Furby</cdict:stem>
     </cdict:entry>
     <cdict:entry>
        <cdict:word>servlets</cdict:word>
        <cdict:stem>servlet</cdict:stem>
     </cdict:entry>
    </cdict:dictionary>
```

Put the contents in a file, for example `/var/tmp/cdict-en.xml`, then edit the file or modify it with XQuery to add new entries and/or delete or modify existing entries.

Next, install your modified dictionary in MarkLogic. The returned value is an empty sequence. Note that you do not have to specify where the dictionary goes, just what language to associate it with. The server knows where to put it. Since there can only be one custom dictionary for a language, this command overwrites any existing custom dictionary associated with the language argument.

```
xquery version "1.0-ml";
import module namespace dict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
let $dict := xdmp:document-get("/var/tmp/cdict-en.xml")/*
return
  cdict:dictionary-write("en",$dict)
=> empty sequence
```

Finally, if you want to delete your English custom dictionary, you would do something similar to the following, which returns the empty sequence.

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
cdict:dictionary-delete("en")
=> empty sequence

xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
cdict:dictionary-read("en");
```

The previous examples use several queries, each query performing its own transaction, to read, save, and modify a custom dictionary. The following example accomplishes something similar in a single transaction, where the XQuery program reads and then modifies a custom dictionary.

```
(: Add an entry to the English custom dictionary :)
xquery version "1.0-ml";
import module namespace cdict =
         "http://marklogic.com/xdmp/custom-dictionary"
              at "/MarkLogic/custom-dictionary.xqy";
  let $language := "en"
(: Get the English custom dictionary :)
  let $dictionary := cdict:dictionary-read($language)
(: Specify the new entry element as XML (not as a string) :)
  let $entry := <cdict:entry>
                  <cdict:word>views</cdict:word>
                  <cdict:stem>view</cdict:stem>
                </cdict:entry>
(: First, check if there are already any dictionary entries :)
  return if (fn:empty($dictionary))
(: If no entries, then we have to create a cdict:dictionary
   element and insert our new entry before writing it out :)
      then cdict:dictionary-write($language,
                                  element cdict:dictionary {
                                      attribute xml:lang {
                                          $language },
                                      $entry})

(: If there are entries, just insert the new entry as a node and
   write out the dictionary :)
      else cdict:dictionary-write($language,
                                  xdmp:node-insert-child(
                                     $dictionary/dictionary,$entry))

(: Finally, test the new mapping; this should return "view":)
cts:stem("views", "en")
```

16.0 Extracting Metadata and Text From Binary Documents

This chapter describes how to extract metadata and/or text from binary documents. It contains the following sections:

- Metadata and Text Extraction Overview
- Usage Examples
- Supported Binary Formats

16.1 Metadata and Text Extraction Overview

Binary documents often have various associated metadata. For example, a JPEG image from a camera may have metadata of the camera's type and model number, a timestamp of when it was taken, and so on.

MarkLogic Server can access binary document metadata and then store it as XML in a properties document. You can then search and retrieve the metatdata using MarkLogic Server's rich XML search capabilities. In addition, for text-based binary documents, such as those in Microsoft Word format, MarkLogic can extract and index their text content.

There are two common ways to extract and associate metadata from binary documents: The 'Filter Documents' transform in Information Studio, and the built-in XQuery function `xdmp:document-filter`.

The 'Filter Documents' transformer automatically extracts text and metadata for certain types of documents and attaches them as document properties. For details, see Extracting Document Properties in the *Information Studio Developer's Guide*.

The `xdmp:document-filter` function extracts metadata and text from binary documents as XHTML. The results may be used as document properties. `xdmp:document-filter`'s extracted text contains little formatting or structure, so it is best used for search, classification, or other text processing.

16.2 Usage Examples

The following sections show how xdmp:document-filter works with various file types. The Microsoft Word section also provides code to extract only the metadata elements from combined metadata and text results.

- Microsoft Word
- File Archives
- PowerPoint

16.2.1 Microsoft Word

The following query and results are for a Microsoft Word document containing only the text “This is a test”:

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.docx"))
```

Returns:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta name="content-type" content="application/msword"/>
    <meta name="filter-capabilities"
          content="text subfiles HD-HTML"/>
    <meta name="AppName" content="Microsoft Office Word"/>
    <meta name="Author" content="Clark Kent"/>
    <meta name="Company" content="MarkLogic"/>
    <meta name="Creation_Date" content="2011-10-11T02:40:00Z"/>
    <meta name=”Description”
          content=”This is my comment.”/>
    <meta name="Last_Saved_Date" content="2011-10-11T02:41:00Z"/>
    <meta name="Line_Count" content="1"/>
    <meta name="Paragraphs_Count" content="1"/>
    <meta name="Revision" content="1"/>
    <meta name="Template" content="Normal"/>
    <meta name="Typist" content="Clark Kent"/>
    <meta name="Word_Count" content="4"/>
    <meta name="isys" content="SubType: Word 2007"/>
    <meta name="size" content="12691"/>
  </head>
  <body>
    <p>
    </p>
    <p>
       This is a test.</p>
    <p>
    </p>
  </body>
</html>
```

In the document, the word “test” is both italicized and bolded. `xdmp:document-filter` does not return such text formatting.

Expanding on the previous example, the following code uses `xdmp:document-filter` to extract only the metadata from that same Microsoft Word document:

```
xquery version "1.0-ml";
let $url := "/documents/test.docx"
return xdmp:document-set-properties(
  $url,
  for $meta in xdmp:document-filter(fn:doc($the-document))//*:meta
  return element {$meta/@name} {fn:string($meta/@content)}
)
```

The properties document now looks as follows:

```
xdmp:document-properties("/documents/test.docx")
```

returns:

```
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <content-type>application/msword</content-type>
  <filter-capabilities>text subfiles HD-HTML</filter-capabilities>
  <AppName>Microsoft Office Word</AppName>
  <Author>Clark Kent</Author>
  <Company>MarkLogic</Company>
  <Creation_Date>2011-10-11T02:40:00Z</Creation_Date>
  <Description>This is my comment.</Description>
  <Last_Saved_Date>2011-10-11T02:41:00Z</Last_Saved_Date>
  <Line_Count>1</Line_Count>
  <Paragraphs_Count>1</Paragraphs_Count>
  <Revision>1</Revision>
  <Subject>Creating binary doc props</Subject>
  <Template>Normal/Template>
  <Typist>Clark Kent</Typist>
  <Word_Count>4</Word_Count>
  <isys>SubType: Word 2007</isys>
  <size>12691</size>
  <prop:last-modified>2011-10-12T09:47:10-07:00</prop:last-modified>
</prop:properties>
```

16.2.2 File Archives

If you need to extract files from `zip` archives for individual processing, use `xdmp:zip-manifest` and `xdmp:zip-get`. Use `xdmp:document-filter` if you just want all the text from the archive, since it does not preserve the embedded files' structure, but includes all of the documents' text. This is useful for finding the original location in search results; if you search for "Elvis" and use `xdmp:document-filter` on the various files, the results include every binary containing "Elvis", whether it is a zip archive, Word document, or photo.

In this example, `xdmp:document-filter` runs on the file archive `test.zip`, which consists of two Word files and a JPEG file,

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.zip"))
```

returns

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta name="content-type" content="application/zip"/>
    <meta name="filter-capabilities" content="subfiles"/>
    <meta name="AppName" content="Microsoft Office Word"/>
    <meta name="Author" content="Lois Lane"/>
    <meta name="Company" content="MarkLogic"/>
    <meta name="Creation_Date" content="2011-10-14T21:11:00Z"/>
    <meta name="Last_Saved_Date" content="2011-10-14T21:11:00Z"/>
    <meta name="Line_Count" content="1"/>
    <meta name="Paragraphs_Count" content="1"/>
    <meta name="Revision" content="2"/>
    <meta name="Template" content="Normal"/>
    <meta name="Typist" content="Lois Lane"/>
    <meta name="Word_Count" content="3"/>
    <meta name="isys" content="SubType: Word 2007"/>
    <meta name="Focal_Length" content="4"/>
    <meta name="Make" content="LG Electronics"/>
    <meta name="Model" content="VM670"/>
    <meta name="Original_Date_Time" content="2011:10:19 14:59:24"/>
    <meta name="Original_Date_Time.datetime"
          content="2011-10-19T14:59:24Z"/>
    <meta name="ResolutionUnit" content="2"/>
    <meta name="XResolution" content="72.000000"/>
    <meta name="YResolution" content="72.000000"/>
    <meta name="AppName" content="Microsoft Office Word"/>
    <meta name="Author" content="Clark Kent"/>
    <meta name="Company" content="MarkLogic"/>
    <meta name="Creation_Date" content="2011-10-11T02:40:00Z"/>
    <meta name="Last_Saved_Date" content="2011-10-11T02:41:00Z"/>
    <meta name="Line_Count" content="1"/>
    <meta name="Paragraphs_Count" content="1"/>
    <meta name="Revision" content="1"/>
    <meta name="Template" content="Normal"/>
    <meta name="Typist" content="Clark Kent"/>
    <meta name="Word_Count" content="2"/>
    <meta name="isys" content="SubType: Word 2007"/>
    <meta name="size" content="47730"/>
  </head>
<body>
 <p>
 </p>
 <p>
  This is a another test.</p>
  <p>
  </p>
  <p>
    This is a test.</p>
  <p>
  </p>
</body>
</html>
```

While each sentence in this example's returned HTML body text is from a different file, there is no way to distinguish which text comes from which file. Similarly, the returned subfile metadata is not guaranteed to be returned in file order (for example, `name="a", name="b"` might be from different documents in the archive) and so also cannot be correctly associated with an individual subfile.

Also, individual subfiles in the archive are not necessarily distinguishable at all. In the above example, you cannot tell from the output how many files, or what file types, are in the archive. When using `xdmp:document-filter` on an archive, you should think of the archive as a single file, rather than a compilation of subfiles. You will get back all the metadata and text contained in the single archive file, but will have no way of associating that returned information with the individual subfiles it came from.

16.2.3 PowerPoint

The following query and results are for a two slide PowerPoint document, where each slide has a title and separate content:

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.pptx"))
```

returns:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta name="content-type"
          content="application/vnd.ms-powerpoint"/>
    <meta name="filter-capabilities" content="text subfiles HD-HTML"/>
    <title>This is a test </title>
    <meta name="AppName" content="Microsoft Office PowerPoint"/>
    <meta name="Author" content="Clark Kent"/>
    <meta name="Company" content="MarkLogic"/>
    <meta name="Creation_Date" content="2011-10-17T19:58:34Z"/>
    <meta name="Last_Saved_Date" content="2011-10-17T20:00:13Z"/>
    <meta name="Paragraphs_Count" content="4"/>
    <meta name="Presentation_Format" content="On-screen Show (4:3)"/>
    <meta name="Revision" content="1"/>
    <meta name="Slide_Count" content="2"/>
    <meta name="Typist" content="Clark Kent"/>
    <meta name="Word_Count" content="12"/>
    <meta name="isys" content="SubType: PowerPoint 2007"/>
    <meta name="size" content="36909"/>
  </head>
  <body>
<p>
</p>
<p>
This is a test </p>
<p>
```

```
Of PowerPoint</p>
<p>


</p>
<p>
Test #3
</p>
<p>
Second Slide.</p>
<p>


</p>
</body>
</html>
```

Similarly, any text formatting is not returned, nor is any indicator of what role the text played on a slide (title, body, etc.), nor is there any way to tell what text belongs to which slide.

16.3 Supported Binary Formats

The following sections list the binary file formats and file extensions from which `xdmp:document-filter` can extract metadata and, depending on the format, text from. Due to the large number of formats, they are first broken down into general application areas, such as Databases or Multimedia, then each area lists its applicable formats and extensions.

Some formats can be identified by `xdmp:document-filter`, but have no text or metadata to extract, such as executables. For these, the returned `<meta name="content-type" content=.../>` identifies the file's format.

- Archives
- Databases
- Email and Messaging
- Multimedia
- Other
- Presentation
- Raster Image
- Spreadsheet
- Text and Markup
- Vector Image
- Word Processing and General Office

16.3.1 Archives

Formats: 7-Zip, ACE, Apple Disk Image, ARJ, Bzip2, ISO Disk Image, Java Archive, LZH, Microsoft Cabinet, Microsoft Office Binder, RedHat Package Manager, Roshal Archive, Self-extracting .exe, StuffIt, StuffIt Self Extracting Archive, SuffIt X, GNU Zip, UNIX cpio, UNIX Tar, Zip, PKZip, WinZip

Extensions: .7Z, .ACE, .ARJ, .BZ2, .CAB, .CPIO, .DMG, .EXE, .GZ, .ISO, JAR, LZH, .ORD, .RAR, .RPM, .SIT, .SEA, .SITX, .TAR, .TBZ2, .ZIP

16.3.2 Databases

Formats: dBase, dBase III, Microsoft Access, Paradox Database

Extensions: .DB, .DBF, .DB3, .MDB

16.3.3 Email and Messaging

Formats: Encoded mail messages of any of the forms MHT, Multipart Alternative, Multipart Digest, Multipart Mixed, Multipart Newsgroup, Multipart Signed, and TNEF. Also, the individual formats Eudora, Microsoft Outlook, Microsoft Outlook3, Microsoft Outlook Express3. Microsoft Outlook Forms Template, Sendmail “mbox”, Thunderbird

Extensions: .EML, .MBOX, .MBX, .MHT, .MSG, .OFT, .PST

16.3.4 Multimedia

Formats: 3GP, Adobe Flash, Adobe Flash Video, Audio Video Interleave (AVI), DVD Information File, DVD Video Object, Microsoft Windows Movie Maker, Musical Instrument Digital Interface (MIDI), MPEG Video, MPEG-1 Audio Layer 3, MPEG-4 Video, MPEG-2 Audio Layer 3, OGG Flac Audio, OGG Vorbis Audio, QuickTime, Real Media, Waveform Audio File Format (WAVE), Window Media Audio, Windows Media Video.

Extensions: .3GP, .AIFF, .AVI, .BUP, .FLAC, .FLV, .IFO, .MID, .MIDI, .MOV, .MP3, .MP4, .MPG, .MSWMM, .OGG, .RM, .SMF, .SWF, .VOB, .WAV, .WMA, .WMV

16.3.5 Other

Formats: Apple Executable, BIN HEX Encoded, BitTorrent Metafile, Linux Executable and Linkable Format, Log File, Microsoft Project, Microsoft Windows DLL, Microsoft Windows Executable, Microsoft Windows Installer, Microsoft Windows, Shortcut, Open Access II (OAII), VCard, Uniplex

Extensions: .BIN, .COM, .DLL, .ELF, .EXE, .HBX, .HEX, .HQXX, .LNK, .LOG, .MPP, .MPX, .MSI, .SYS, .TORRENT, .VCF

16.3.6 Presentation

Formats: IBM Lotus Symphony Presentation, LibreOffice Presentation, Microsoft PowerPoint for Windows or Macintosh, OpenOffice Impress, StarOffice Impress

Extensions: .ODP, .ODS, .PPT, .PPTX, .SDI, .SDP, .SXI

16.3.7 Raster Image

Formats: Encapsulated PostScript, Grapics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Microsoft Document Imaging, Microsoft Windows Bitmap, PCX, Portable Network Graphic (PNG), Progressive JPEG, Tagged Image Format File (TIFF)

Extensions: .BMP, .EPS, .GFA, .GIF, .GIFF, .JIF, .JPEG, .JPG, .JPE, .MDI, .PCX, .PNG, .TIF, .TIFF

16.3.8 Spreadsheet

Formats: Comma Separated Values, Franeword Spreadsheet, IBM Lotus Symphony, LibreOffice Spreadsheet, Lotus 1-2-3, Microsoft Excel for Windows or Mac, Microsoft Works SS for DOS or Windows, OpenOffice Calc, StarOffice Calc

Extensions: .CSV, .FW3, .ODS, .SX, .SXC, .SXS, .XLS, XLSB, .XLSX, .WK., .WK3, .WK4, .WKS, .WPS

16.3.9 Text and Markup

Formats: ASCII Text (7 and 8 bit) , ANSI Text (7 and 8 bit), HTML (text only, codes revealed, metadata only), IBM DCA, Microsoft HTML Help, Microsoft OneNote, Rich Text Format, SGML Text, Source, Transcript, Unicode UTF8 and UTF16 and UCS2, XML, Windows Enhanced Meta File, Windows Meta File

Extensions: .CHM, .DCA, .EMF, .HTM, .HTML,.ONE, .RFT, .RTF, .SGML, .TXT, .XML, .WMF

16.3.10 Vector Image

Formats: Adobe Illustrator, Adobe InDesign, Adobe Photoshop, AutoCAD Drawing, AutoCAD drawing Exchange Format, Corel Draw Image, Intergraph-Microstation CAD, MathCAD, Microsoft XPS, Microsoft Visio

Extensions: .AI, .CDR, .DGN, .DWG, .DXF, .INDD, .MCD, .OXPS, .PSD, .VSD, .XMCD, .XPS

16.3.11 Word Processing and General Office

Formats: Adobe PDF, Adobe PostScript, Ami Pro for Windows, Apple iWork, Framework WP, Hangul, IBM DCA/FFT, IBM DisplayWrite, IBM Lotus Symphony Document, JustSystems Ichitaro, LibreOfffice Document, Lotus Manuscript, Lotus Notes, Mass 11, Microsoft Publisher, Microsoft Word for DOS/Windows/Macintosh, QuarkXpress, MultiMate, MultiMate Advantage, OpenOffice Writer, Professional Write for DOS, Professional Write Plus for Windows, Q&A Write, QuickBooks Backup, QuickBooks for Windows, StarOffice Writer, TrueType Font, Wang IWP, Wang WP Plus, Windows Write, WinWord, WordPerfect for DOS/Macintosh/Windows, Wordstar for DOS/Windows, Wordstar 2000 for DOS, XYwrite

Extensions: .AMI, .DCA, DOC, .DOCX, .DOX, .DW4, .FFT, .FW3, .IWP, .JTD, .JBW, .JTT, ,KEY, .M11, .MAN, .MANU, .MNU, .NSF, .NUMBERS, .ODT, PAGES, .PDF, .PS, .PUT, .QCx, .QXx, .PW, .PW1, .PW2, .QA, .QA3, .QBB, .QBW, .RFT, .SAM, .SXW, .SDW, .TTF, .WPD, WRI, .WS, .WS2, .WSD, .XY

17.0 Understanding and Using Wildcard Searches

This chapter describes wildcard searches in MarkLogic Server. The following sections are included:

- Wildcards in MarkLogic Server
- Enabling Wildcard Searches
- Interaction with Other Search Features

17.1 Wildcards in MarkLogic Server

Wildcard searches enable MarkLogic Server to return results that match combinations of characters and wildcards. Wildcard searches are not simply exact string matches, but are based on character pattern matching between the characters specified in a query and words in documents that contain those character patterns. This section describes wildcards and includes the following topics:

- Wildcard Characters
- Rules for Wildcard Searches

17.1.1 Wildcard Characters

MarkLogic Server supports two wildcard characters: `*` and `?`.

- `*` matches zero or more non-space characters.
- `?` matches exactly one non-space character.

For example, `he*` will match any word starting with `he`, such as `he`, `her`, `help`, `hello`, `helicopter`, and so on. On the other hand, `he?` will only match three-letter words starting with `he`, such as `hem`, `hen`, and so on.

17.1.2 Rules for Wildcard Searches

The following are the basic rules for wildcard searches in MarkLogic Server:

- There can be more than one wildcard in a single search term or phrase, and the two wildcard characters can be used in combination. For example, `m*??` will match words starting with `m` with three or more characters.
- Spaces are used as word breaks, and wildcard matching only works within a single word. For example, `m*th*` will match `method` but not `meet there`.
- If the `*` wildcard is specified by itself in a value query (for example, `cts:element-value-query`, `cts:element-value-match`), it matches everything (spanning word breaks). For example, `*` will match the value `meet me there`.

- If the * wildcard is specified with a non-wildcard character, it will match in value lexicon queries (for example, `cts:element-value-match`), but will not match in value queries (for example, `cts:element-value-query`). For example, `m*` will match the value `meet me there` for a value lexicon search (for example, `cts:element-value-match`) but will not match the value for a value query search (for example, `cts:element-value-query`), because the value query only matches the one word. A value search for `m* *` will match the value (because `m*` matches the first word and * matches everything after it).
- If `"wildcarded"` is explicitly specified in the `cts:query` expression, then the search is performed as a wildcard search.
- If neither `"wildcarded"` nor `"unwildcarded"` is specified in the `cts:query` expression, the database configuration and query text determine wildcarding. If the database has any wildcard indexes enabled (`three character searches`, `two character searches`, `one character searches`, or `trailing wildcard searches`) and if the query text contains either of the wildcard characters `?` or `*`, then the wildcard characters are treated as wildcards and the search is performed `"wildcarded"`. If none of the wildcard indexes are enabled, the wildcard characters are treated as punctuation and the search is performed `unwildcarded` (unless `"wildcarded"` is specified in the `cts:query` expression).
- If the query has the `punctuation-sensitive` option, then punctuation is treated as word characters for wildcard searches. For example, a `punctuation-sensitive` wildcard search for `d*benz` would match `daimler-benz`.
- If the query has the `whitespace-sensitive` option, then whitespace is treated as word characters. This can be useful for matching spaces in wildcarded value queries. You can use the `whitespace-sensitive` option in wildcarded word queries, too, although it might not make much sense, as it will match more than you might expect.

17.2 Enabling Wildcard Searches

Wildcard searches use character indexes, lexicons, and trailing wildcard indexes to speed performance. To ensure that wildcard searches are fast, you should enable at least one wildcard index (three character searches, trailing wildcard searches, two character searches, and/or one character searches) and fast element character searches (if you want fast searches within specific elements) in the Admin Interface database configuration screen. Wildcard are disabled by default. If you enable character indexes, you should plan on allocating an additional amount of disk space approximately three times the size of the source content.

This section describes the following topics:

- Specifying Wildcards in Queries
- Recommended Wildcard Index Settings
- Understanding the Different Wildcard Indexes

17.2.1 Specifying Wildcards in Queries

If any wildcard indexes are enabled for the database, you can further control the use of wildcards at the query level. You can use wildcards with any of the MarkLogic `cts:query` leaf-level functions, such as `cts:word-query`, `cts:element-word-query, and cts:element-value-query`. For details on the `cts:query` functions, see "Composing cts:query Expressions" on page 121. You can use the `"wildcarded"` and `"unwildcarded"` query option to turn wildcarding on or off explicitly in the `cts:query` constructor functions. See the *MarkLogic XQuery and XSLT Function Reference* for more details.

If you leave the wildcard option unspecified and there are any wildcard indexes enabled, MarkLogic Server will perform a wildcard query if `*` or `?` is present in the query. For example, the following search function:

```
cts:search(fn:doc(), cts:word-query("he*"))
```

will result in a wildcard search. Therefore, as long as any wildcard indexes are enabled in the database, you do not have to turn on wildcarding explicitly to perform wildcard searches.

When wildcard indexing is enabled in the database, the system will also deliver higher performance for `fn:contains`, `fn:matches`, `fn:starts-with` and `fn:ends-with` for most query expressions.

> **Note:** If character indexes, lexicons, and trailing wildcard indexes are all disabled in a database and wildcarding is explicitly enabled in the query (with the `"wildcarded"` option to the leaf-level `cts:query` constructor), the query will execute, but might require a lot of processing. Such queries will be fast if they are very selective and only need to do the wildcard searches over a relatively small amount of content, but can take a long time if they actually need to filter out results from a large amount of content.

17.2.2 Recommended Wildcard Index Settings

To enable any kind of wildcard query functionality with a good combination of performance and database size, MarkLogic recommends the following index settings:

- `word searches`
- `three character word searches`
- `word positions`
- `word lexicon` in the codepoint collation
- `three character word positions`

This combination will provide accurate and fast wildcard queries for a wide variety of wildcard searches, including leading and trailing wildcarded searches. If you add the `trailing wildcard searches` index, you will get slightly more efficient trailing wildcard searches, but with increased database size. The next section describes the various different wildcard indexes.

17.2.3 Understanding the Different Wildcard Indexes

You configure the index settings at the database level, using the Admin Interface. For details on configuring database settings and on other text indexes, see the "Databases" and Text Indexes" chapters in the *Administrator's Guide*.

The following table describes the indexing options that apply to wildcard searches.

Index	Description
Word lexicon	Speeds up wildcard searches. Works in combination with any other available wildcard indexes to improve search index resolution and performance. When used in conjunction with the `three character search` index, improves wildcard index resolution and speeds up wildcard searches. If you have `three character search` and a word lexicon enabled for a database, then there is no need for either the `one character` or `two character search` indexes. For best performance, the word lexicon should be in the codepoint collation (`http://marklogic.com/collation/codepoint`). Additionally, enabling `word searches` in the database will improve the accuracy of wildcard index resolution (more accurate `xdmp:estimate` queries).
`trailing wildcard searches`	Speeds up wildcard searches where the search pattern contains the wildcard character at the end (for example, `abc*`). Turn this index on if you want to speed wildcard searches that match a trailing wildcard. The `trailing wildcard search` index will use a similar amount of space as the `three character search` index, but will generally be more efficient for trailing wildcard queries.
`three character searches`	Speeds up wildcard searches where the search pattern contains three or more consecutive non-wildcard characters (for example, `*abc`). Turn this index on if you want to speed wildcard searches that match three or more characters anywhere in the wildcard expression. When combined with a codepoint word lexicon, speeds the performance of any wildcard search (including searches with fewer than three consecutive non-wildcard characters). MarkLogic recommends combining the `three character search` index with a codepoint collation word lexicon.
`fast element character searches`	Speeds up wildcard searches within a specific element. Also, speeds up element-based wildcard searches. Turn this index on if you want to improve performance of wildcard searches that query specific elements.
`two character searches`	Speeds up wildcard searches where the search pattern contains two or more consecutive non-wildcard characters. Turn this index on if you want to speed up wildcard searches that match two or more characters (for example, `ab*`). This index is not needed if you have `three character searches` and a word lexicon.

Index	Description
Element, Attribute, and Field word lexicons	Speeds up wildcard searches for `cts:element-value-query` and `cts:element-attribute-value-query` expressions. Works in combination with any other available wildcard indexes to improve search index resolution and performance. When used in conjunction with the `three character search` index, improves wildcard index resolution and speeds up wildcard searches.
`one character searches`	Speeds up wildcard searches where the search pattern contains only a single non-wildcard character. Turn this index on if you want to speed up wildcard searches that match one or more characters (for example, `a*`). This index is not needed if you have `three character searches` and a word lexicon.

As with all indexing, choosing which indexes you need is a trade-off. More indexes provides improved query performance, but uses more disk space and increases load and reindexing time. For most environments where wildcard searches are required, MarkLogic recommends enabling the `three character searches` and a codepoint collation word lexicon, but disabling one and two character searches.

The `three character searches` index combined with the word lexicon provides the best performance for most queries, and the `fast element character searches` index is useful when you submit element queries. One and two character searches indexes are only used if you submit wildcard searches that try and match only one or two characters and you do not have the combination of a word lexicon and the `three character searches` index. Because one and two character searches generally return a large number of matches, they might not justify the disk space and load time trade-offs.

> **Note:** If you have the `three character searches` index enabled and two and one character indexes disabled, and if you have no word lexicon, it is still possible to issue a wildcard query that searches for a two or one character stem (for example, `ab*` or `a*`); these searches are allowed, but will not be fast. If you have a search user interface that allows users to enter such queries, you might want to check for these two or one character wildcard search patterns and issue an error, as these searches without the corresponding indexes can be slow and resource-intensive. Alternatively, add a codepoint collation word lexicon to your database.

17.3 Interaction with Other Search Features

This section describes the interactions between wildcard, stemming, and other search features in MarkLogic Server. The following topics are included:

- Wildcarding and Stemming
- Wildcarding and Punctuation Sensitivity

17.3.1 Wildcarding and Stemming

Wildcard searches can be used in combination with stemming (for details on stemming, see "Understanding and Using Stemmed Searches" on page 212); that is, queries can perform stemmed searches and wildcard searches at the same time. However, the system will not perform a stemmed search on words that are wildcarded. For example, assume a search phrase of `running car*`. The term `running` will be matched based on its stem. However, `car*` will be matched based on a wildcard search, and will match `car`, `cars`, `carriage`, `carpenter` and so on; stemmed word matches for the words matching the wildcard are *not* returned.

17.3.2 Wildcarding and Punctuation Sensitivity

Stemming and punctuation sensitivity perform independently of each other. However, there is an interaction between wildcarding and punctuation sensitivity. This section describes this interaction and includes the following parts:

- Implicitly and Explicitly Specifying Punctuation
- Rules for Punctuation and Wildcarding Interaction
- Examples of Wildcard and Punctuation Interactions

17.3.2.1 Implicitly and Explicitly Specifying Punctuation

MarkLogic Server allows you to explicitly specify whether a query is punctuation sensitive and whether it uses wildcards. You specify this in the options for the query, as in the following example:

```
cts:search(fn:doc(), cts:word-query("hello!", "punctuation-sensitive") )
```

If you include a wildcard character in a punctuation sensitive search, it will treat the wildcard as punctuation. For example, the following query matches `hello*`, but not `hellothere`:

```
cts:search(fn:doc(), cts:word-query("hello*", "punctuation-sensitive") )
```

If the punctuation sensitivity option is left unspecified, the system performs a punctuation sensitive search if there is any non-wildcard punctuation in the query terms. For example, if punctuation is not specified, the following query:

```
cts:search(fn:doc(), cts:word-query("hello!") )
```

will result in a punctuation sensitive search, and the following query:

```
cts:search(fn:doc(), cts:word-query("hello") )
```

will result in a punctuation insensitive search.

If a search is punctuation sensitive (whether implicitly or explicitly), MarkLogic Server will match the punctuation as well as the search term. Note that punctuation is not considered to be part of a word. For example, `mark!` is considered to be a word `mark` next to an exclamation point. If a search is punctuation insensitive, punctuation will match spaces.

17.3.2.2 Rules for Punctuation and Wildcarding Interaction

The characters `?` and `*` are considered punctuation in documents loaded into the database. However, `?` and `*` are also treated as wildcard characters in a query. This makes for interesting (and occasionally confusing) interaction between wildcarding and punctuation sensitivity.

The following are the rules for the interaction between punctuation and wildcarding. They will help you determine how the system behaves when there are interactions between the punctuation and wildcard characters.

1. When wildcard indexes are disabled in the database, all queries default to `"unwildcarded"`, and wildcard characters are treated as punctuation. If you specify `"wildcarded"` in the query, the query is a wildcard query and wildcard characters are treated as wildcards.

2. Wildcarding trumps (has precedence over) punctuation sensitivity. That is, if the `*` and/or `?` characters are present in a query, `*` and `?` are treated as wildcards and not punctuation unless wildcarding is turned off. If wildcarding is turned off in the query (`"unwildcarded"`), they are treated as punctuation.

3. If wildcarding and punctuation sensitivity are both explicitly off and punctuation characters (including `*` and `?`) are in the query, they are treated as spaces.

4. Wildcarding and punctuation sensitivity can be on at the same time. In this case, punctuation in a document is treated as characters, and wildcards in the query will match any character in the query, including punctuation characters. Therefore, the following query will match both `hello*` and `hellothere`:

```
cts:search(fn:doc(),
           cts:word-query("hello*",
                          ("punctuation-sensitive", "wildcarded") )
           )
```

17.3.2.3 Examples of Wildcard and Punctuation Interactions

This section contains examples of the output of queries in the following categories:

- Wildcarding and Punctuation Sensitivity Not Specified (Wildcard Indexes Enabled)
- Wildcarding Explicitly Off, Punctuation Sensitivity Not Specified
- Wildcarding Not Specified, Punctuation Sensitivity Explicitly On (Wildcard Indexes Enabled)

Wildcarding and Punctuation Sensitivity Not Specified (Wildcard Indexes Enabled)

The following examples show queries that are run when at least one wildcard index is enabled and no options are explicitly set on the `cts:word-query`.

- **Example query:** `cts:word-query("hello world")`

  Actual behavior: Wildcarding off, punctuation insensitive

  Will match: `hello world`, `hello ?! world`, `hello? world!` and so on

- **Example query:** `cts:word-query("hello?world")`

  Actual behavior: Wildcarding on, punctuation insensitive

  Will match: `helloaworld`

  Will not match: `hello world`, `hello!world`

- **Example query:** `cts:word-query("hello*world")`

  Actual behavior: Wildcarding on, punctuation insensitive

  Will match: `helloabcworld`

  Will not match: `hello to world`, `hello-to-world`

- **Example query:** `cts:word-query("hello * world")`

  Actual behavior: Wildcarding on, punctuation insensitive

  Will match: `hello to world`, `hello-to-world`

  Will not match: `helloaworld`, `hello world`, `hello ! world`

  Note: Adjacent spaces are collapsed for string comparisons in the server. In the query phrase `hello * world`, the two spaces on each side of the asterisk are not collapsed for comparison since they are not adjacent to each other. Therefore, `hello world` is not a match since there is only a single space between `hello` and `world` but `hello * world` requires two spaces because the spaces were not collapsed. The phrase `hello ! world` is also not a match because `!` is treated as a space (punctuation insensitive), and then all three consecutive spaces are collapsed to a single space before the string comparison.

- **Example query:** `cts:word-query("hello! world")`

  Actual behavior: Wildcarding off, punctuation sensitive

  Will match: `hello! world`

  Will not match: `hello world`, `hello; world`

- **Example query:** `cts:word-query("hey! world?")`

  Actual behavior: Wildcarding on, punctuation sensitive

  Will match: `hey! world?`, `hey! world!`, `hey! worlds`

  Will not match: `hey. world`

Wildcarding Explicitly Off, Punctuation Sensitivity Not Specified

The following examples show the matches for queries that specify `"unwildcarded"` and do not specify anything about punctuation-sensitivity.

- **Example query:** `cts:word-query("hello?world", "unwildcarded")`

  Actual behavior: Wildcarding off, punctuation sensitive

  Will match: `hello?world`

  Will not match: `hello world`, `hello;world`

- **Example query:** `cts:word-query("hello*world", "unwildcarded")`

  Actual behavior: Wildcarding off, punctuation sensitive

  Will match: `hello*world`

  Will not match: `helloabcworld`

Wildcarding Not Specified, Punctuation Sensitivity Explicitly On (Wildcard Indexes Enabled)

The following examples show queries that are run when at least one wildcard index is enabled and the `"punctuation-sensitive"` option is explicitly set on the `cts:word-query`.

- **Example query:** `cts:word-query("hello?world", "punctuation-sensitive")`

  Actual behavior: Wildcarding on, punctuation sensitive

  Will match: `hello?world`, `hello.world`, `hello*world`

  Will not match: `hello world`, `hello ! world`

- **Example query:** `cts:word-query("hello * world", "punctuation-sensitive")`

  Actual behavior: Wildcarding on, punctuation sensitive

  Will match: `hello abc world`, `hello ! world`

  Will not match: `hello-!- world`

- **Example query:** `cts:word-query("hello? world", "punctuation-sensitive")`

  Actual behavior: Wildcarding on, punctuation sensitive

  Will match: `hello! world`, `(hello) world`

  Note: `(hello) world` is a match because `?` matches `)` and `(` is not considered part of the word `hello`.

  Will not match: `ahello) world`, `hello to world`.

18.0 Collections

MarkLogic Server includes *collections*, which are groups of documents that enable queries to efficiently target subsets of content within a MarkLogic database.

Collections are described as part of the W3C XQuery specification, but their implementation is undefined. MarkLogic has chosen to emphasize collections as a powerful and high-performance mechanism for selecting sets of documents against which queries can be processed. This chapter introduces the `collection()` function, explains how collections are defined and accessed, and describes some of the basic performance characteristics with which developers should be familiar. This chapter includes the following sections:

- The collection() Function
- Collections Versus Directories
- Defining Collections
- Collection Membership
- Collections and Security
- Performance Characteristics

18.1 The collection() Function

The `collection()` function can be used anywhere in your XQuery that the `doc()` or `input()` functions are used. The `collection()` function has the following signature:

```
fn:collection($URI as xs:string*) as node*
```

Note: The MarkLogic Server implementation of the `collection()` function takes a sequence of URIs, so you can call the `collection()` function on one or more collections. The signature of the function in the W3C XQuery documentation only takes a single string. Also, the `fn` namespace is built-in to MarkLogic Server, so it is not necessary to prefix the function with its namespace.

To illustrate what the `collection()` function is used for, consider the following two XPath expressions:

```
fn:doc()//sonnet/line[cts:contains(., "flower")]

collection("english-lit/shakespeare")//sonnet/
                                      line[cts:contains(., "flower")]
```

The first expression returns a sequence of line nodes, each of which must be the child of a sonnet node, and each of which must contain the term *flower*, matched on a case-insensitive basis.

The second expression returns the same sequence, except that only line nodes contained within documents that are members of the `english-lit/shakespeare` collection. MarkLogic Server optimizes this expression. The operation that uses the `collection()` function, along with the rest of the XPath expression, is executed very efficiently through a series of index lookups.

As mentioned previously, the `collection()` function accepts either a single collection, as illustrated above, or a sequence of collections, as illustrated below:

```
collection(("english-lit/shakespeare",
            "american-lit/poetry"))//sonnet/
                                line[cts:contains(., "flower")]
```

The query above returns a sequence of line nodes that match the stated predicates that are members of either the `english-lit/shakespeare` collection or the `american-lit/poetry` collection or both. With this modification to the `collection()` function, its format now closely matches the format of the `doc()` function, which also takes a sequence of URIs. While there is currently no XPath-level support for more complex boolean membership conditions (for example, requiring membership in multiple collections (and), excluding documents that belong to certain collections (not) or requiring pure either-or membership (exclusive or)), you can achieve these conditions through the `where` clause in a surrounding FLWOR expression (see “Collection Membership” on page 245 for an example).

18.2 Collections Versus Directories

Collections are used to organize documents in a database. You can also use directories to organize documents in a database. The key differences in using collections to organize documents versus using directories are:

- Collections do not require member documents to conform to any URI patterns. They are not hierarchical; directories are. Any document can belong to any collection, and any document can also belong to multiple collections.
- You can delete all documents in a collection with the `xdmp:collection-delete` function. Similarly, you can delete all documents in a directory (as well as all recursive subdirectories and any documents in those directories) with the `xdmp:directory-delete` function.
- You cannot set properties on a collection; you can on a directory.

Except for the fact that you can use both collections and directories to organize documents, collections are unrelated to directories. For details on directories, see Properties Documents and Directories in the *Application Developer’s Guide*.

18.3 Defining Collections

Collection membership for a document is defined implicitly. Rather than describing collections top-down (that is, specifying the list of documents that belong to a given collection), MarkLogic Server determines membership in a bottoms-up fashion, by aggregating the set of documents that describe themselves as being a member of the collection. You can use MarkLogic Server's security scheme to manage policies around collection membership.

Collections are named using URIs. Any URI is a legal name for a collection. The URI must be unique within the set of collections (both protected and unprotected) in your database.

The URIs that are used to name collections serve *only* as identifiers to the server. In particular, collections are *not* modeled on filesystem directories. Rather, collections are interpreted as sets, not as hierarchies. A document that belongs to collection `english-lit/poetry/sonnets` need not belong to collection `english-lit/poetry`. In fact, the existence of a collection with URI `english-lit/poetry/sonnets` does not imply the existence of collections with URI `english-lit/poetry` or URI `english-lit`.

There are two types of collections supported by MarkLogic Server: unprotected collections and protected collections. The two types are identical in terms of the syntactic application of the `collection()` function. However, differences emerge in the way they are defined, in who can access the collections, and in who can modify, add or remove documents from them. The following subsections descripe these two ways of defining collection:

- Implicitly Defining Unprotected Collections
- Explicitly Defining Protected Collections

18.3.1 Implicitly Defining Unprotected Collections

Unprotected collections are created *implicitly.*

When a document is first loaded into the system, the load directive (whether through XQuery or XDBC) optionally can specify the collections to which that document belongs. In that list of collections, the specification of a collection URI that has not previously been used is the only action that is needed to create that new unprotected collection.

If collections are left unspecified in the load directive, the document is added to the database with collection membership determined by the default collections that are defined for the current user through the security model and by inheritance from the current user's roles. The invocation of these default settings can also result in the creation of a new unprotected collection. If collections are left unspecified in the load directive and the current user has no default collections defined, the document will be added to the database without belonging to any collections.

In addition, once a document is loaded into the database, you can adjust its membership in collections with any of the following built-in XQuery functions (assuming you possess the appropriate permissions to modify the document in question):

- `xdmp:document-add-collections`
- `xdmp:document-remove-collections`
- `xdmp:document-set-collections`

If a collection URI that is not otherwise used in the database is passed as a parameter to `xdmp:document-add-collections` or `xdmp:document-set-collections`, a new unprotected collection is created.

Unprotected collections disappear when there are no documents in the database that are members. Consequently, using `xdmp:document-remove-collections`, `xdmp:document-set-collections` or `xdmp:document-delete` may result in unprotected collections disappearing.

The `xdmp:collection-delete` function, which deletes every document in a database that belongs to a particular collection (assuming that the current user has the required permissions on a per-document basis), always results in the specified unprotected collection disappearing.

> **Note:** The `xdmp:collection-delete` function will delete all documents in a collection, regardless of their membership in other collections.

18.3.2 Explicitly Defining Protected Collections

Protected collections are created *explicitly.*

Protected collections afford certain security protections not available with unprotected collections (see “Collections and Security” on page 245). Consequently, rather than the implicit model described above, protected collections must be explicitly defined using the Admin Interface *before* any documents are assigned to that collection.

Once a protected collection and its security policies have been defined, documents can be added to that collection through the same mechanisms as described above for unprotected collections. However, in addition to the appropriate permissions to modify the document, the user also needs to have the appropriate permissions to modify the protected collection.

Just as protected collections are created explicitly, the collection does not disappear if the state of the database changes and there are no documents currently belonging to that protected collection. To remove a protected collection from the database, the Admin Interface must be used to delete that collection's definition.

18.4 Collection Membership

As described above, the collections (unprotected and protected) to which a specific document belongs can be specified at load-time and can be modified once the document has been loaded into the database. Documents can belong to many collections simultaneously.

If specific collections are not defined at load-time, the server will automatically assign collection membership for the document based on both the user's and the user's aggregate roles' default collection membership settings. To load a document that does not belong to any collections, explicitly specify the empty sequence as the collections parameter.

Collection membership can be leveraged in any XPath expression that the `collection()`, `doc()`, or `input()` functions are used. In addition, collection membership for a particular document or node can be queried using the `xdmp:document-get-collections` built-in.

For example, the following expression returns a sequence of line nodes, each of which must be the child of a sonnet node, and each of which must contain the term *flower*, matched on a case-insensitive basis, that belong to either the `english-lit/shakespeare` collection or the `american-lit/poetry` collection or both:

```
collection(("english-lit/shakespeare",
            "american-lit/poetry"))//sonnet/
                                line[cts:contains(., "flower")]
```

By contrast, the following expression returns a similar sequence of line nodes, except that the resulting nodes must belong to either the `english-lit/poetry` collection or the `american-lit/poetry` collection or both, but not to the `english-lit/shakespeare` collection:

```
for $line in collection(("english-lit/poetry", "american-lit/
     poetry"))//sonnet/line[cts:contains(., "flower")]
where xdmp:document-get-collections($line) !=
     "english-lit/shakespeare"
return $line
```

18.5 Collections and Security

Collections interact with the MarkLogic Server security model in three basic ways:

- All users and roles can optionally specify default collections. These are the collections to which newly inserted documents are added if collections are not explicitly specified at load-time.
- Adding a document to a collection—both at load-time and after the document has been loaded into the database—is contingent on the user possessing permissions to insert or update the document in question.

- Removing a document from a collection and using `xdmp:collection-delete` are similarly contingent on the user's having appropriate permissions to update the document(s) in question.

Protected collections interact with the MarkLogic Server security model in three additional ways:

- Protected collections must be configured using the security module of the Admin Interface.
- Protected collections specify the roles that have read, insert and/or update permissions for the protected collection.
- Adding or removing documents from protected collections requires not only the appropriate permissions for the documents, but also the appropriate permissions for the collections involved.

18.5.1 Unprotected Collections

To add to the database a new document that belongs to one or more unprotected collections, the user must have (directly or indirectly) the permissions required to add the document. This means that the user must either possess the admin role or have both of the following:

- The privilege to execute the `xdmp:document-load` function, if that is the document insertion directive being used.
- Either the `unprotected-uri` privilege, the `any-uri` privilege, or an appropriate URI privilege on the specific path of the document to be inserted. For example, if the document being inserted has the URI `/docs/poetry/love.xml`, the appropriate URI privileges are `/`, `/docs`, `/docs/poetry`.

To modify the set of collections to which a document belongs, the user must either possess the admin role or have update permissions on the document.

To access an unprotected collection in an XPath expression, no special permissions are used. Access to each of the individual documents that belong to the specified collection is governed by that individual document's read permissions.

18.5.2 Protected Collections

To add to the database a new document that belongs to one or more protected collections, the user must have (directly or indirectly) the permissions required to add the document as well as the permissions required to add to the protected collection(s). This means that the user must either possess the admin role or have all of the following:

- The insert permission on the protected collection.
- The privilege to execute the `xdmp:document-load` function, if that is the document insertion directive being used.
- Either the `unprotected-uri` privilege, the `any-uri` privilege, or an appropriate URI privilege on the specific path of the document to be inserted. For example, if the document being inserted has the URI `/docs/poetry/love.xml`, the appropriate URI privileges are `/`, `/docs`, `/docs/poetry`.

To modify the set of protected collections to which a document belongs, the user must either possess the admin role or have:

- Update permissions on the collection
- Update permissions on the document

To access a protected collection in an XPath expression, the user must have read permissions on the collection. In addition, access to each of the individual documents that belong to the specified collection is governed by that individual document's read permissions. Note that access to the documents themselves (as opposed to membership in the collection) is governed by the current user's roles and the permissions associated with each document.

The user can convert an unprotected collection into a protected collection using the Security Function Library module `sec:protect-collection`. Access to this library module is dependent on the user's having the `protect-collection` privilege.

The user can convert a protected collection into an unprotected collection using the Security Function Library module `sec:unprotect-collection`. Access to this library module is dependent on the user's having the `unprotect-collection` privilege and update permissions on the protected collection.

18.6 Performance Characteristics

MarkLogic's implementation of collections is designed to optimize query performance against large volumes of documents. As with all designs, the implementation involves some trade-offs. This section provides a brief overview of the performance characteristics of collections and includes the following subsections:

- Number of Collections to Which a Document Belongs
- Adding/Removing Existing Documents To/From Collections

18.6.1 Number of Collections to Which a Document Belongs

At document load time, collection information is embedded into the document and stored in the database.

This design enables a MarkLogic database to handle millions of collections without difficulty. It also enables the `collection()` function itself to be extremely efficient, able to subset large datasets by collection with a single index operation. If the `collection()` function specifies more than one collection, an additional index operation is required for each collection specified. Assuming queries target similar collections, these index operations should be resolved within cache at extremely high performance.

One trade-off with this design is a practical constraint on the number of collections to which a single document should belong. While there is no architectural limit, the size of the database will grow as the average number of collections per document increases. This database growth is driven by an increase in the size of individual document fragments. The fragment size increases because each collection to which the document belongs embeds a small amount of information in the fragment. As fragments grow, the corresponding storage I/O time increases, resulting in performance degradation. It is important to note that the average number of collections per document does not impact *index resolution* time, merely the time to retrieve the content (fragments) from storage.

A practical guideline is that a document with fragments averaging 50K in size should not belong to more than 100 collections. This should keep the average fragment size increase to less than 10%.

18.6.2 Adding/Removing Existing Documents To/From Collections

A second trade-off with MarkLogic's implementation of collections is that adding or removing documents from collections once those documents are already in the database can be relatively resource-intensive. Changing the collections to which a document belongs requires rewriting every fragment of the document. For large documents, this can be demanding on both CPU and I/O resources. If collection membership is highly dynamic in your application, a better approach may be to use elements within the document itself to characterize membership.

19.0 Using the Thesaurus Functions

MarkLogic Server includes functions that enable applications to provide thesaurus capabilities. Thesaurus applications use thesaurus (synonym) documents to find words with similar meaning to the words entered by a user. A common example application expands a user search to include words with similar meaning to those entered in a search. For example, if the application uses a thesaurus document that lists car brands as synonyms for the word *car*, then a search for car might return results for *Alfa Romeo*, *Ford*, and *Hyundai*, as well as for the word *car*.

This chapter describes how to use the thesaurus functions and contains the following sections:

- The Thesaurus Module
- Function Reference
- Thesaurus Schema
- Capitalization
- Managing Thesaurus Documents
- Expanding Searches Using a Thesaurus

19.1 The Thesaurus Module

The thesaurus functions are installed into the following XQuery module file:

- *install_dir*/Modules/MarkLogic/thesaurus.xqy

where *install_dir* is the directory in which MarkLogic Server is installed. The functions in the thesaurus module use the `thsr:` namespace prefix, which you must specify in your XQuery program (or specify your own namespace). To use any of the functions, include the module and namespace declaration in the prolog of your XQuery program as follows:

```
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";
```

19.2 Function Reference

The reference information for the thesaurus module functions is included in the *MarkLogic XQuery and XSLT Function Reference* available through developer.marklogic.com.

19.3 Thesaurus Schema

Any thesaurus documents loaded into MarkLogic Server must conform to the thesaurus schema, installed into the following file:

- *install_dir*/Config/thesaurus.xsd

where *install_dir* is the directory in which MarkLogic Server is installed.

19.4 Capitalization

Thesaurus documents and the thesaurus functions are case-sensitive. Therefore, a thesaurus term for *Car* is different from a thesaurus term for *car* and any lookups for these terms are case-sensitive.

If you want your applications to be case-insensitive (that is, if you want the term *Car* to return thesaurus entries for both *Car* and *car*), your application must handle the case of the terms you want to lookup. There are several ways to handle case. For example, you can lowercase all the entries in your thesaurus documents and then lowercase the terms before performing the lookup from the thesaurus. For an example of lowercasing terms in a thesaurus document, see "Lowercasing Terms When Inserting a Thesaurus Document" on page 251.

19.5 Managing Thesaurus Documents

You can have any number of thesaurus documents in a database. You can also add to or modify any thesaurus documents that already exist. This section describes how to load and update thesaurus documents, and contains the following sections:

- Loading Thesaurus Documents
- Lowercasing Terms When Inserting a Thesaurus Document
- Loading the XML Version of the WordNet Thesaurus
- Updating a Thesaurus Document
- Security Considerations With Thesaurus Documents
- Example Queries Using Thesaurus Management Functions

19.5.1 Loading Thesaurus Documents

To use a thesaurus in a query, use the `thsr:load` function or the `thsr:insert` function to load a document as a thesaurus. For example, to load a thesaurus document with a URI `/myThsrDocs/wordnet.xml`, execute a query similar to the following:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:load("c:\thesaurus\wordnet.xml", "/myThsrDocs/wordnet.xml")
```

This XQuery adds all of the `<entry>` elements from the `c:\thesaurus\wordnet.xml` file to a thesaurus with the URI `/myThsrDocs/wordnet.xml`. If the document already exists, then it is overwritten with the new content from the specified file.

Note: If you have a thesaurus document that is too large to fit into an in-memory list, you can split the thesaurus into multiple documents. If you do this, you must specify all of the thesaurus documents in the thesaurus APIs that take URIs as a parameter. Also, ensure that there are no duplicate entries between the different thesaurus documents.

19.5.2 Lowercasing Terms When Inserting a Thesaurus Document

You can use the `thsr:insert` function to perform transformation on a document before inserting it as a thesaurus document. The following example shows how you can use the `xdmp:get` function to load a document into memory, then walk through the in-memory document and construct a new document which has lowercase terms.

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                              at "/MarkLogic/thesaurus.xqy";

thsr:insert("newThsr.xml",
     let $thsrMem := xdmp:get("C:\myFiles\thesaurus.xml")
     return
<thesaurus xmlns="http://marklogic.com/xdmp/thesaurus">
{
   for $entry in $thsrMem/thsr:entry
   return
       (: Write out and lowercase the term, then write out all of
          the children of this entry except for the term, which was
          already written out and lowercased :)
     <thsr:entry>
       <thsr:term>{lower-case($entry/thsr:term)}</thsr:term>
          {$entry/*[. ne $entry/thsr:term]}
      </thsr:entry>
}
</thesaurus>
            )
```

19.5.3 Loading the XML Version of the WordNet Thesaurus

You can download an XML version of the *WordNet* from the MarkLogic Developer site (developer.marklogic.com). Once you download the thesaurus file, you can load it as a thesaurus document using the `thsr:load` function.

Perform the following steps to download and load the *WordNet Thesaurus*:

1. Go to the *Workshop* page of developer.marklogic.com:

   ```
   http://developer.marklogic.com/code/default.xqy
   ```

2. Navigate to the thesaurus document section and find the `thesaurus.xml` document.

3. Save `thesaurus.xml` to a file (for example, `c:\thesaurus\thesaurus.xml`).

4. Load the thesaurus with a command similar to the following:

   ```
   xquery version "1.0-ml";
   import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                               at "/MarkLogic/thesaurus.xqy";

   thsr:load("c:\thesaurus\thesaurus.xml", "/myThsrDocs/wordnet.xml")
   ```

This loads the thesaurus with a URI of `/myThsrDocs/wordnet.xml`. You can now use this URI with the thesaurus module functions.

19.5.4 Updating a Thesaurus Document

Use the following thesaurus functions to modify existing thesaurus documents:

- `thsr:set-entry`
- `thsr:add-synonym`
- `thsr:remove-entry`
- `thsr:remove-term`
- `thsr:remove-synonym`

Additionally, the `thsr:insert` function adds entries to an existing thesaurus document (as well as creates a new one if one does not exist at the specified URI).

> **Note:** The transactional unit in MarkLogic Server is a query; therefore, if you are performing multiple updates to the same thesaurus document, be sure to perform those updates as part of separate queries. You can place a semi-colon between the update statements to start a new query (and therefore a new transaction). If you use a semicolon to start any new queries that uses thesaurus functions, each query must include the `import` statement in the prolog to resolve the thesaurus namespace.

19.5.5 Security Considerations With Thesaurus Documents

Thesaurus documents are stored in XML format in the database. Therefore, they can be queried just like any other document. Note the following about security and thesaurus documents:

- By default, thesaurus documents are loaded into the following collections:
  - http://marklogic.com/xdmp/documents
  - http://marklogic.com/xdmp/thesaurus
- Thesaurus documents are loaded with the default permissions of the user who loads them. Make sure users who load thesaurus documents have approriate privileges, otherwise the documents might not have the needed permissions for reading and updating. For more information, see Setting Document Permissions in the *Loading Content Into MarkLogic Server Guide*.
- If you want to control access (read and/or write) to thesaurus documents beyond the default permissions with which the documents are loaded, perform an `xdmp:document-set-permissions` after a `thsr:load` operation.

19.5.6 Example Queries Using Thesaurus Management Functions

This section includes the following examples:

- Example: Adding a New Thesaurus Entry
- Example: Removing a Thesaurus Entry
- Example: Removing Term(s) from a Thesaurus
- Example: Adding a Synonym to a Thesaurus Entry
- Example: Removing a Synonym From a Thesaurus

19.5.6.1 Example: Adding a New Thesaurus Entry

The following XQuery uses the `thsr:set-entry` function to add an entry for *Car* to the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:set-entry("/myThsrDocs/wordnet.xml",
<entry xmlns="http://marklogic.com/xdmp/thesaurus">
    <term>Car</term>
    <synonym>
      <term>Ford</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
    <synonym>
      <term>automobile</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
    <synonym>
      <term>Fiat</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
</entry>)
```

If the `/myThsrDocs/wordnet.xml` thesaurus has an identical entry, there will be no change to the thesaurus. If the thesaurus has no entry for *car* or has an entry for *car* that is not identical (that is, where the nodes are not equivalent), it will add the new entry. The new entry is added to the end of the thesaurus document.

19.5.6.2 Example: Removing a Thesaurus Entry

The following XQuery uses the `thsr:remove-entry` function to remove the entry for *Car* from the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:remove-entry("/myThsrDocs/wordnet.xml",
                  thsr:lookup("/myThsrDocs/wordnet.xml","Car")[2])
```

This removes the second entry for *Car* from the `/myThsrDocs/wordnet.xml` thesaurus document.

19.5.6.3 Example: Removing Term(s) from a Thesaurus

The following XQuery uses the `thsr:remove-term` function to remove all entries for the term *Car* from the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:remove-term("/myThsrDocs/wordnet.xml", "Car")
```

This removes all of the *Car* terms from the `/myThsrDocs/wordnet.xml` thesaurus document. If you only have a single term for *Car* in the thesaurus, the `thsr:remove-term` function does the same as the `thsr:remove-entry` function.

19.5.6.4 Example: Adding a Synonym to a Thesaurus Entry

The following XQuery adds the synonym *Alfa Romeo* to the thesaurus entry for *car* in the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:add-synonym(thsr:lookup("/myThsrDocs/wordnet.xml", "car"),
                <thsr:synonym>
                   <thsr:term>Alfa Romeo</thsr:term>
                </thsr:synonym>)
```

This query assumes that the lookup for the *car* thesaurus entry returns a single entry. If the *car* lookup returns multiple entries, you must specify a single entry. For example, if you wanted to add the synonym to the first *car* entry in the thesaurus, specify the first argument as follows:

```
thsr:lookup("/myThsrDocs/wordnet.xml", "car")[1]
```

19.5.6.5 Example: Removing a Synonym From a Thesaurus

The following XQuery removes the synonym *Fiat* from the thesaurus entry for *car* in the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";

thsr:remove-synonym(thsr:lookup("/myThsrDocs/wordnet.xml", "car"),
                <thsr:synonym>
                   <thsr:term>Fiat</thsr:term>
                </thsr:synonym>)
```

This query assumes that the lookup for the *car* thesaurus entry returns a single entry. If the *car* lookup returns multiple entries, you must specify a single entry. For example, if you wanted to remove the synonym from the first *car* entry in the thesaurus, specify the first argument as follows:

```
thsr:lookup("/myThsrDocs/wordnet.xml", "car")[1]
```

19.6 Expanding Searches Using a Thesaurus

You can expand a search to include terms from a thesaurus as well as the terms entered in the search. Consider the following query:

```
xquery version "1.0-ml";
import module namespace thsr="http://marklogic.com/xdmp/thesaurus" at
"/MarkLogic/thesaurus.xqy";

cts:search(
doc("/Docs/hamlet.xml")//LINE,
thsr:expand(
  cts:word-query("weary"),
  thsr:lookup("/myThsrDocs/thesaurus.xml", "weary"),
  (),
  (),
  () )
)
```

This query finds all of the lines in Shakespeare's *Hamlet* that have the word *weary* or any of the synonyms of the word *weary.*

Thesaurus entries can have many synonyms, though. Therefore, when you expand a search, you might want to create a user interface in the application which provides a form allowing a user to specify the desired synonyms from the list returned by `thsr:expand`. Once the user chooses which synonyms to include in the search, the application can add those terms to the search and submit it to the database.

20.0 Using the Spelling Correction Functions

MarkLogic Server includes functions that enable applications to provide spelling capabilities. Spelling applications use dictionary documents to find possible misspellings for words entered by a user. A common example application will prompt a user for words that might be misspelled. For example, if a user enters a search for the word *albetros*, an application that uses the spelling correction functions might prompt the user if she means *albatross*.

This chapter describes how to use the spelling correction functions and contains the following sections:

- Overview of Spelling Correction
- The Spelling Dictionary Management Module Functions
- Function Reference
- Dictionary Documents
- Capitalization
- Managing Dictionary Documents
- Testing if a Word is Spelled Correctly
- Getting Spelling Suggestions for Incorrectly Spelled Words

20.1 Overview of Spelling Correction

The spelling correction functions enable you to create applications that check if words are spelled correctly. It uses one or more dictionaries that you load into the database and checks words against a dictionary you specify. You can control everything about what words are in the dictionary. There are functions to manage the dictionaries, check spelling, and suggest words for misspellings.

20.2 Function Reference

The reference information for the spelling module functions is included in the *MarkLogic XQuery and XSLT Function Reference* available through developer.marklogic.com. The spelling functions are divided into the following categories:

- The Spelling Built-In Functions
- The Spelling Dictionary Management Module Functions

20.2.1 The Spelling Built-In Functions

The spelling correction functions are built-in functions and do not require the `import module` statement in the XQuery prolog. The following are the spelling correction functions:

- `spell:is-correct`
- `spell:suggest`
- `spell:suggest-detailed`
- `spell:double-metaphone`
- `spell:levenshtein-distance`

The `spell:double-metaphone` and `spell:levenshtein-distance` functions return the raw values from which `spell:suggest`, `spell:suggest-detailed`, and `spell:is-correct` calculate their values.

The difference between `spell:suggest` and `spell:suggest-detailed` is that `spell:suggest-detailed` provides some of the information used in calculating the suggestions, and it returns in in an XML representaiton (whereas `spell:suggest` returns a sequence of suggested words). For most spelling applications, `spell:suggest` is sufficient, but if you want finer control of the suggestions you provide (for example, if you want to calculate your own order of returning the suggestions), you can use `spell:suggest-detailed` and then filter on some of the criteria returned in its XML output.

20.2.2 The Spelling Dictionary Management Module Functions

There is an XQuery module to perform management of dictionary documents. The spelling correction dictionary management functions are installed into the following XQuery module file:

- *`install_dir`*`/Modules/MarkLogic/spell.xqy`

where *`install_dir`* is the directory in which MarkLogic Server is installed. The functions in the spelling module use the `spell:` namespace prefix, which is predefined in the server. To use the functions in this module, include the module declaration in the prolog of your XQuery program as follows:

```
import module "http://marklogic.com/xdmp/spell" at "/MarkLogic/spell.xqy";
```

20.3 Dictionary Documents

Any dictionary documents loaded into MarkLogic Server must have the following basic structure:

```
<dictionary xmlns="http://marklogic.com/xdmp/spell">
    <metadata>
    </metadata>
    <word></word>
    <word></word>
    ......
</dictionary>
```

There are sample dictionary documents available on developer.marklogic.com. You can use these documents or create your own dictionaries. You can also use the `spell:make-dictionary` spelling management function to create a dictionary document, and then use `spell:load` to load the dictionary into the database.

20.4 Capitalization

The spelling lookup functions (`spell:is-correct`, `spell:suggest`, and `spell:suggest-detailed`) are case-sensitive, so case is important for words in a dictionary. Additionally, there are some special rules to handle the first character in a spelling lookup. The following are the capitalization rules for the spelling correction functions:

- A capital first letter in a spelling lookup query does not make the spelling incorrect for `spell:is-correct`. For example, *Word* will match an entry for *word* in the dictionary.
- If a word has the first letter capitalized in the dictionary, then only exact matches will be correct for `spell:is-correct`. For example, if *Word* is in the dictionary, then *word* is incorrect.
- If a word has other letters capitalized in the dictionary, then only exact matches (or exact matches except for the case of the first letter in the word) will match for `spell:is-correct`.. For example, *word* will not match an entry for *woRd*, nor will *WOrd*, but *WoRd* will match.
- The `spell:suggest` function (and the `spell:suggest-detailed` function) observes the capitalization of the first letter only. For example, `spell:suggest("THe")` will return *The*, *Thee*, *They*, and so on as suggestions, while `spell:suggest("tHe")` will give *the*, *thee*, *they*, and so on. In other words, if you capitalize the first letter of the argument to the `spell:suggest` function, the suggestions will all begin with a capital letter. Otherwise, you will get lowercase suggestions.

If you want your applications to ignore case, then you should create a dictionary with all lowercase words and lowercase (using the XQuery `fn:lower-case` function, for example) the word arguments of all `spell:is-correct` and `spell:suggest` functions before submitting your queries.

20.5 Managing Dictionary Documents

You can have any number of dictionary documents in a database. You can also add to or modify any dictionary documents that already exist. This section describes how to load and update dictionary documents, and contains the following topics:

- Loading Dictionary Documents
- Loading one of the Sample XML Dictionaries
- Updating a Dictionary Document
- Security Considerations With Dictionary Documents

20.5.1 Loading Dictionary Documents

To use a dictionary in a query, use the `spell:load` function or the `spell:insert` function to load a document as a dictionary. For example, to load a dictionary document with a URI `/mySpell/spell.xml`, execute a query similar to the following:

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at "/MarkLogic/spell.xqy";

spell:load("c:\dictionaries\spell.xml", "/mySpell/spell.xml")
```

This XQuery adds all of the `<word>` elements from the `c:\dictionaries\spell.xml` file to a dictionary with the URI `/mySpell/spell.xml`. If the document already exists, then it is overwritten with the new content from the specified file.

20.5.2 Loading one of the Sample XML Dictionaries

You can download a sample dictionary from the MarkLogic Community site (developer.marklogic.com/code#dictionaries). The community site links to github, which has small, medium, and large versions of the dictionary. Once you download a dictionary XML file, you can load it as a dictionary document using the `spell:load` function.

Perform the following steps to download and load a sample dictionary:

1. Go to the *Code* page of developer.marklogic.com:

   http://developer.marklogic.com/code/#dictionaries

2. Navigate to the dictionary document section, then click the github link:

   `https://github.com/marklogic/dictionaries`

3. In the dictionaries folder, choose the `small-dictionary.xml`, `medium-dictionary.xml`, or `large-dictionary.xml` file (or any other dictionary docments that might be available). The large dictionary has approximately 100,000 words and is about 3 MB to download.

4. Save `<size>-dictionary.xml` to a file (for example, `c:\dictionaries\spell.xml`).

5. Load the dictionary with a command similar to the following:

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at
              "/MarkLogic/spell.xqy";

spell:load("c:\dictionaries\spell.xml", "/mySpell/spell.xml")
```

This loads the dictionary with a URI of `/mySpell/spell.xml`. You can now use this URI with the spelling correction module functions.

20.5.3 Updating a Dictionary Document

Use the following dictionary functions to modify existing dictionary documents:

- `spell:add-word`
- `spell:remove-word`

The `spell:insert` function will overwrite an existing dictionary if you specify an existing dictionary document (as well as creates a new one if one does not exist at the specified URI).

Note: The transactional unit in MarkLogic Server is a query; therefore, if you are performing multiple updates to the same dictionary document, be sure to perform those updates as part of separate queries. You can place a semi-colon between the update statements to start a new query (and therefore a new transaction). If you use a semicolon to start any new queries that uses spelling correction functions, each query must include the `import` statement in the prolog to resolve the spelling module.

20.5.3.1 Example: Adding a New Word to a Dictionary

The following XQuery uses the `spell:add-word` function to add an entry for *albatross* to the dictionary with URI `/mySpell/Spell.xml`:

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at
              "/MarkLogic/spell.xqy";

spell:add-word("/mySpell/spell.xml", "albatross")
```

If the `/mySpell/spell.xml` dictionary has an identical entry, there will be no change to the dictionary. Otherwise, an entry for *albatross* is added to the dictionary.

20.5.3.2 Example: Removing a Word From a Dictionary

The following XQuery uses the `spell:remove-word` function to remove the entry for *albatross* dictionary with URI `/mySpell/Spell.xml`:

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at
              "/MarkLogic/spell.xqy";

spell:remove-word("/mySpell/spell.xml", "albatross")
```

This removes the word *albatross* from the `/mySpell/spell.xml` dictionary document.

20.5.4 Security Considerations With Dictionary Documents

Dictionary documents are stored in XML format in the database. Therefore, they can be queried just like any other document. Note the following about security and dictionary documents:

- By default, dictionary documents are loaded into the following collections:
  - `http://marklogic.com/xdmp/documents`
  - `http://marklogic.com/xdmp/spell`
- Dictionary documents are loaded with the default permissions of the user who loads them. Make sure users who load dictionary documents have appropriate privileges, otherwise the documents might not have the needed permissions for reading and updating. For more information, see Setting Document Permissions in the *Loading Content Into MarkLogic Server Guide*.
- If you want to control access (read and/or write) to dictionary documents beyond the default permissions with which the documents are loaded, perform an `xdmp:document-set-permissions` after a `spell:load` operation.

20.6 Testing if a Word is Spelled Correctly

You can use the `spell:is-correct` function test to see if a word is spelled correctly (according to the specified dictionary). Consider the following query:

```
spell:is-correct("/mySpell/spell.xml", "alphabet")
```

This query returns true because the word alphabet is spelled correctly. Now consider the following query:

```
spell:is-correct("/mySpell/spell.xml", "alfabet")
```

This query returns `false` because the word *alfabet* is not spelled correctly.

20.7 Getting Spelling Suggestions for Incorrectly Spelled Words

You can write a query which returns spelling suggestions based on words in the specified dictionary. Consider the following query:

```
spell:suggest("/mySpell/spell.xml", "alfabet")
```

This query returns the following results:

```
alphabet albeit alphabets aloft abet alphabeted affable alphabet's
alphabetic offbeat
```

The results are ranked in the order, where the first word is the one most likely to be the real spelling. Your application can then prompt the user if one of the suggested words was the actual word intended.

Now consider the following query:

```
spell:suggest("/mySpell/spell.xml", "alphabet")
```

This query returns the empty sequence, indicating that the word is spelled correctly.

Note: The spelling correction functions only provide suggestions for words that are less than 64 characters in length, and the functions only return suggestions that are less than 64 characters.

21.0 Distinctive Terms and cts:similar-query

MarkLogic Server includes `cts:similar-query` and `cts:distinctive-terms`. With these search APIs, you can find what is distinctive about nodes, typically from search results, from a search perspective. This chapter describes `cts:similar-query` and `cts:distinctive-terms`, and includes the following sections:

- Understanding cts:similar-query
- Finding the Distinctive Terms of a Set of Nodes
- Understanding the cts:distinctive-terms Output
- Example Design Pattern: Making a Tag Cloud

21.1 Understanding cts:similar-query

You can use `cts:similar-query` to find nodes that are similar, from a search perspecitve, to the model nodes that you pass into the first parameter. The `cts:similar-query` constructor is a `cts:query` constructor, and you can combine it with other `cts:query` constructors as described in “Composing cts:query Expressions” on page 121.

Instead of looking in the indexes to find the terms that match the query, like other `cts:query` constructors, `cts:similar-query` takes the nodes passed in, runs them through an indexing process, and returns a `cts:query` that would match the model nodes with a high degree of relevance. You can pass various index and score options into `cts:similar-query` to influence the `cts:query` that it produces.

The query that it generates finds distinctive terms of the model nodes *based on the other documents in the database*.

21.2 Finding the Distinctive Terms of a Set of Nodes

If you want to find the terms that `cts:similar-query` uses to generate its `cts:query`, you can use `cts:distinctive-terms`. The output of `cts:distinctive-terms` is a `cts:class` element with several `cts:term` children. Each `cts:term` element contains a cts:query constructor, representing a term. Each `cts:term` element also contains scores and confidence for that term. MarkLogic Server uses these scores in calculating relevance.

You can pass many different options into `cts:distinctive-terms` to control which terms it generates. The database options control which terms will be most “relevant” to the model nodes, and therefore affect the `cts:distinctive-terms` output. If you take an iterative approach, you can try different indexing options to see which ones give the best results for your model nodes.

The distinctive terms generated or distinctive based on the other documents in the database, therfore, you will get much better results running this against a sizable database.

21.3 Understanding the cts:distinctive-terms Output

The following shows a simple `cts:distinctive-terms` query with its output:

```
let $node := doc("/shakespeare/plays/hamlet.xml")
return cts:distinctive-terms($node,
   <options xmlns="cts:distinctive-terms"
         xmlns:db="http://marklogic.com/xdmp/database">
    <use-db-config>false</use-db-config>
    <max-terms>3</max-terms>
    <db:word-searches>false</db:word-searches>
    <db:stemmed-searches>basic</db:stemmed-searches>
    <db:fast-phrase-searches>false</db:fast-phrase-searches>
    <db:fast-element-word-searches>false</db:fast-element-word-searches>
    <db:fast-element-phrase-searches>false</db:fast-element-phrase-searches>
   </options>)
=>
<cts:class name="dterms /shakespeare/plays/hamlet.xml" offset="0"
xmlns:cts="http://marklogic.com/cts">
  <cts:term id="7783238741996929314" val="981" score="981"
confidence="0.811494" fitness="1">
    <cts:word-query>
      <cts:text xml:lang="en">guildenstern</cts:text>
      <cts:option>case-insensitive</cts:option>
      <cts:option>diacritic-insensitive</cts:option>
      <cts:option>stemmed</cts:option>
      <cts:option>unwildcarded</cts:option>
    </cts:word-query>
  </cts:term>
  <cts:term id="4731147985682913359" val="956" score="956"
confidence="0.801087" fitness="1">
    <cts:word-query>
      <cts:text xml:lang="en">polonius</cts:text>
      <cts:option>case-insensitive</cts:option>
      <cts:option>diacritic-insensitive</cts:option>
      <cts:option>stemmed</cts:option>
      <cts:option>unwildcarded</cts:option>
    </cts:word-query>
  </cts:term>
  <cts:term id="1100490632300558572" val="949" score="949"
confidence="0.798149" fitness="1">
    <cts:word-query>
      <cts:text xml:lang="en">horatio</cts:text>
      <cts:option>case-insensitive</cts:option>
      <cts:option>diacritic-insensitive</cts:option>
      <cts:option>stemmed</cts:option>
      <cts:option>unwildcarded</cts:option>
    </cts:word-query>
  </cts:term>
</cts:class>
```

The output is a `cts:class` element, and each child is a `cts:term` element. The `cts:term` elements represent terms in a database, identified by a `cts:query`. Each term has numbers for `val`, `score`, `confidence`, and `fitness`.

The `val` and `score` attributes are values that approximate the score contribution of that term. The `confidence` attribute represents the `cts:confidence` value for the term. The `fitness` attribute represents the `cts:fitness` value for the term. For details on score, fitness, and confidence, see "Relevance Scores: Understanding and Customizing" on page 136.

The previous query only consider word-query terms. You can also have `cts:element-word-query` terms and `cts:near-query` terms for terms that are within an element or that are a word pair (a `cts:near-query` with a `distance` of 1). To see some of these kind of terms, try running a query like the following:

```
let $node := doc("/shakespeare/plays/hamlet.xml")
return cts:distinctive-terms($node,
   <options xmlns="cts:distinctive-terms"
         xmlns:db="http://marklogic.com/xdmp/database">
    <use-db-config>false</use-db-config>
    <max-terms>100</max-terms>
    <db:word-searches>false</db:word-searches>
    <db:stemmed-searches>basic</db:stemmed-searches>
    <db:fast-phrase-searches>true</db:fast-phrase-searches>
    <db:fast-element-word-searches>true</db:fast-element-word-searches>
    <db:fast-element-phrase-searches>true</db:fast-element-phrase-searches>
   </options>)
```

This query enables the `db:fast-element-word-searches` and `db:fast-element-phrase-searches` options, which will cause terms to appear in the output that are constrained to a particular element. Changing the database options to `cts:distictive-terms` and looking at the differences in the output will help you to understand both how the index options affect which terms are distinctive and, since `cts:similar-query` can use these same settings, how `cts:similar-query` decides if a document is "similar" to the model nodes.

21.4 Example Design Pattern: Making a Tag Cloud

Tag clouds are a popular visualization that show various terms, usually relevant to a search, and show the more relevant ones in a larger and/or more colorful font. You can use `cts:distinctive-terms` feed the data used to make a tag cloud. The basic design pattern is as follows:

- Experiment with options to create a `cts:distinctive-terms` query that produces results you are happy with.
- Set a `max-terms` size that is equal to the number of terms you want in your tag cloud.
- Come up with some algorithm to convert score (or fitness) into font size. For example, you might want to take the fitness and multiply it by 20 to get a font size.

- Use the above algorithm to iterate through your results and generate some html that creates a tag cloud.

The following sample code is a simplied example of this design pattern:

```
let $hits :=
  let $terms :=
   let $node := doc("/shakespeare/plays/hamlet.xml")//LINE
   return cts:distinctive-terms($node,
   <options xmlns="cts:distinctive-terms"
         xmlns:db="http://marklogic.com/xdmp/database">
    <use-db-config>false</use-db-config>
    <max-terms>100</max-terms>
    <db:word-searches>false</db:word-searches>
    <db:stemmed-searches>basic</db:stemmed-searches>
    <db:fast-phrase-searches>false</db:fast-phrase-searches>
    <db:fast-element-word-searches>false</db:fast-element-word-searches>
    <db:fast-element-phrase-searches>false</db:fast-element-phrase-searches>
   </options>)//cts:term
  for $wq in $terms
  where $wq/cts:word-query
  return element word {
           attribute score {
             fn:round(($wq/@val div 20))},
           $wq/cts:word-query/cts:text/string() }
return <p>{
for $hit in $hits
order by $hit/string()
return (
<span style={fn:concat("font-size: ",
          $hit/@score)}>{$hit/string()}
</span>, " " ) }</p>
```

The above query returns html which, when displayed in a browser, shows the 100 most distinctive with the most “relevant” terms in a larger font.

22.0 Training the Classifier

MarkLogic Server includes an XML support vector machine (SVM) classifier. This chapter describes the classifier and how to use it on your content, and includes the following sections:

- Understanding How Training and Classification Works
- Classifier API
- Leveraging XML With the Classifier
- Creating a Training Set
- Methodology For Determining Thresholds For Each Class
- Example: Training and Running the Classifier

22.1 Understanding How Training and Classification Works

The *classifier* is a set of APIs that allow you to define *classes*, or categories of nodes. By running samples of classes through the classifier to train it on what constitutes a given class, you can then run that trained classifier on unknown documents or nodes to determine to which classes each belongs. The process of classification uses the full-text indexing capabilities of MarkLogic Server, as well as its XML-awareness, to perform statistical analysis of terms in the training content to determine class membership. This section describes the concepts behind the classifier and includes the following parts:

- Training and Classification
- XML SVM Classifier
- Hyper-Planes and Thresholds for Classes
- Training Content for the Classifier

22.1.1 Training and Classification

There are two basic steps to using the classifier: training and classification. *Training* is the process of taking content that is known to belong to specified classes and creating a classifier on the basis of that known content. *Classification* is the process of taking a classifier built with such a training content set and running it on unknown content to determine class membership for the unknown content. Training is an iterative process whereby you build the best classifier possible, and classification is a one-time process designed to run on unknown content.

22.1.2 XML SVM Classifier

The MarkLogic Server classifier implements a support vector machine (SVM). An SVM classifier uses a well-known algorithm to determine membership in a given class, based on training data. For background on the mathematics behind support vector machine (SVM) classifiers, try doing a web search for `svm classifier`, or start by looking at the information on Wikipedia.

The basic idea is that the classifier takes a set of training content representing known examples of classes and, by performing statistical analysis of the training content, uses the knowledge gleaned from the training content to decide to which classes other unknown content belongs. You can use the classifier to gain knowledge about your content based on the statistical analysis performed during training.

Traditional SVM classifiers perform the statistical analysis using term frequency as input to the support vector machine calculations. The MarkLogic XML SVM classifier takes advantage of MarkLogic Server's XML-aware full-text indexing capabilities, so the terms that act as input to the classifier can include content (for example, words), structure information (for example, elements), or a combination of content and structure (for example, element-word relationships). All of the MarkLogic Server index options that affect terms are available as options in the classifier API, so you can use a wide variety of indexing techniques to tune the classifier to work the best for your sample content.

First you define your classes on a set of training content, and then the classifier uses those classes to analyze other content and determine its classification. When the classifier analyzes the content, there are two sometimes conflicting measurements it uses to help determine if the information in the new content belongs in or out of a class:

- *Precision*: The probability that what is classified as being in a class is actually in that class. High precision might come at the expense of missing some results whose terms resemble those of other results in other classes.
- *Recall*: The probability that an item actually in a class is classified as being in that class. High recall might come at the expense of including results from other classes whose terms resemble those of results in the target class.

When you are tuning your classifier, you need to find a balance between high precision and high recall. That balance depends on what your application goals and requirements are. For example, if you are trying to find trends in your content, then high precision is probably more important; you want to ensure that your analysis does not include irrelevant nodes. If you need to identify every instance of some classification, however, you probably need a high recall, as missing any members would go against your application goals. For most applications, you probably need somewhere in between. The process of training your classifier is where you determine the optimal values (based on your training content set) to make the trade-offs that make sense to your application.

22.1.3 Hyper-Planes and Thresholds for Classes

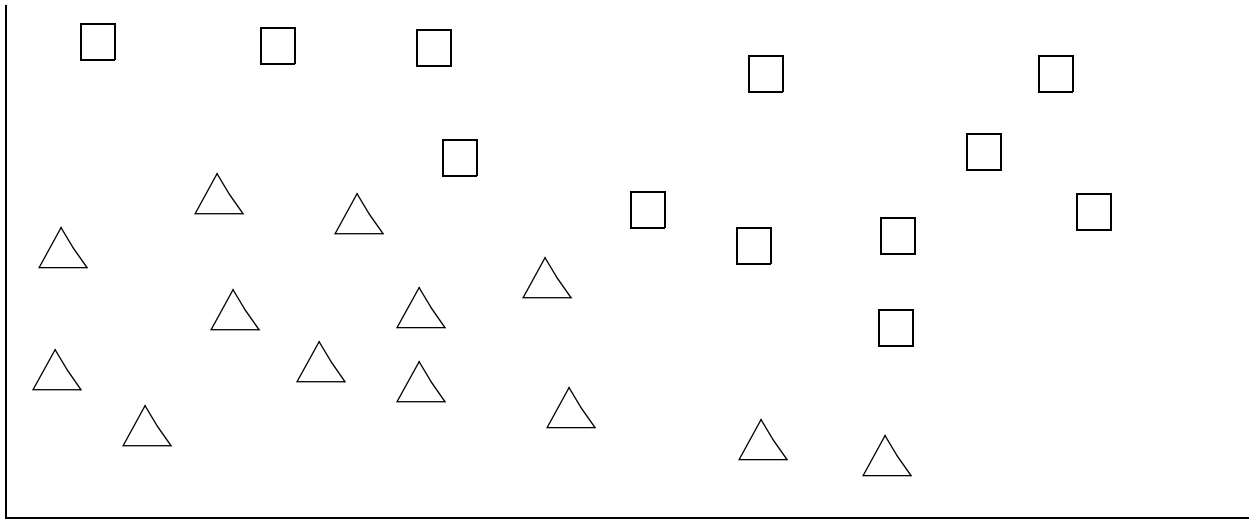
There are two main things that the computations behind the XML SVM classifier do:

- Determine the boundaries between each class. This is done during training.
- Determine the threshold for which the boundaries return the most distinctive results when determining class membership.

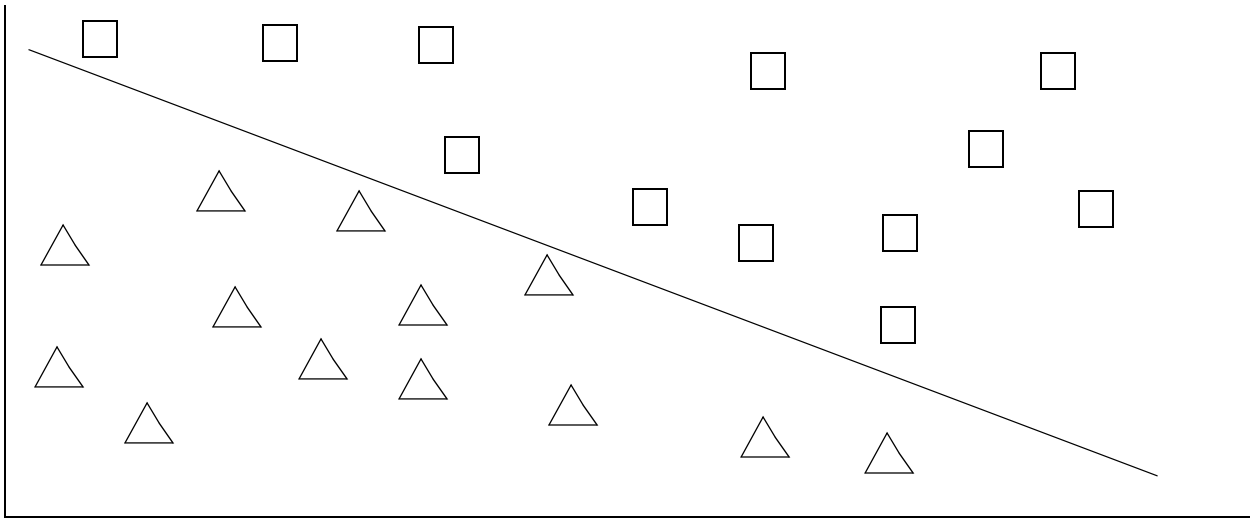
There can be any number of classes. A *term vector* is a representation of all of the terms (as defined by the index options) in a node. Therefore, classes consist of sets of term vectors which have been deemed similar enough to belong to the same class.

Imagine for a moment that each term forms a dimension. It is easy to visualize what a 2-dimensional picture of a class looks like (imagine an x-y graph) or even a 3-dimensional picture (imagine a room with height, width, and length). It becomes difficult, however, to visualize what the picture of these dimensions looks like when there are more than three dimensions. That is where *hyper-planes* become a useful concept.

Before going deeper into the concept of hyper-planes, consider a content set with two classes, one that are squares and one that are triangles. In the following figures, each square or triangle represents a term vector that is a member of either the square or triangle class, respectively.

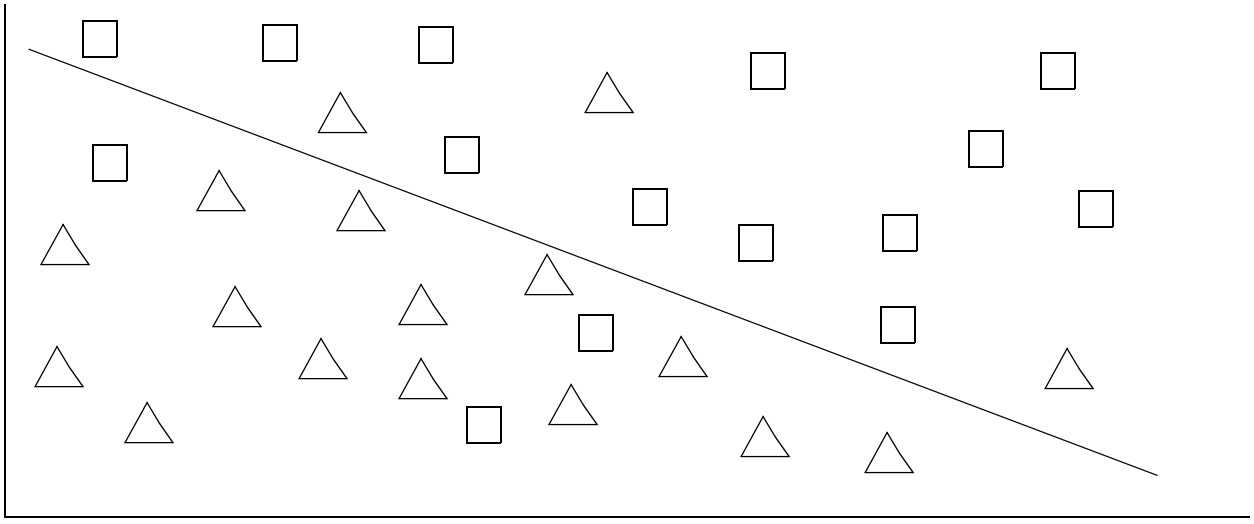


Now try to draw a line to separate the triangles from the squares. In this case, you can draw such a line that nicely divides the two classes as follows:

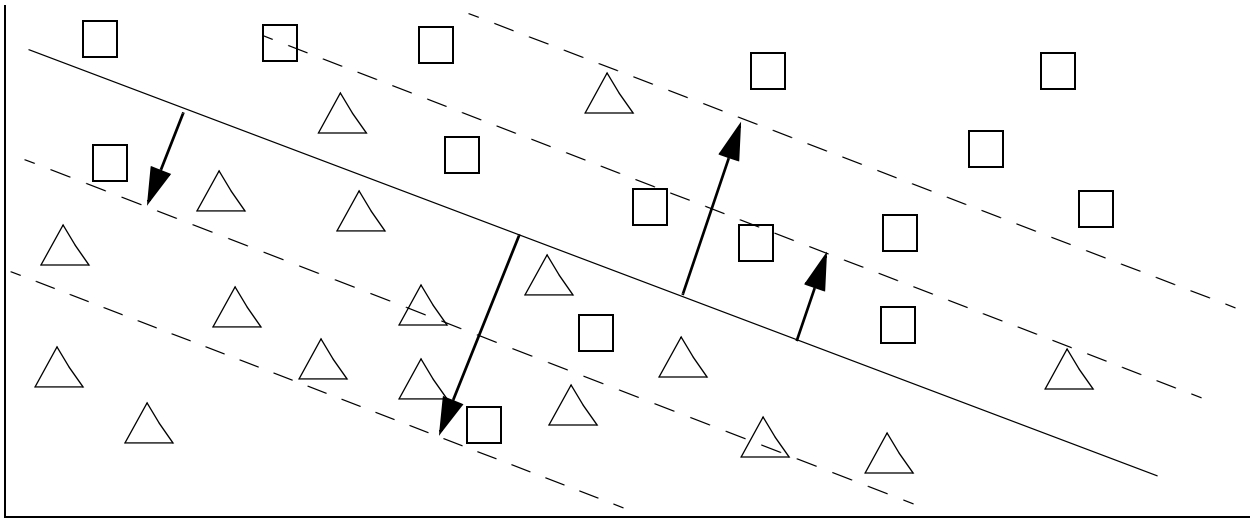


If this were three dimensions, instead of a line between the classes it would be a *plane* between the classes. When the number of dimensions grows beyond three, the extension of the plane is called a *hyper-plane*; it is the generalized representation of a boundary of a class (sometimes called the edge of a class).

The previous examples are somewhat simplified; they are set up such that the hyper-planes can be drawn such that one class is completely on one side and the other is completely on the other. For most real-world content, there are members of each class on the other side of the boundaries as follows:



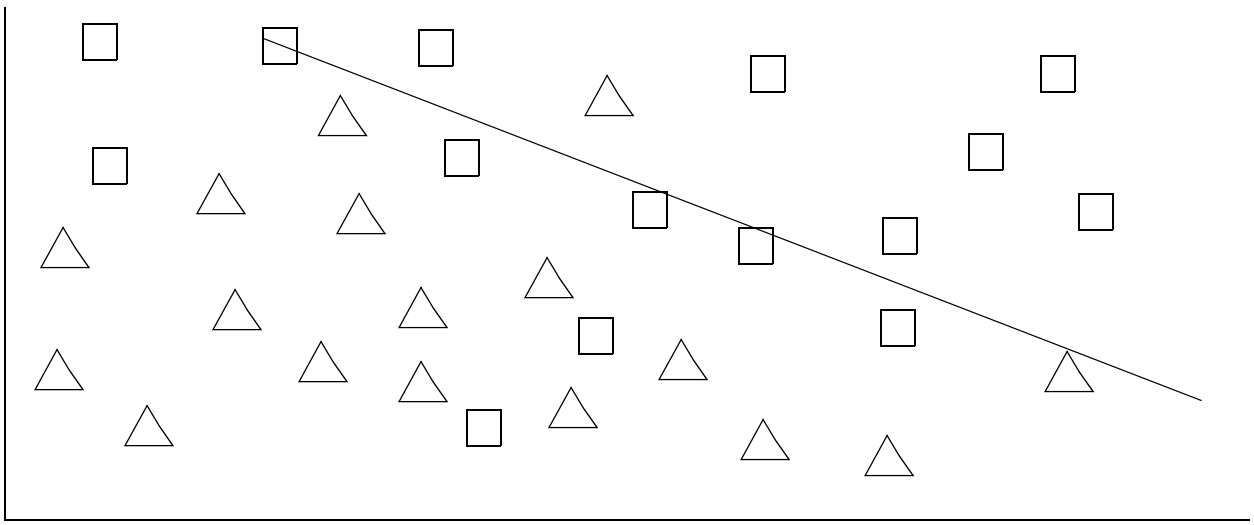
In these cases, you can draw other lines parallel to the boundaries (or in the *n*-dimensional cases, other hyper-planes). These other lines represent the *thresholds* for the classes. The distance between the boundary line and the threshold line represents the threshold value, which is a negative number indicating how far the outlier members of the class are from the class boundary. The following figure represents these thresholds.



The dotted lines represent some possible thresholds. The lines closer to the boundary represent thresholds with higher precision (but not complete precision), while the lines farther from the boundaries represent higher recall. For members of the triangle class that are on the other side of the square class boundaries, those members are not in the class, but if they are within the threshold you choose, then they are considered part of the class.

One of the classifier APIs (`cts:thresholds`) helps you find the right thresholds for your training content set so you can get the right balance between precision and recall when you run unknown content against the classifier to determin class membership.

The following figure shows the triangle class boundary, including the precision and recall calculations based on a threshold (the triangle class is below the threshold line):



Triangle Precision = 16/25 = .64
Triangle Recall = 16/17 = .94

22.1.4 Training Content for the Classifier

To find the best thresholds for your content, you need to *train* the classifier with sample content that represents members of all of the classes. It is very important to find good training samples, as the quality of the training will directly impact the quality of your classification.

The samples for each class should be statistically relevant, and should have samples that include both solid examples of the class (that is, samples that fall well into the positive side of the threshold from the class boundary) and samples that are close to the boundary for the class. The samples close to the boundary are very important, because they help determine the best thresholds for your content. For more details about training sets and setting the threshold, see "Creating a Training Set" on page 275 and "Methodology For Determining Thresholds For Each Class" on page 277.

22.2 Classifier API

The classifier has three XQuery built-in functions. This section gives an overview and explains some of the features of the API, and includes the following parts:

- XQuery Built-In Functions
- Data Can Reside Anywhere or Be Constructed
- API is Extremely Tunable
- Supports Versus Weights Classifiers
- Kernels (Mapping Functions)
- Find Thresholds That Balance Precision and Recall

For details about the syntax and usage of the classifier API, see the *MarkLogic XQuery and XSLT Function Reference*.

22.2.1 XQuery Built-In Functions

The classifier API includes three XQuery functions:

- `cts:classify`
- `cts:thresholds`
- `cts:train`

You use these functions to take training nodes use them to compute classifiers. Creating a classifier specification is an iterative process whereby you create training content, train the classifier (using `cts:train`) with the training content, test your classifier on some other training content (using `cts:classify`), compute the thresholds on the training content (using `cts:threshold`), and repeat this process until you are satisfied with the results. For details about the syntax and usage of the classifier API, see the *MarkLogic XQuery and XSLT Function Reference*.

22.2.2 Data Can Reside Anywhere or Be Constructed

The classifier APIs take nodes and elements, so you can either use XQuery to construct the data for the nodes you are classifying or training, or you can store them in the database (or somewhere else), whichever is more convenient. Because the APIs take nodes as parameters, there is a lot of flexibility in how you store your training and classification data.

> **Note:** There is an exception to this: if you are using the `supports` form of the classifier, then the training data must reside in the database, and you must pass in the training nodes when you perform classification (that is, when you run `cts:classify`) on unknown content.

22.2.3 API is Extremely Tunable

The classifier API has many options, and is therefore extremely tunable. You can choose the different index options and kernel types for `cts:train`, as well as specify limits and thresholds. When you change the kernel type for `cts:train`, it will effect the results you get from classification, as well as effect the performance. Because classification is an iterative process, experimentation with your own content set tends to help get better results from the classifier. You might change some parameters during different iterations and see which gives the better classification for your content.

The following section describes the differences between the supports and weights forms of the classifier. For details on what each option of the classifier does, see the *MarkLogic XQuery and XSLT Function Reference*.

22.2.4 Supports Versus Weights Classifiers

There are two forms of the classifier:

- `supports`: allows the use of some of the more sophisticated kernels. It encodes the classifier by reference to specific documents in the training set, and is therefore more accurate because the whole training document can be used for classification; however, that means that the whole training set must be available during classification, and it must be stored in the database. Furthermore, since constructing a term vector is exactly equivalent to indexing, each time the classifier is invoked it regenerates the index terms for the whole training set. On the other hand, the actual representation of the classifier (the XML returned from `cts:train`) may be a lot more compact. The other advantage of the `supports` form of the classifier is that it can give you error estimates for specific training documents, which may be a sign that those are misclassified or that other parameters are not set to optimal values.
- `weights`: encodes weights for each of the terms. For mathematical reasons, it cannot be used with the Gaussian or Geodesic kernels, although for many problems, those kernels give the best results. Since there will not be a weight for every term in training set (because of term compression), this form of the classifier is intrinsically less precise. If there are a lot of classes and a lot of terms, the classifier representation itself can get quite large. However, there is no need to have the training set on hand during classification, nor

to construct term vectors from it (in essence to regenerate the index terms), so `cts:classify` runs much faster with the `weights` form of the classifier.

Which one you choose depends on your answers to several questions and criteria, such as performance (does the `supports` form take too much time and resources for your data?), accuracy (are you happy with the results you get with the `weights` form with your data?), and other factors you might encounter while experimenting with the different forms. In general, the classifier is extremely tunable, and getting the best results for your data will be an iterative process, both on what you use for training data and what options you use in your classification.

22.2.5 Kernels (Mapping Functions)

You can choose different kernels during the training phase. The kernels are mapping functions, and they are used to determine the distance of a term vector from the edge of the class. For a description of each of the kernel mapping functions, see the documentation for `cts:train` in the *MarkLogic XQuery and XSLT Function Reference*.

22.2.6 Find Thresholds That Balance Precision and Recall

As part of the iterative nature of training to create a classifier specification, one of the overriding goals is to find the best threshold values for your classes and your content set. Ideally, you want to find thresholds that strike a balance between good precision and good recall (for details on precision and recall, see "XML SVM Classifier" on page 268). You use the `cts:thresholds` function to calculate the thresholds based on a training set. For an overview of the iterative process of finding the right thresholds, see "Methodology For Determining Thresholds For Each Class" on page 277.

22.3 Leveraging XML With the Classifier

Because the classifier operates from an XQuery context, and because it is built into MarkLogic Server, it is intrinsically XML-aware. This has many advantages. You can choose to classify based on a particular element or element hierarchy (or even a more complicated XML construct), and then use that classifier against either other like elements or element hierarchies, or even against a totally different set of element or element hierarchies. You can perform XML-based searches to find the best training data. If you have built XML structure into your content, you can leverage that structure with the classifier.

For example, if you have a set of articles that you want to classify, you can classify against only the `<executive-summary>` section of the articles, which can help to exclude references to other content sections, and which might have a more universal style and language than the more detailed sections of the articles. This approach might result in using terms that are highly relevant to the topic of each article for determining class membership.

22.4 Creating a Training Set

This section describes the training content set you use to create a classifier, and includes the following parts:

- Importance of the Training Set
- Defining Labels for the Training Set

22.4.1 Importance of the Training Set

The quality of your classification can only be as good as the training set you use to run the classifier. It is extremely important to choose sample training nodes that not only represent obvious examples of a class, but also samples which represent edge cases that belong in or out of a class.

Because the process of classification is about determining the edges of the classes, having good samples that are close to this edge is important. You cannot always determine what constitutes an edge sample, though, by examining the training sample. It it therefore good practice to get as many different kinds of samples in the training set as possible.

As part of the process of training the classifier, you might need to add more samples, verify that the samples are actually good samples, or even take some samples away (if they turn out to be poor samples) from some classes. Also, you can specify negative samples for a class. It is an iterative process of finding the right training data and setting the various training options until you end up with a classifier that works well for your data.

22.4.2 Defining Labels for the Training Set

The second parameter to `cts:train` is a label specification, which is a sequence of `cts:label` elements, each one having a one `cts:class` child. Each `cts:label` element represents a node in the training set. The `cts:label` elements must be in the order corresponding to the specified training nodes, and they each specify to which class the corresponding training node belongs. For example, the following `cts:label` nodes specifies that the first training node is in the class `comedy`, the second in the class `tragedy`, and the third in the class `history`:

```
<cts:label>
  <cts:class name="comedy"/>
</cts:label>
<cts:label>
  <cts:class name="tragedy"/>
</cts:label>
<cts:label>
  <cts:class name="history"/>
</cts:label>
```

Because the labels must be in the order corresponding to the training nodes, you might find it convenient to generate the labels from the training nodes. For example, the following code extracts the class name for the labels from a property names `playtype` stored in the property corresponding to the training nodes:

```
for $play in xdmp:directory("/plays/", "1")
return
  <cts:labels>
```

```
    <cts:class name={
      xdmp:document-property(xdmp:node-uri($play))//playtype/text()}/>
  </cts:labels>
```

If you have training samples that represent negative samples for a class (that is, they are examples of what does *not* belong in the class), you can label them such by specifying the `val="-1"` attribute on the `cts:class` element as follows:

```
<cts:class name="comedy" val="-1"/>
```

Additionally, you can include multiple classes in a label (because membership in one class is independent of membership in another). For example:

```
<cts:label>
  <cts:class name="comedy" val="-1"/>
  <cts:class name="tragedy"/>
  <cts:class name="history"/>
</cts:label>
```

22.5 Methodology For Determining Thresholds For Each Class

Use the following methodology to determine appropriate per-class thresholds for classification:

1. Partition the training set into two parts. Ideally, the partitions should be statistically equal. One way to achieve this is to randomize which nodes go into one partition and which go into the other.

2. Run `cts:train` on the first half of the training set.

3. Run `cts:classify` on the second half of the training set with the output of `cts:train` from the first half in the previous step. This is to validate that the training data you used produced good classification. Use the default value for the `thresholds` option for this run. The default value is a very large negative number, so this run will measure the distance from the actual class boundary for each node in the training set.

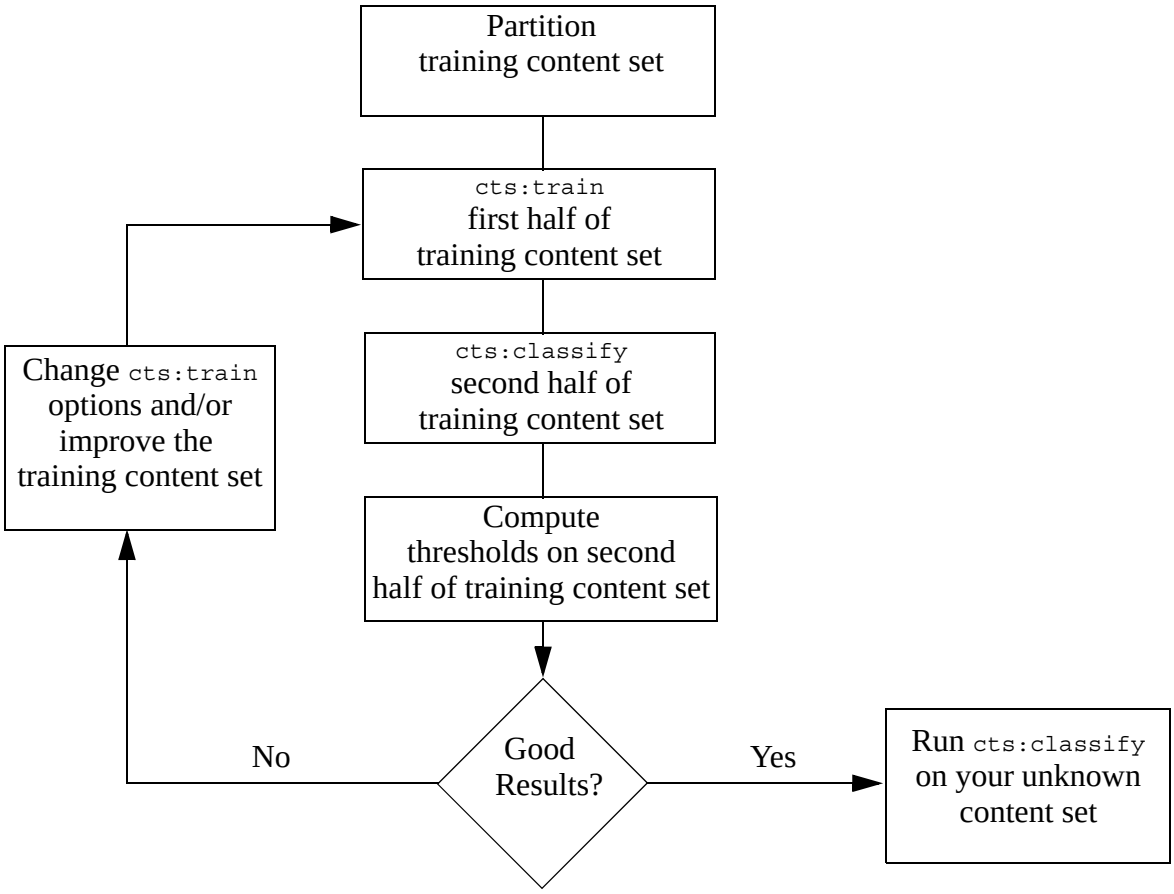
4. Run `cts:thresholds` to compute thresholds for the second half of the training set. This will further validate your training data and the parameters you set when running `cts:train` on your training data.

5. Iterate through the previous steps until you are satisfied with the results from your training content (that is, you until you are satisfied with the classifier you create). You might need to experiment with the various option settings for `cts:train` (for example, different kernels, different index settings, and so on) until you get the classification you desire.

6. After you are satisfied that you are getting good results, run `cts:classify` on the unknown documents, using the computed thresholds (the values from `cts:thresholds`) as the boundaries for deciding on class membership.

Note: Any time you pass thresholds to `cts:train`, the thresholds apply to `cts:classify`. You can pass them either with `cts:train` or `cts:classify`, though, and the effect is the same.

The following diagram illustrates this iterative process:



22.6 Example: Training and Running the Classifier

This section describes the steps needed to train the classifier against a content set of the plays of William Shakespeare. This is meant is a simple example for illustrating how to use the classifier, not necessarily as an example of the best results you can get out of the classifier. The steps are divided into the following parts:

- Shakespeare's Plays: The Training Set
- Comedy, Tragedy, History: The Classes
- Partition the Training Content Set
- Create Labels on the First Half of the Training Content
- Run cts:train on the First Half of the Training Content

- Run cts:classify on the Second Half of the Content Set
- Use cts:thresholds to Compute the Thresholds on the Second Half
- Evaluating Your Results, Make Changes, and Run Another Iteration
- Run the Classifier on Other Content

22.6.1 Shakespeare's Plays: The Training Set

When you are creating a classifier, the first step is to choose some training content. In this example, we will use the plays of William Shakespeare as the training set from which to create a classifier.

The Shakespeare plays are available in XML at the following URL (subject to the copyright restrictions stated in the plays):

http://www.oasis-open.org/cover/bosakShakespeare200.html

This example assumes the plays are loaded into a MarkLogic Server database under the directory `/shakespeare/plays/`. There are 37 plays.

22.6.2 Comedy, Tragedy, History: The Classes

After deciding on the training set, the next step is to choose classes in which you divide the set, as well as choosing labels for those classes. For Shakespeare, the classes are `COMEDY`, `TRAGEDY`, and `HISTORY`. You must decide which plays belong to each class. To determine which Shakespeare plays are comedies, tragedies, and histories, consult your favorite Shakespeare scholars (there is reasonable, but not complete agreement about which plays belong in which classes).

For convenience, we will store the classes in the properties document at each play URI. To create the properties for each document, perform something similar to the following for each play (inserting the appropriate class as the property value):

```
xdmp:document-set-properties("/shakespeare/plays/hamlet.xml",
      <playtype>TRAGEDY</playtype>)
```

For details on properties in MarkLogic Server, see Properties Documents and Directories in the *Application Developer's Guide*.

22.6.3 Partition the Training Content Set

Next, we will divide the training set into two parts, where we know the class of each node in both parts. We will use the first part to train and the second part to validate the classifier built from the first half of the training set. The two parts should be statistically random, and to do that we will simply take the first half in the order that the documents return from the `xdmp:directory` call. You can choose a more sophisticated randomization technique if you like.

22.6.4 Create Labels on the First Half of the Training Content

As we are taking the first half of the play for the training content, we will need labels for each node (in this example, we are using the document node for each play as the training nodes). To create the labels on the first half of the content, run a query statement similar to the following:

```
for $x in xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
return
<cts:label>
  <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
       //playtype/text()}/>
</cts:label>
```

Note: For simplicity, this example uses the first 19 items of the content set as the training nodes. The samples you use should use a statistically random sample of the content for the training set, so you might want to use a slightly more complicated method (that is, one that ensures randomness) for choosing the training set.

22.6.5 Run cts:train on the First Half of the Training Content

Next, you run `cts:train` with your training content and labels. The following code constructs the labels and runs `cts:train` to generate a classifier specification:

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $labels := for $x in $firsthalf
       return
       <cts:label>
         <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                //playtype/text()}/>
       </cts:label>
return
cts:train($firsthalf, $labels,
       <options xmlns="cts:train">
         <classifier-type>supports</classifier-type>
       </options>)
```

You can either save the generated classifier specification in a document in the database or run this code dynamically in the next step.

22.6.6 Run cts:classify on the Second Half of the Content Set

Next, you take the classifier specification created with the first half of the training set and run `cts:classify` on the second half of the content set, as follows:

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $secondhalf := xdmp:directory("/shakespeare/plays/", "1")[20 to 37]
let $classifier :=
  let $labels := for $x in $firsthalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                           //playtype/text()}/>
        </cts:label>
  return
  cts:train($firsthalf, $labels,
      <options xmlns="cts:train">
        <classifier-type>supports</classifier-type>
      </options>)
return
cts:classify($secondhalf, $classifier,
      <options xmlns="cts:classify"/>,
      $firsthalf)
```

22.6.7 Use cts:thresholds to Compute the Thresholds on the Second Half

Next, calculate `cts:label` elements for the second half of the content and use it to compute the thresholds to use with the classifier. The following code runs `cts:train` and `cts:classify` again for clarity, although the output of each could be stored in a document.

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $secondhalf := xdmp:directory("/shakespeare/plays/", "1")[20 to 37]
let $firstlabels := for $x in $firsthalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                                    //playtype/text()}/>
        </cts:label>
let $secondlabels := for $x in $secondhalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                                    //playtype/text()}/>
        </cts:label>
let $classifier :=
    cts:train($firsthalf, $firstlabels,
      <options xmlns="cts:train">
        <classifier-type>supports</classifier-type>
      </options>)
let $classifysecond :=
  cts:classify($secondhalf, $classifier,
        <options xmlns="cts:classify"/>,
        $firsthalf)
return
cts:thresholds($classifysecond, $secondlabels)
```

This produces output similar to the following:

```
<thresholds xmlns="http://marklogic.com/cts">
  <class name="TRAGEDY" threshold="-0.00215207" precision="1"
         recall="0.666667" f="0.8" count="3"/>
  <class name="COMEDY" threshold="0.216902" precision="0.916667"
         recall="1" f="0.956522" count="11"/>
  <class name="HISTORY" threshold="0.567648" precision="1"
         recall="1" f="1" count="4"/>
</thresholds>
```

22.6.8 Evaluating Your Results, Make Changes, and Run Another Iteration

Finally, you can analyze the results from `cts:thresholds`. As an ideal, the thresholds should be zero. In practice, a negative number relatively close to zero makes a good threshold. The threshold for tragedy above is quite good, but the thresholds for the other classes are not quite as good. If you want the thresholds to be better, you can try running everything again with different parameters for the kernel, for the indexing options, and so on. Also, you can change your training data (to try and find better examples of comedy, for example).

22.6.9 Run the Classifier on Other Content

Once you are satisfied with your classifier, you can run it on other content. For example, you can try running it on SPEECH elements in the shakespeare plays, or try it on plays by other playwrights.

23.0 Results Clustering Using cts:cluster

MarkLogic Server includes `cts:cluster`, which uses statistical algorithms to find and label clusters of search results. This chapter describes `cts:cluster` and includes the following sections:

- Understanding cts:cluster
- Options to cts:cluster
- Understanding the cts:cluster Output
- Example that Creates an HTML Report of the Cluster

For details about the signature, the parameter syntax, and more examples, see `cts:cluster` in the *MarkLogic XQuery and XSLT Function Reference*.

23.1 Understanding cts:cluster

The `cts:cluster` function takes a set of nodes, typically from a search result set (although it can be any set of nodes), and provides a report that categorizes the result nodes in *clusters*. A *cluster* is a subset of the results that are statistically similar. For each cluster, it generates a label from the most distinctive terms in that cluster.

The output is an XML node, and you can use the output to generate a user interface that displays the results. For sample output, see “Understanding the cts:cluster Output” on page 286.

The clusterer creates clusters by taking the nodes you pass into `cts:cluster` and running it through the MarkLogic Server indexer. This is very similar to the process when you load a document into the database, but the indexing for results clustering is all done in memory, whereas in the database the indexes are stored to disk. The product of indexing is terms, with each term having a frequency (the number of times it occurs in the document and in the result set). Depending on which index settings you use, you will get a different set of terms. The clusterer takes into account each of the terms, as well as information about the terms (for example, weights and term frequency), to calculate the clusters.

You pass options into `cts:cluster` that determine the behavior of the cluster as well as specify the index settings to use when creating the clusters. For more information about the options, see “Options to cts:cluster” on page 285, as well as the API documentation for `cts:cluster` in the *MarkLogic XQuery and XSLT Function Reference*.

When deciding how to use the clusterer, think about what your requirements are. Many settings you choose in the clusterer are trade-offs between performance and the quality of the results clusters. You might need to experiment to find what works well for your application.

Note the following about the clusterer:

- Every time you cluster, the indexer is run on the supplied nodes to generate the data.
- The more nodes you send to `cts:cluster`, the longer it will take. For real time analysis, more than a few thousand might get too slow for a user to wait. Ideally, between 100 and 1000 nodes is a good balance between performance and good results.
- You can set `<hierarchical-levels>` to a value of greater than 1 to generate clusters of clusters. The parent `attribute` tells you which cluster is its parent. You can then iterate through the result set to create a user interface that shows the tree-like hierarchy.
- The labels might change from run-to-run. Specifying a higher value of `<num-tries>` tends to make the labels more consistent from run-to-run, but will increase the time it takes to produce the clusters.
- The labels come from the most distinctive terms. Some terms (such element terms) are turned into strings. If you want to see the terms used to create the labels, set the `<details>true</details>` option.

23.2 Options to cts:cluster

You can set options to `cts:cluster` in an options node. You can set the following types of options:

- Clustering (cts:cluster) Options
- Indexing (db:) Options

Each of these types of options is in its own namespace.

23.2.1 Clustering (cts:cluster) Options

The clustering options are in the `cts:cluster` namespace. These options determine the output and the behavior of the clusterer. Note the following about the clusterer options:

- When tuning the options, try to balance performance, accuracy, and quality of the results.
- The `<details>` option returns the distinctive terms (these are `cts` terms) used for each cluster. You can use these to try and construct your own labels by generating `cts:query` constructors from each term. You can then use those queries against some of your data to generate some labels, if that makes sense for your application.
- The `<algorithm>` option sets the algorithm MarkLogic Server uses to calculate the clusters: `k-means` or `lsi`. Both are statistical algorithms and have well-known and published papers describing them (to learn more, you can start here: http://en.wikipedia.org/wiki/K-means_clustering and http://en.wikipedia.org/wiki/Latent_semantic_indexing). The default is `k-means`, which tends to be slightly faster, but gives slightly less stable results than `lsi`.

- You can control the number of clusters using `<min-clusters>` and `<max-clusters>` settings. It is possible for `cts:cluster` to return less than the number of clusters in `<min-clusters>` if the most it can calculate based on your data is less than that value.
- The `<num-tries>` option specifies the number of times to run the clusterer against the specified data. The default is 1. Because of the way the algorithms work, running the cluster multiple times will increase the number of terms, and tends to improve the accuracy of the clusters. It does so at the cost of performance, as each time it runs, it has to do more work.

23.2.2 Indexing (db:) Options

The indexing options control which terms are created. MarkLogic Server uses these terms to calculate the clusters, based on term frequency, distinctive terms, and other factors relating to relevancy. Note the following about the `db` options:

- They are set in the options node, and are in the `http://marklogic.com/xdmp/database` namespace.
- The `cts:cluster` database options are the same as the database options for `cts:distinctive-terms`.
- You can construct the options by hand or use the Admin API to construct the options.
- Fields are a good way of indexing only the words you are interested in, and allows you to set weights for certain elements. For details on how fields work, see Fields Database Settings in the *Administrator's Guide*.
- The `<use-db-options>` `cts:cluster` option (in the `cts:cluster` namespace) takes the combination of the database options set in the context database, the specified database options, and any default values for options. This can be a convenient way for setting complicated options.
- Iterate with different options to get the right mix of performance and term choices.

23.3 Understanding the cts:cluster Output

The following shows sample `cts:cluster` output:

```
<clustering xmlns="http://marklogic.com/cts">
  <cluster id="15899142696064772767" label="law, his, hath" count="8" nodes="2
11 22 24 27 30 40 78"/>
  <cluster id="161987570467386344" label="earth, lose, hast" count="1"
nodes="28"/>
  <cluster id="14947979602052601851" label="mark, most, talbot" count="91"
nodes="1 3 4 5 6 7 8 9 10 12 13 14 15 16 17 18 19 20 21 23 25 26 29 31 32 33 34
35 36 37 38 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61
62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 79 80 81 82 83 84 85 86 87 88
89 90 91 92 93 94 95 96 97 98 99 100"/>
  <cluster id="143845517505877166" parent-id="15899142696064772767"
```

```
label="note, captain, antony" count="4" nodes="2 22 30 40"/>
  <cluster id="12625796822979427066" parent-id="15899142696064772767"
label="king, from, so" count="4" nodes="11 24 27 78"/>
  <cluster id="9134217245415181471" parent-id="14947979602052601851"
label="talbot, somerset, who" count="4" nodes="62 72 73 74"/>
  <cluster id="12485013516686263 61" parent-id="14947979602052601851"
label="pompey, wall, cleopatra" count="44" nodes="1 4 5 6 12 13 14 19 33 34 37
39 41 42 45 46 47 48 49 50 51 53 54 55 56 58 60 61 64 65 68 71 75 77 84 87 88
89 92 95 96 97 98 99"/>
  <cluster id="6447791006134911106" parent-id="14947979602052601851"
label="our, voice, these" count="10" nodes="17 29 59 69 79 80 91 93 94 100"/>
  <cluster id="7874080124275500326" parent-id="14947979602052601851"
label="which, peace, blood" count="33" nodes="3 7 8 9 10 15 16 18 20 21 23 25
26 31 32 35 36 38 43 44 52 57 63 66 67 70 76 81 82 83 85 86 90"/>
  <options xmlns="cts:cluster" xmlns:db="http://marklogic.com/xdmp/database">
    <algorithm>k-means</algorithm>
    <db:word-searches>true</db:word-searches>
    <db:fast-phrase-searches>false</db:fast-phrase-searches>
    <db:fast-element-word-searches>true</db:fast-element-word-searches>
    <db:language>en</db:language>
    <max-clusters>10</max-clusters>
    <min-clusters>3</min-clusters>
    <hierarchical-levels>2</hierarchical-levels>
    <initialization>smart</initialization>
    <max-terms>200</max-terms>
    <label-max-terms>3</label-max-terms>
    <label-ignore-words>a as of s the when</label-ignore-words>
    <num-tries>1</num-tries>
    <score>logtfidf</score>
    <use-db-config>false</use-db-config>
    <details>false</details>
    <overlapping>false</overlapping>
  </options>
</clustering>
```

The output is a `cts:clustering` element. The output includes each cluster, as well as the options node used to create it. You can use XQuery or XSLT to iterate through the output, creating a report (for example, in HTML) of the results.

The attributes on the <cluster> element describe the cluster. The following table describes the attributes on the <cluster> element:

cluster Attribute	Description
id	A random number used to identify the cluster.
parent-id	The ID of the parent cluster, when <hierarchical-levels> is set to a value greater than 1.
label	The terms that comprise the label, comma separated. To make your own label, return the <details> and use the terms to generate a label.
count	The number of nodes in the cluster.
nodes	A set of NMTOKEN values, where each value lists the position of the node. The position is ordered by relevance, the first being the most relevant to the cluster and the last being the least relevant. The number refers to the position in the nodes input to cts:cluster. For example, a value of 10 indicates that it is the tenth node in the sequence passed into the first parameter of cts:cluster.

23.4 Example that Creates an HTML Report of the Cluster

The following example creates an HTML report of the cluster. It uses the Shakespeare plays database. To see the results, cut and paste the example and run it against a database that contains the Shakespeare plays (modify the URI of the directory used in the cts:search to the URI of the database directory in which you have loaded the Shakespeare plays).

```
xquery version "1.0-ml" ;

(: cluster the Shakespeare speeches, disregarding the speaker,
   and show the results in an html table :)

declare namespace db="http://marklogic.com/xdmp/database" ;
declare namespace cl="cts:cluster" ;
declare namespace dt="cts:distinctive-terms" ;

(: generally we want to cluster the top N results, where N is
   around 100 to 1,000 (smaller numbers for best performance).
   all speeches = 31,029;
   speeches that contain "love" = 1,864;
   "war" = 359; "joy" = 201;
   "beast" = 94;
   "aunt"=24
:)
let $search-term := xdmp:get-request-field("search-term", "aunt")
```

```
let $max-terms := xdmp:get-request-field("max-terms", "100")
let $use-db-config :=
  xdmp:get-request-field("use-db-config", "false")
let $algorithm := xdmp:get-request-field("algorithm", "k-means")
let $options-node :=
   <options xmlns="cts:cluster" >
      <hierarchical-levels>5</hierarchical-levels>
      <overlapping>false</overlapping>
      <label-max-terms>1</label-max-terms>
      <label-ignore-words>a of the when s as</label-ignore-words>
      <max-clusters>10</max-clusters>
      <algorithm>{ $algorithm }</algorithm>
     <!-- turn all database-level indexing options OFF - only use field
terms -->
      <db:word-searches>false</db:word-searches>
      <db:stemmed-searches>false</db:stemmed-searches>

<db:fast-case-sensitive-searches>false</db:fast-case-sensitive-searche
s>

<db:fast-diacritic-sensitive-searches>false</db:fast-diacritic-sensiti
ve-searches>
      <db:fast-phrase-searches>false</db:fast-phrase-searches>
      <db:phrase-throughs/>
      <db:phrase-arounds/>

<db:fast-element-word-searches>false</db:fast-element-word-searches>

<db:fast-element-phrase-searches>false</db:fast-element-phrase-searche
s>
      <db:element-word-query-throughs/>

<db:fast-element-character-searches>false</db:fast-element-character-s
earches>
      <db:range-element-indexes/>
      <db:range-element-attribute-indexes/>
      <db:one-character-searches>false</db:one-character-searches>
      <db:two-character-searches>false</db:two-character-searches>
      <db:three-character-searches>false</db:three-character-searches>

<db:trailing-wildcard-searches>false</db:trailing-wildcard-searches>

<db:fast-element-trailing-wildcard-searches>false</db:fast-element-tra
iling-wildcard-searches>
      <db:fields>
        <field xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://marklogic.com/xdmp/database">
          <field-name>speeches</field-name>
          <include-root>false</include-root>
          <word-lexicons/>
          <!-- create stem and phrase terms for this field -->
          <!-- if the XML were richer, we would have used
               fast-element-word-searches and
               fast-element-phrase-searches too -->
```

```
            <stemmed-searches>advanced</stemmed-searches>
            <db:fast-phrase-searches>true</db:fast-phrase-searches>
            <included-elements>
             <included-element>
               <namespace-uri/>
               <localname>LINE</localname>
               <weight>1.0</weight>
               <attribute-namespace-uri/>
               <attribute-localname/>
               <attribute-value/>
             </included-element>
             <included-element>
               <namespace-uri/>
               <localname>SPEECH</localname>
               <weight>1.0</weight>
               <attribute-namespace-uri/>
               <attribute-localname/>
               <attribute-value/>
             </included-element>
           </included-elements>
         <excluded-elements>
           <excluded-element>
             <namespace-uri/>
             <localname>SPEAKER</localname>
           </excluded-element>
          </excluded-elements>
         </field>
     </db:fields>
  </options>

(: build the page :)
let $page :=
<html>
<head><title>Example - clustering - speeches</title></head>
<body>
<table border="1" cellpadding="1" cellspacing="1">
<tr>
<th>Label</th>
<th>Count</th>
<th>Speakers</th>
</tr>
{
let $things-to-cluster :=
  cts:search(
     (: specify the directory in which you have loaded the plays :)
     xdmp:directory( "/shakespeare/plays/" )//SPEECH,
        $search-term
        )
(: iterate through the cts:cluster results node :)
for $cluster in
    cts:cluster( $things-to-cluster, $options-node )/cts:cluster
return
    <tr>
     <td>{ fn:data( $cluster/@label ) }</td>
```

```
      <td>{ fn:data( $cluster/@count ) }</td>
      <td>
        <table>{
 for $clustered-node-ref in fn:data( $cluster/@nodes )
 return
      <tr><td>{ fn:string(
        $things-to-cluster[$clustered-node-ref]//SPEAKER )
      }</td></tr>
        }</table>
       </td>
      </tr>}
</table>
</body>
</html>

return ( xdmp:set-response-content-type("text/html"),
 $page, xdmp:elapsed-time() )
```

24.0 Language Support in MarkLogic Server

MarkLogic Server supports loading and querying content in multiple languages. This chapter describes how languages are handled in MarkLogic Server, and includes the following sections:

- Overview of Language Support in MarkLogic Server
- Tokenization and Stemming
- Language Aspects of Loading and Updating Documents
- Querying Documents By Languages
- Supported Languages
- Generic Language Support

24.1 Overview of Language Support in MarkLogic Server

In MarkLogic Server, the language of the content is specified when you load the content and the language of the query is specified when you query the content. At load-time, the content is tokenized, indexed, and stemmed (if enabled) based on the language specified during the load. Also, MarkLogic Server uses any languages specified at the element level in the XML markup for the content (see “xml:lang Attribute” on page 296), making it possible to load documents with multiple languages. Similarly, at query time, search terms are tokenized (and stemmed) based on the language specified in the `cts:query` expression. The result is that a query performed in one language might not yield the same results as the same query performed in another language, as both the indexes that store the information about the content and the queries against the content are language-aware.

Even if your content is entirely in a single language, MarkLogic Server is still multiple-language aware. If your content is all in a single language, and if that language is the default language for that database, and if the content does not have any language (`xml:lang`) attributes, and if your queries all specify (or default to) the language in which the content is loaded, then everything will behave as if there is a single language.

Because MarkLogic Server is multiple-language aware, it is important to understand the fundamental aspects of languages when loading and querying content in MarkLogic Server. The remainder of this chapter, particularly “Language Aspects of Loading and Updating Documents” on page 296 and “Querying Documents By Languages” on page 298, describe these details.

24.2 Tokenization and Stemming

To understand the language implications of querying and loading documents, you must first understand tokenization and stemming, which are both language-specific. This section describes these topics, and has the following parts:

- Language-Specific Tokenization
- Stemmed Searches in Different Languages

24.2.1 Language-Specific Tokenization

When you search for a string (typically a word or a phrase) in MarkLogic Server, or when you load content (which is made up of text strings) into MarkLogic Server, the string is broken down to a set of parts, each of which is called a *token*. Each token is classified as a word, as punctuation, or as whitespace. The process of breaking down strings into tokens is called *tokenization*. Tokenization occurs during document loading as well as during query evaluation, and they are independent of each other.

Tokenization is language-specific; that is, a given string is tokenized differently depending on the language in which it is tokenized. The language is determined based on the language specified at load or query time (or the database default language if no language is specified) and on any `xml:lang` attributes in the content (for details, see “xml:lang Attribute” on page 296).

Note the following about the way strings are tokenized in MarkLogic Server:

- The `cts:tokenize` API will return how text is tokenized in the specified language.
- Using `xdmp:describe` of a `cts:tokenize` expression returns the tokens and the type of tokens produced from the specified string. For example:

```
xdmp:describe(cts:tokenize("this is, obviously, a phrase", "en"), 100)
=> (cts:word("this"), cts:space(" "), cts:word("is"),
   cts:punctuation(","), cts:space(" "), cts:word("obviously"),
   cts:punctuation(","), cts:space(" "), cts:word("a"),
   cts:space(" "), cts:word("phrase"))
```

- Every query has a language associated with it; if the language is not explicitly specified in the `cts:query` expression, then it takes on the default language of the database.

- MarkLogic Server comes configured such that when an element is in an Asian or Middle Eastern language, the Latin characters tokenize as English. This allows searches to find English words inside Asian or Middle Eastern language elements. For example, a search in English can find Latin characters in a Simplified Chinese element as in the following:

```
let $x := <el xml:lang="zh">Chinese-text-here hello</el>
return
$x//el[cts:contains(.,
         cts:word-query("hello", ("stemmed", "lang=en")))]

=> <el xml:lang="zh">Chinese-text-here hello</el>
```

  A stemmed search for the Latin characters in a non-English language, however, will not find the non-English word stems (it will only find the non-English word itself, which stems to itself). Similarly, Asian or Middle Eastern characters will tokenize in a language appropriate to the character set, even when they occur in elements that are not in their language. The result is that searches in English sometimes match content that is labeled in an Asian or Middle Eastern character set, and vice-versa. For example, consider the following (`zh` is the language code for Simplified Chinese):

```
let $x :=
<root>
 <el xml:lang="en">hello</el>
 <el xml:lang="fr">hello</el>
 <el xml:lang="zh">hello</el>
</root>
return
$x//el[cts:contains(.,
         cts:word-query("hello", ("stemmed", "lang=en")))]

=> <el xml:lang="en">hello</el>
   <el xml:lang="zh">hello</el>
```

  This search, even though in English, returns both the element in English and the one in Chinese. It returns the Chinese element because the word “hello” is in Latin characters and therefore tokenizes as English, and it matches the Chinese query (which also tokenizes “hello” in English).

24.2.2 Stemmed Searches in Different Languages

A *stemmed* search for a term matches all the terms that have the same stem as the search term (which includes the exact same terms in the language specified in the query). Words that are derived from the same meaning and part of speech have the same stem (for example, “mouse” and “mice”). Some words can have multiple stems (if the same word can be used as a different part of speech, or if there are two words with the same spelling), and if you use advanced stemming (which can find multiple stems for a word), then stemmed searches find all of the words having the same stem as any of the stems. The purpose of stemming is to increase the recall for a search. For details about how stemming works in MarkLogic Server, including the different stemming types of stemming available, see “Understanding and Using Stemmed Searches” on page 212. This sections describes how the language settings affect stemmed searches.

To get the stem of a search term, you must take the language into consideration. For example, the word “chat” is a different word in French than it is in English (in French, it is a noun meaning “cat”, in English, it is a verb meaning to converse informally). In French, “chatting” is not a word, and therefore it does not stem to “chat”. But in English, “chatting” does stem to “chat”. Therefore, stemming is language-specific, and stemmed searches in one language might find different results than stemmed searches in another.

At query time, you can specify a language (or if you do not specify a language, the default language of the database is used). This language is used when performing a stemmed search. The language specification is in the options to the `cts:query` expression. For example, the following `cts:query` expression specifies a stemmed search in French for the word “chat”, and it only matches tokens that are stemmed in French.

```
cts:word-query("chat", ("stemmed", "lang=fr"))
```

For more details about how languages affect queries, see “Querying Documents By Languages” on page 298.

At load time, the specified language is used to determine in which language to stem the words in the document. For more details about the language aspects of loading documents, see “Language Aspects of Loading and Updating Documents” on page 296.

For details about the syntax of the various `cts:query` constructors, see the *MarkLogic XQuery and XSLT Function Reference*.

24.3 Language Aspects of Loading and Updating Documents

This section describes the impact of languages on loading and updating documents, and includes the following sections:

- Tokenization and Stemming
- xml:lang Attribute
- Language-Related Notes About Loading and Updating Documents

24.3.1 Tokenization and Stemming

Tokenization and stemming occur when loading documents, just as they do when querying documents (for details, see “Language-Specific Tokenization” on page 293 and “Stemmed Searches in Different Languages” on page 295). When loading documents, the `stemmed search` indexes are created based on the language. The tokenization and stemming at load time is completely independent from the tokenization and stemming at query time.

24.3.2 xml:lang Attribute

You can specify languages in XML documents at the element level by using the `xml:lang` attribute. MarkLogic Server uses the `xml:lang` attribute to determine the language with which to tokenize and stem the contents of that element. Note the following about the `xml:lang` attribute:

- The `xml:lang` attribute (see http://www.w3.org/TR/2006/REC-xml-20060816/#sec-lang-tag) has some special properties such as not needing to declare the namespace bound to the `xml` prefix, and that it is inherited by all children of the element (unless they explicitly have a different `xml:lang` value).
- You can explicitly add an `xml:lang` attribute to the root node of a document during loading by specifying the `default-language` option to `xdmp:document-load`; without the `default-language` option, the root node will remain as-is.
- If no `xml:lang` attribute is present, then the document is processed in the default language of the database into which it is loaded.
- For the purpose of indexing terms, the language specified by the `xml:lang` attribute only applies to stemmed search terms; the `word searches` (unstemmed) database configuration setting indexes terms irrespective of language. Tokenization of terms honors the `xml:lang` value for both `stemmed searches` and `word searches` index settings in the database configuration.
- All of the text node children and text node descendants of an element with an `xml:lang` attribute are treated as the language specified in the `xml:lang` attribute, unless a child element has an `xml:lang` attribute with a different value. If so, any text node children and text node descendants are treated as the new language, and so on until no other `xml:lang` attributes are encountered.

- The value of the `xml:lang` attribute must conform to the following lexical standard: http://www.ietf.org/rfc/rfc3066.txt. The following are some typical `xml:lang` attributes (specifying French, Simplified Chinese, and English, respectively):

```
xml:lang="fr"
xml:lang="zh"
xml:lang="en"
```

- If an element has an `xml:lang` attribute with a value of the empty string (`xml:lang=""`), then any `xml:lang` value in effect (from some ancestor `xml:lang` value) is overridden for that element; its value takes on the database language default. Additionally, if a `default-language` option is specified during loading, any empty string `xml:lang` values are replaced with the language specified in the `default-language` option. For example, consider the following XML:

```
<rhone xml:lang="fr">
   <wine>vin rouge</wine>
   <wine xml:lang="">red wine</wine>
</rhone>
```

  In this sample, the phrase “vin rouge” is treated as French, and the phrase “red wine” is treated in the default language for the database (English by default).

  If this sample was loaded with a default-language option specifying Italian (specifying `<default-language>it</default-language>` for the `xdmp:document-load` option, for example), then the resulting document would be as follows:

```
<rhone xml:lang="fr">
   <wine>vin rouge</wine>
   <wine xml:lang="it">red wine</wine>
</rhone>
```

24.3.3 Language-Related Notes About Loading and Updating Documents

When you load content into MarkLogic Server, it determines how to index the content based on several factors, including the language specified during the load operation, the default language of the database, and any languages encoded into the content with `xml:lang` attributes. Note the following about languages with respect to loading content, updating content, and changing language settings on a database:

- Changing the default language starts a reindex operation if `reindex enable` is set to `true`.
- Documents with no `xml:lang` attribute are indexed upon load or update in the database default language.
- Any content within an element having an `xml:lang` attribute is indexed in that language. Additionally, the `xml:lang` value is inherited by all of the descendants of that element, until another `xml:lang` value is encountered.

- MarkLogic Server comes configured such that when an element is in an Asian or Middle Eastern language, the Latin characters tokenize as English. Therefore, a document with Latin characters in a non-English language will create stemmed index terms in English for those Latin characters. Similarly, Asian or Middle Eastern characters will tokenize in their respective languages, even in elements that are not in their language.

24.4 Querying Documents By Languages

Full-text search queries (queries that use `cts:search` or `cts:contains`) are language-aware; that is, they search for text, tokenize the search terms, and stem (if enabled) in a particular language. This section describes how queries are language-aware and describes their behavior. It includes the following topics:

- Tokenization, Stemming, and the xml:lang Attribute
- Language-Aware Searches
- Unstemmed Searches
- Unknown Languages

24.4.1 Tokenization, Stemming, and the xml:lang Attribute

Tokenization and stemming are both language-specific; that is, a string can be tokenized and stemmed differently in different languages. For searches, the language is specified by the cts:query constructors (or by the default language of the database if a language is not specified). For more details, see “Tokenization and Stemming” on page 293. For nodes constructed in XQuery, any `xml:lang` attributes are treated the same way as if the document were loaded into a database. For details, see “xml:lang Attribute” on page 296.

24.4.2 Language-Aware Searches

All searches in MarkLogic Server are language-aware. You can construct searches using `cts:search or cts:contains`, each of which takes a `cts:query` expression. Each leaf-level `cts:query` constructor in the `cts:query` expression specifies a language (or defaults to a language). For details on the `cts:query` constructors, see “Composing cts:query Expressions” on page 121.

All searches use the language setting in the `cts:query` constructor to determine how to tokenize the search terms. Stemmed searches also use the language setting to derive stems. Unstemmed searches use the specified language for tokenization but use the unstemmed (`word searches`) indexes, which are language-independent.

24.4.3 Unstemmed Searches

An *unstemmed* search matches terms that are exactly like the search term; it does not take into consideration the stem of the word. Unstemmed searches match terms in a language independent way, but tokenize the search according to the specified language. Therefore, when you specify a language in an unstemmed query, the language applies only to tokenization; the unstemmed query will match any text in any language that matches the query.

Note the following characteristics of unstemmed searches:

- Unstemmed searches require `word search` indexes, otherwise they throw an exception (this is a change of behavior from MarkLogic Server 3.1, see the *Release Notes* for details). You can perform unstemmed searches without `word search` indexes using `cts:contains`, however. To perform unstemmed searches without the `word search` indexes enabled, use a `let` to bind the results of a stemmed search to a variable, and then filter the results using `cts:contains` with an unstemmed query.

  The following example demonstrates this. It binds the unstemmed search to a varaiable, then iterates over the results of the search in a FLWOR loop, filtering out all but the unstemmed results in the `where` clause (using `cts:contains` with a `cts:query` that specifies the `unstemmed` option).

  ```
  let $search := cts:search(doc(), cts:word-query("my words",
                                      ("stemmed", "lang=en")))
  for $x in $search
  where cts:contains($x, cts:word-query("my words", "unstemmed"))
  return $x
  ```

  Note: While it is likely that everything returned by this search will have an English match to the `cts:query`, it does not necessarily *guarantee* that everything returned is in English. Because this search returns documents, it is possible for a document to contain words in another language that do not match the language-specific query, but do match the unstemmed query (if the document contains text in multiple languages, and if it has “my words” in some other language than the one specified in the stemmed `cts:query`).

- The `word search` indexes have no language information.
- Unstemmed searches use the `lang=<language>` option to determine the language for tokenization.

- Unstemmed searches search all content, regardless of language (and regardless of `lang=<language>` option). The language only affects how the search terms are tokenized. For example, the following unstemmed search returns true:

```
(: returns true :)
let $x := <el xml:lang="fr">chat</el>
return
cts:contains($x, cts:word-query("chat", ("unstemmed", "lang=en")))
```

  whereas the following stemmed search returns false:

```
(: returns false :)
let $x := <el xml:lang="fr">chat</el>
return
cts:contains($x, cts:word-query("chat", ("stemmed", "lang=en")))
```

24.4.4 Unknown Languages

If the language specified in a search is not one of the languages in which language-specific stemming and tokenization are supported, or if it is a language for which you do not have a license key, then it is treated as a generic language. Typically, generic languages with Latin script are tokenized the same way as English, with token breaks at whitespace and punctuation, and with each word stemming to itself, but this is not always the case (especially for languages supported by MarkLogic Server—see “Supported Languages” on page 301—but for which you are not licensed). For details, see “Generic Language Support” on page 302.

24.5 Supported Languages

This section lists languages with advanced stemming and tokenization support in MarkLogic Server. All of the languages except English require a license key with support for the language. If your license key does not include support for a given language, the language is treated as a generic language (see “Generic Language Support” on page 302). The following are the supported languages:

- English
- French
- Italian
- German
- Russian
- Spanish
- Arabic
- Chinese (Simplified and Traditional)
- Korean
- Persian (Farsi)
- Dutch
- Japanese
- Portuguese

For a list of base collations and character sets used with each language, see “Collations and Character Sets By Language” on page 311.

24.6 Generic Language Support

You can load and query documents in any language into MarkLogic Server, as long as you can convert the character encoding to UTF-8. If the language is not one of the languages with advanced support, or if the language is one for which you are not licensed, then the tokenization is performed in a generic way (on whitespace and punctuation characters for non-Asian characters, and on each character for Asian characters), and each term stems to itself.

For example, if you load the following document:

```
<doc xml:lang="nn">
  <a>Some text in any language here.</a>
</doc>
```

then that document is loaded as the language `nn`, and a stemmed search in any other language would not match. Therefore, the following does not match the document:

```
(: does not match because it was stemmed as "nn" :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=en"))
```

and the following search does match the document:

```
(: does match because the query specifies "nn" as the language :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=nn"))
```

Generic language support allows you to query documents in any language, regardless of which languages you are licensed for or which languages have advanced support. Because the generic language support only stems words to themselves, queries in these languages will not include variations of words based on their meanings in the results. If you desire further support than the generic language support for some particular language, contact MarkLogic Technical Support.

25.0 Encodings and Collations

In addition to the language support described in "Language Support in MarkLogic Server" on page 292, MarkLogic Server also supports many character encodings and has the ability to sort the content in a variety of collations. This chapter describes the MarkLogic Server support of encodings and collations, and includes the following sections:

- Character Encoding
- Collations
- Collations and Character Sets By Language

25.1 Character Encoding

MarkLogic Server stores all content in the UTF-8 encoding. If you try to load non-UTF-8 content into MarkLogic Server without translating it to UTF-8, the server throws an exception. If you have non-UTF-8 content, then you can specify the encoding for the content during ingestion, and MarkLogic Server will translate it to UTF-8. To specify an encoding, use the `encoding` option to `xdmp:document-load`, `xdmp:document-get`, and `xdmp:zip-get`. This option tells MarkLogic Server that your content is in that encoding, and MarkLogic Server will attempt to translate that encoding to UTF-8. If the content cannot be translated, MarkLogic Server throws an exception indicating that there is non-UTF-8 content.

Note the following about character encodings and the `encoding` option to the `xdmp:document-load`, `xdmp:document-get`, `xdmp:unquote`, and `xdmp:zip-get` functions:

- The functions always convert the content into UTF-8.
- If no option is specified and there is no HTTP header, the encoding defaults to UTF-8.
- If no option is specified and there is an HTTP header specifying the encoding, then that encoding is used.
- Otherwise, the functions default to UTF-8 for the encoding.
- If the encoding is UTF-8 and any non-UTF-8 characters are found, an exception is thrown indicating the content contains non-UTF-8 characters.
- The `encoding` option is available to `xdmp:document-load`, `xdmp:document-get`, and `xdmp:zip-get`.
- MarkLogic Server assumes the character set you specify is actually the character set of the content. If you specify an encoding that is different from the actual encoding of the characters, the result can be unpredictable: in some cases you might get an exception, but in some cases, you might end up with the wrong characters. Therefore, specifying the wrong encoding can produce incorrect characters.

For details on the syntax of the encoding option, see the *MarkLogic XQuery and XSLT Function Reference*.

25.2 Collations

This section describes collations in MarkLogic Server. Collations specify the order in which strings are sorted and how they are compared. The section includes the following parts:

- Overview of Collations
- Two Common Collation URIs
- Collation URI Syntax
- Backward Compatibility with 3.1 Range Indexes and Lexicons
- UCA Root Collation
- How Collation Defaults are Determined
- Specifying Collations

25.2.1 Overview of Collations

A *collation* specifies the order for sorting strings. The collation settings determine the order for operations where the order is specified (either implicitly or explicitly) and for operations that use Range Indexes. Examples of operations that specify the order are XQuery statements with an `order by` clause, XQuery standard functions that compare order (for example, `fn:compare`, `fn:substring-after`, `fn:substring-before`, and so on), and lexicon functions (for example, `cts:words`, `cts:element-word-match`, `cts:element-values`, and so on). Additionally, collations determine uniqueness in string comparisons, so two strings that are equal according to one collation might be not be equal according to another.

The codepoint-order collation sorts according to the Unicode codepoint order, which does not take into account any language-specific information. There are other collations that are often used to specify language-specific sorting differences. For example, a code point sort puts all uppercase letters before lower-case letters, so the word `Zounds` sorts before the word `abracadabra`. If you use a collation that sorts upper and lower-case letters together (for example, the order `A a B b C c`, and so on), then `abracadabra` sorts before `Zounds`.

Collations are specified with a URI (for example, `http://marklogic.com/collation/`). The collation URIs are specific to MarkLogic Server, but they specify collations according to the Unicode collation standards. There are many variations to collations, and many sort orders that are based on preferences and traditions in various languages. The following section describes the syntax of collation URIs. Although there are a huge number of collation URIs possible, most applications will use only a small number of collations. For more information about collations, see http://icu.sourceforge.net/userguide/Collate_Concepts.html.

25.2.2 Two Common Collation URIs

The following are two very common collation URIs used in MarkLogic Server:

- `http://marklogic.com/collation/`
- `http://marklogic.com/collation/codepoint`

The first one is the UCA Root Collation (see "UCA Root Collation" on page 309), and is the system default. The second is the codepoint order collation, and was the default in pre-3.2 releases of MarkLogic Server.

25.2.3 Collation URI Syntax

Collations in MarkLogic Server are specified by a URI. All collations begin with the string `http://marklogic.com/collation/`. The syntax for collations is as follows:

```
http://marklogic.com/collation/<locale>[/<attribute>]*
```

This section describes the following parts of the syntax:

- Locale Portion of the Collation URI
- Attribute Portion of the Collation URI

25.2.3.1 Locale Portion of the Collation URI

The `<locale>` portion of the collation URI must be a valid locale, and is defined as follows:

```
<locale> ::= <language>[-<script>][_<region>][@(collation=<value>;)+]
```

For a list of valid language codes, see the following:

```
http://www.loc.gov/standards/iso639-2/php/code_list.php
```

For a list of valid script codes, see the following:

```
http://www.unicode.org/iso15924/iso15924-codes.html
```

For a list of valid region codes, see the following:

```
http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html
```

Some languages (for example, German and Chinese) have multiple collations you can specify in the locale. To specify one of these language-specific collation variants, use the `@collation=<value>` portion of the syntax.

If you do not specify a locale in the collation URI, the UCA Root Collation is used by default (for details, see "UCA Root Collation" on page 309).

Note: While you can specify many valid language, script, or region codes, MarkLogic Server only fully supports those that are relevant to and most commonly used with the supported languages. For a list of supported languages along with their common collations, see "Collations and Character Sets By Language" on page 311.

The following table lists some typical locales, along with a brief description:

Locale	Description	Collation URI
`en`	English language	`http://marklogic.com/collation/en`
`en_US`	English language with United States region	`http://marklogic.com/collation/en_US`
`zh`	Chinese language	`http://marklogic.com/collation/zh`
`de@collation=phonebook`	German language with the `phonebook` collation	`http://marklogic.com/collation/de@collation=phonebook`

25.2.3.2 Attribute Portion of the Collation URI

There can be zero or more `<attribute>` portions of the collation URI. Attributes further specify characteristics such as which collation to use, whether to be case sensitive or case insensitive, and so on. You only need to specify attributes if they differ from the defaults for the specified locale. Attributes have the following syntax:

```
<attribute> ::= <strength> | <case-level> | <case-first> |
                <alternate> | <numeric-collation> |
                <variable-top> | <normalization-checking> |
                <french> | <hiragana>
```

The following table describes the various attributes. For simplicity, terms like case-sensitive, diacritic-sensitive, and others are used. In actuality, the definitions of these terms for use in collations are somewhat more complicated. For the exact technical meaning of each attribute, see http://icu.sourceforge.net/userguide/Collate_Concepts.html.

Attribute	Legal Values	Descriptions
`<strength>` The level of comparison to use.	`S1`	Specifies case and diacritic insensitive.
	`S2`	Specifies diacritic sensitive and case insensitive.
	`S3`	Specifies case and diacritic sensitive.
	`S4`	Specifies punctuation sensitive.
	`SI`	Specifies identity (codepoint differentiated).
`<case-level>` Enable or disable the case sensitive level, skipping the diacritic sensitive level. So diacritic insensitive, case sensitive is `/S1/EO` Default: `EX`	`EO`	Specifies enable case-level.
	`EX`	Specifies disable case-level.
`<case-first>` Specifies whether uppercase sorts before or after lowercase. Default: `CX`	`CU`	Specifies that uppercase sorts first.
	`CL`	Specifies that lowercase sorts first.
	`CX`	Off.
`<alternate>` Specifies how to handle variable characters. (As completely ignorable or as normal characters.) Default: `AN`	`AN`	Specifies that all characters are non-ignorable; that is, include all spaces and punctuation characters when sorting characters.
	`AS`	Specifies that variable characters are shifted (ignored) according to the `variable-top` setting.

Attribute	Legal Values	Descriptions
`<numeric-collation>` Order numbers as numbers rather than collation order (for example, 20 < 100). Default: `MX`	`MO`	Specifies numeric ordering.
	`MX`	Specifies non-numeric ordering (order according to the collation).
`<variable-top>` Used with `alternate` to specify which variable characters are ignorable. Any character that is primary-less-than (for details on this concept, see the Unicode link in "UCA Root Collation" on page 309) the cutoff character will be treated as ignorable. Only meaningful in combination with `AS`. Default: `T0000`	`T0000`	Specifies that all variable characters (typically whitespace and punctuation) are ignored for sorting variable characters.
	`T0020`	Specifies that whitespace is ignorable when sorting characters. For example, `/T0020/AS` means that period (a variable character) would be treated as a regular character but space would be ignorable. Therefore: `A B = AB and AB < A.B.`
	`T00BB`	Specifies that most punctuation and space characters are ignorable when sorting characters. Specifically, characters whose sort key is less than or equal to `00BB` are ignorable.
`<normalization-checking>` Specifies whether to perform Unicode normalization on the input string. Default: `NX`	`NO`	Specifies normalize Unicode.
	`NX`	Specifies do not normalize Unicode.

Attribute	Legal Values	Descriptions
`<french>` Specifies whether to apply the French accent ordering rule (that is, to reverse the ordering at the `S3` level). Default: `FX`	`FO`	Specifies French accent ordering.
	`FX`	Specifies normal ordering (according to the collation).
`<hiragana>` Specifies whether to add an additional level to distinguish Hiragana from Katakana. Default: `HX`	`HO`	Hiragana mode on.
	`HX`	Hiragana mode off.

25.2.4 Backward Compatibility with 3.1 Range Indexes and Lexicons

Range Indexes and lexicons that were created in MarkLogic Server 3.1 use the Unicode codepoint collation order. If you want them to use a different collation in any of these indexes and/or lexicons, you must change the collation and re-create the index, and then reindex the database (if `reindex enable` is set to true, it will automatically begin reindexing).

25.2.5 UCA Root Collation

The Unicode collation algorithm (UCA) root collation in MarkLogic Server is used when no default exists. It uses the Unicode codepoint collation with S3 (case and diacritic sensitive) strength, and it has the following URI:

```
http://marklogic.com/collation/
```

The UCA root collation adds more useful case and diacritic sensitivity to the Unicode codepoint order, so it will make more sensible sort orders when you take case sensitivity and diacritic sensitivity into consideration. For more details about the UCA, see http://www.unicode.org/unicode/reports/tr10/.

25.2.6 How Collation Defaults are Determined

The collation used for requests in MarkLogic Server is based on the settings of various parameters in the Admin Interface and on what is specified in your XQuery code. Each App Server has a default collation specified, and that is used in the absence of anything else that overrides it. Note the following about collations and their defaults.

- Collations are specified at the App Server level, on Range Indexes, and on lexicons.
- App Servers, Range Indexes, and lexicons upgraded from 3.1 remain in codepoint order (`http://marklogic.com/collation/codepoint`).
- New App Servers default to the UCA Root Collation (`http://marklogic.com/collation/`).
- New Range Indexes and lexicons default to UCA Root Collation (`http://marklogic.com/collation/`).
- You can specify a default collation in an XQuery prolog, which overrides the App Server default. For example, the following query will use the French collation:

```
xquery version "1.0-ml";
declare default collation "http://marklogic.com/collation/fr";

for $x in ("côte", "cote", "coté", "côté", "cpte" )
order by $x
return $x
```

- The codepoint collation URI is as follows:

```
http://marklogic.com/collation/codepoint
```

  The following is an alias to the codepoint collation URI (used with the `1.0` strict XQuery dialect):

```
http://www.w3.org/2005/xpath-functions/collation/codepoint
```

- Collation URIs displayed in the Admin Interface are stored and displayed as the canonical representation of the URI entered. The canonical representation is equivalent to the URI entered, but changes the order and simplifies portions of the collation URI string to a predetermined order. The `xdmp:collation-canonical-uri` built-in XQuery function returns the canonical URI of any valid collation URI.
- The empty string URI becomes codepoint collation. Therefore, the following returns as shown:

```
xdmp:collation-canonical-uri("")
=> http://marklogic.com/collation/codepoint
```

- The collation used in an XQuery module is determined on a per-module basis. Therefore, a module might call another module that uses a different collation, as each module determines its collation independent of the module that called it (based on the App Server defaults, collation prolog declaration, and so on).
- When a module is invoked or spawned from another module, or when a request is submitted via an `xdmp:eval` call from another module, the new request inherits the collation context of the calling module. That context can be overridden in the query (for

example, with a `declare default collation` expression in the prolog), but it will default to the context from the calling module.

- If no other collations are in effect (for example, for scheduled tasks), the codepoint collation is used.

25.2.7 Specifying Collations

You can specify collations in many places. Some common places to specify collations are:

- In the `order by` clause of a FLWOR expression.
- In an App Server configuration in the Admin Interface.
- In a lexicon or Range Index specification in the Admin Interface.
- In many W3C standard XQuery functions (for example, `fn:compare`, `fn:contains`, `fn:starts-with`, `fn:ends-with`, `fn:substring-after`, `fn:substring-before`, `fn:deep-equals`, `fn:distinct-values`, `fn:index-of`, `fn:max`, `fn:min`).
- In the lexicon APIs (`cts:words`, `cts:word-match`, `cts:element-words`, `cts:element-values`, and so on).
- In the range query constructors (`cts:element-range-query`, `cts:element-attribute-range-query`).

25.3 Collations and Character Sets By Language

The following table lists the languages in which MarkLogic Server supports language-specific tokenization and stemming. It also lists some common collations and character sets for each language.

Language	Base Collations		Character Sets
English	`http://marklogic.com/collation/en`	case/diacritic sensitive	`ISO-8859-1` `cp1252`
	`http://marklogic.com/collation/en/S1`	case/diacritic insensitive	
	`http://marklogic.com/collation/en/S2`	diacritic sensitive	
	`http://marklogic.com/collation/en/S1/EO`	case sensitive	

Language	Base Collations		Character Sets
French	http://marklogic.com/collation/fr	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/fr/S1	case/diacritic insensitive	
	http://marklogic.com/collation/fr/S2	diacritic sensitive	
	http://marklogic.com/collation/fr/S1/EO	case sensitive	
Italian	http://marklogic.com/collation/it	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/it/S1	case/diacritic insensitive	
	http://marklogic.com/collation/it/S2	diacritic sensitive	
	http://marklogic.com/collation/it/S1/EO	case sensitive	
German	http://marklogic.com/collation/de	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/de/S1	case/diacritic insensitive	
	http://marklogic.com/collation/de/S2	diacritic sensitive	
	http://marklogic.com/collation/de/S1/EO	case sensitive	
	http://marklogic.com/collation/de@collation=phonebook	alternate German collation	
Spanish	http://marklogic.com/collation/es	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/es/S1	case/diacritic insensitive	
	http://marklogic.com/collation/es/S2	diacritic sensitive	
	http://marklogic.com/collation/es/S1/EO	case sensitive	
	http://marklogic.com/collation/es@collation=traditional	Treats ll and ch as distinct characters	

Language	Base Collations		Character Sets
Russian	http://marklogic.com/collation/ru	case/diacritic sensitive	cp1251 KOI8-R ISO-8859-5
	http://marklogic.com/collation/ru/S1	case/diacritic insensitive	
	http://marklogic.com/collation/ru/S2	diacritic sensitive	
	http://marklogic.com/collation/ru/S1/EO	case sensitive	
Arabic	http://marklogic.com/collation/ar	form-variant sensitive	cp1256 ISO-8859-6
	http://marklogic.com/collation/ar/S1	form-variant insensitive	
Chinese (Simplified and Traditional)	http://marklogic.com/collation/zh (simplified)	case/diacritic sensitive	Simplified: GB18030 GB2312 EUC-CN hz-gb-2312 cp936 Traditional: Big5 Big5-HKSCS cp950 GB18030
	http://marklogic.com/collation/zh-Hant (traditional)	case/diacritic sensitive	
	http://marklogic.com/collation/zh-Hant@collation=stroke (traditional with simplified order)	locale-specific variant	
	http://marklogic.com/collation/zh@collation=pinyan (simplified with traditional order)	locale-specific variant	
Korean	http://marklogic.com/collation/ko	case/diacritic sensitive	ISO 2022-KR EUC-KR KS X 1001 cp949 GB12052 KSC 5636
	http://marklogic.com/collation/ko/S1	case/diacritic insensitive	

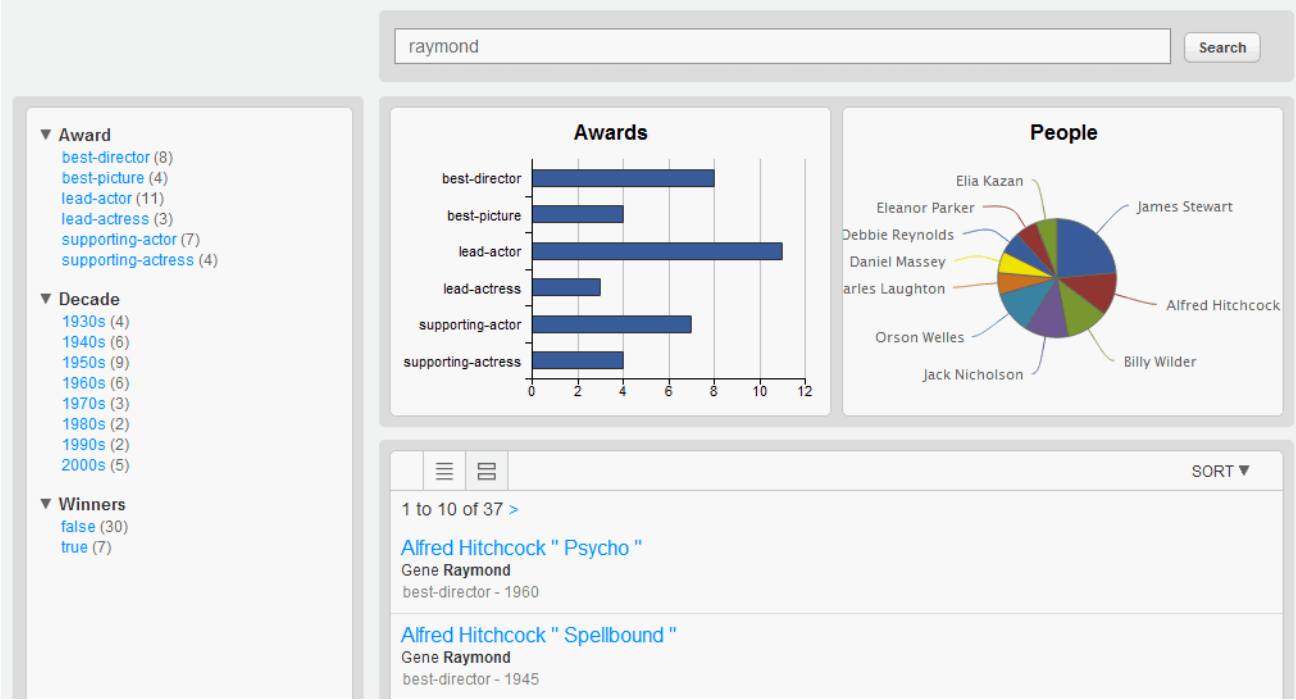
Language	Base Collations		Character Sets
Persian (Farsi)	http://marklogic.com/collation/fa	case/diacritic sensitive	cp1256 ISO-8859-6
	http://marklogic.com/collation/fa/S1	case/diacritic insensitive	
	http://marklogic.com/collation/fa/S2	diacritic sensitive	
	http://marklogic.com/collation/fa/NX	disable normalization	
Dutch	http://marklogic.com/collation/nl	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/nl/S1	case/diacritic insensitive	
	http://marklogic.com/collation/nl/S2	diacritic sensitive	
	http://marklogic.com/collation/nl/S1/EO	case sensitive	
Japanese	http://marklogic.com/collation/ja http://marklogic.com/collation/ja/S1	case/diacritic insensitive	Shift JIS: cp932 ibm-942 ibm-943 EUC-JP: EUC-JISX0213 ibm-954 ISO-2022-JP: ISO-2022-JP-1 ISO-2022-JP-2 ISO-2022-JP-3 ISO-2022-JP-2004
	http://marklogic.com/collation/ja/S2	diacritic sensitive	
	http://marklogic.com/collation/ja/S1/EO	case sensitive	
	http://marklogic.com/collation/ja/S4/HX	Hiragana mode off	
Portuguese	http://marklogic.com/collation/pt	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/pt/S1	case/diacritic insensitive	
	http://marklogic.com/collation/pt/S2	diacritic sensitive	
	http://marklogic.com/collation/pt/S1/E0	case sensitive	

All of the languages except English require a license key to enable. If you do not have the license key for one of the supported languages, it is treated as a generic language, and each word is stemmed to itself and it is tokenized in a generic way (on whitespace and punctuation characters for non-Asian characters, and on each character for Asian characters). For more information, see “Generic Language Support” on page 302. The language-specific collations are available to all languages, regardless of what languages are enabled in the license key.

26.0 Data Visualization Widgets

This chapter introduces the use of visualization widgets to display MarkLogic Server search results. These widgets provide a means to display data in picture or graph form. You can incorporate visualization widgets directly in the display output of applications you write, or incorporate them in applications created with Application Builder.

The following is the search page from the Oscars application created by Application Builder. Every object on the page is a visualization widget.



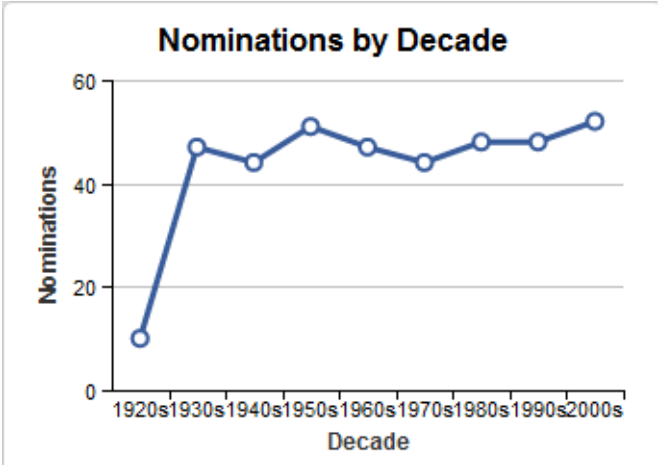
This chapter contains the following sections:

- Overview of Visualization Widgets
- Working with the Visualization Widgets
- Overview of the Widget Architecture
- Adding Visualization Widgets to an HTML Page
- Example: Adding Widgets to Applications
- Visualization Widget Limitations
- Set Up A Proxy
- Widget API Reference

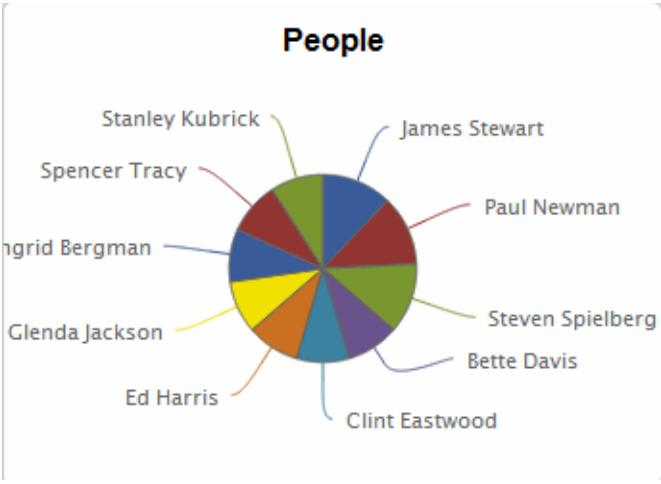
26.1 Overview of Visualization Widgets

MarkLogic Server provides six visualization widget types:

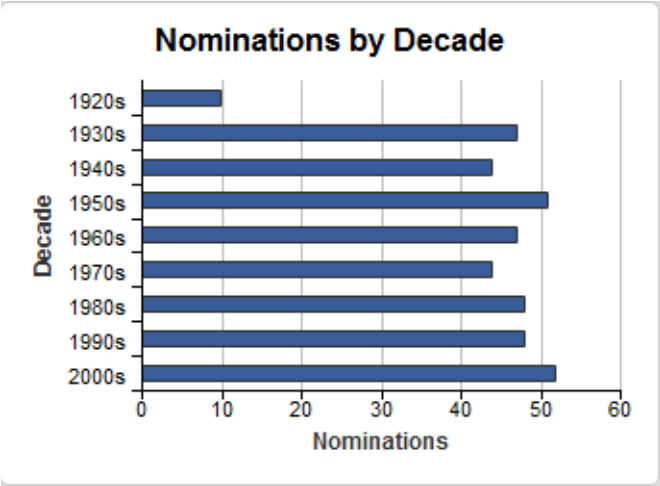
- Line Chart; a two-dimensional graph that shows a left to right series of data points, with each two consecutive data points connected by a straight line. Though line graphs can be associated with any facet type, they are typically used to show a time series, with the line representing chronological movement.



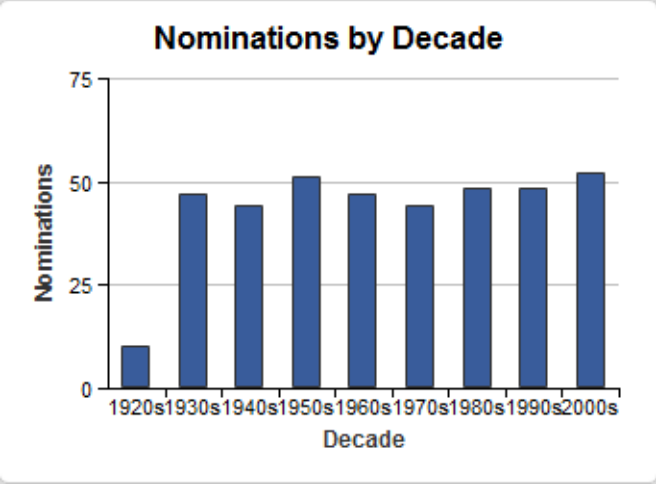
- Pie Chart: a two-dimensional circle, divided into different colored sections, one for each data value. A section's percentage of the circle's area is equivalent to the percentage of the data value's contents relative to the whole of the data (for example, if there are 16 total datums, and 8 of them have the value "red", the pie chart section corresponding to the "red" value will be half of the circle). Data values can be either non-continuous (colors, restaurant dishes) or continuous (height, weight, test scores).



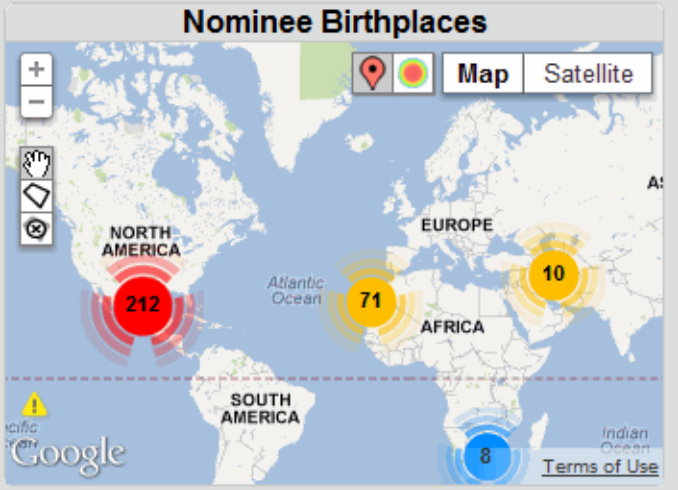
- Column Chart: a chart with horizontal rectangular bars representing the amount of a particular discrete value. Values can be either non-continuous (colors, restaurant dishes) or continuous (height, weight, test scores).



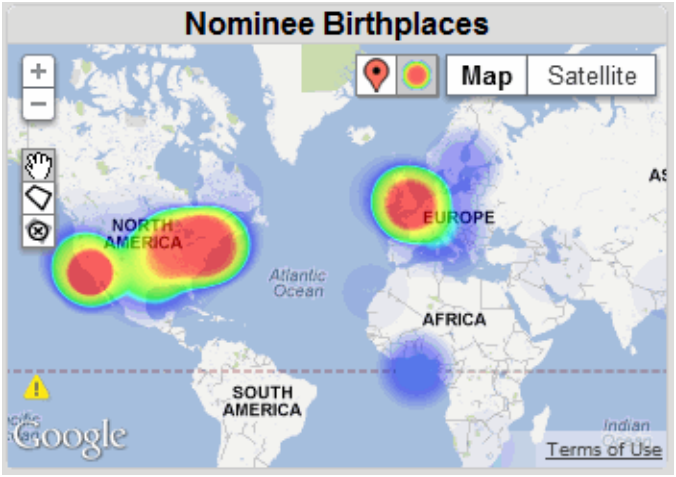
- Bar Chart: a chart with vertical rectangular bars representing the amount of a particular discrete value. Values can be either non-continuous (colors, restaurant dishes) or continuous (height, weight, test scores).



- Point Map: A Google map with data points representing search results, each associated with a geospatial value. For example, assuming the data is in the database and the map widget was configured for this, a pin would appear over the town of Nevada, Missouri, to indicate the birthplace of actor/director John Huston. You can zoom in and out on the map, as well as pan in all directions.



- Heat Map: A Google map with results' geographic frequency represented as colored blobs located at distinct points on the map. For example, if there was a large number of results located in San Carlos, CA, there would be a large blob centered on that location on the map. If there was a small number of results located in Los Angeles, there would be a small blob centered on that map location.



26.2 Working with the Visualization Widgets

This section contains the following subsections:

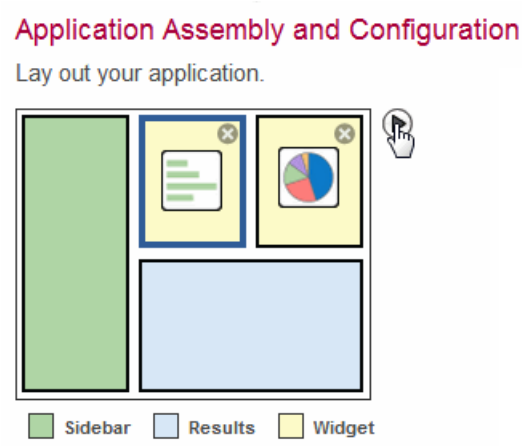
- Building The Oscars Example Application
- General Operation of Widgets
- Working with the Line Chart Widget
- Working with the Bar and Column Chart Widgets
- Working with the Pie Chart Widget
- Working with the Map Widget

26.2.1 Building The Oscars Example Application

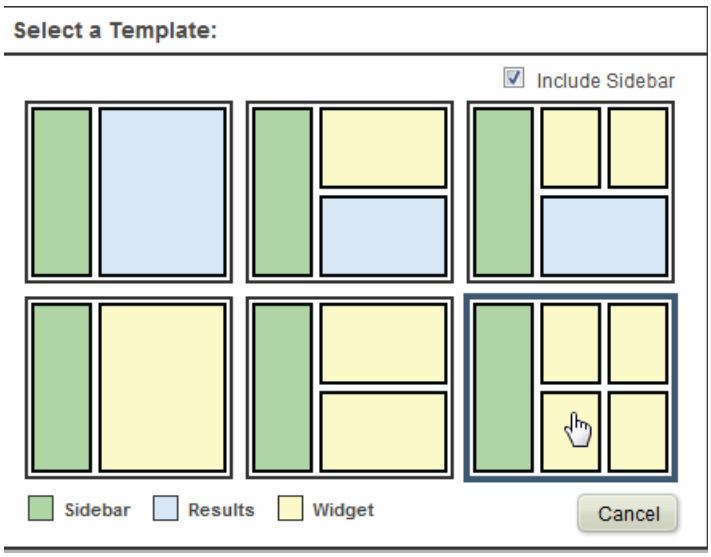
The quickest and easiest way to learn how to use visualization widgets is to use Application Builder to build an example Oscars application, as described in Building the Oscars Sample Application in the *Application Builder Developer's Guide*.

To add all of the widget types to the Oscars application, navigate to the Assemble page in Application Builder and do the following:

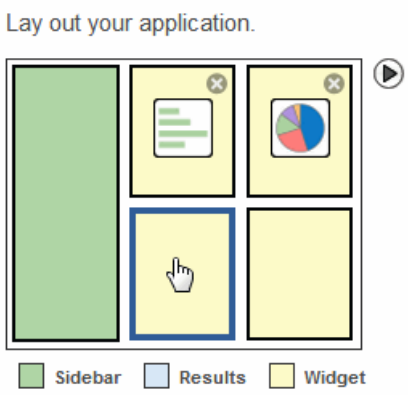
1. Click on the circled pointer at the right of the "Layout your application" section in the top left of the page.



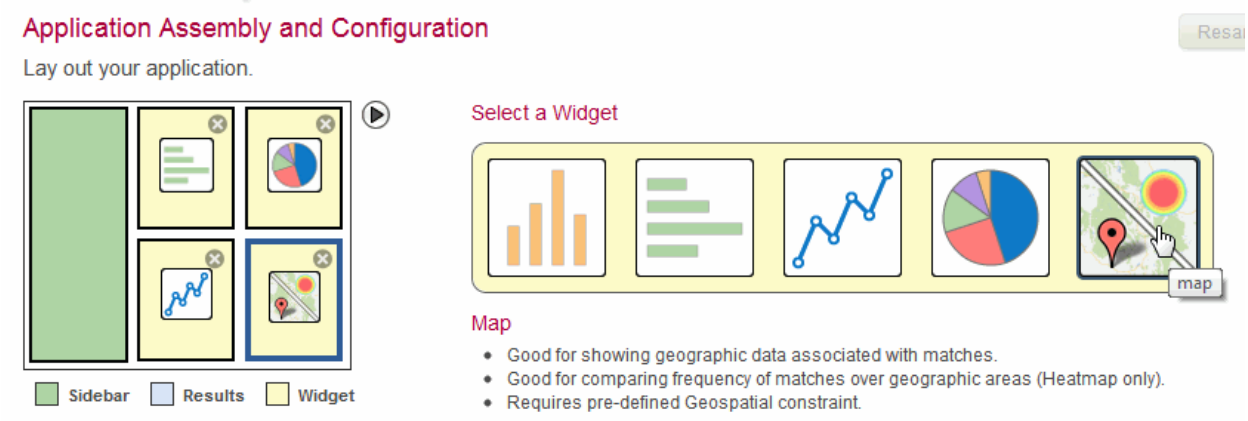
2. A “Select a Template” popup window appears, giving you layout choices for your applications results page. Select the four-widget layout at the bottom on the right.



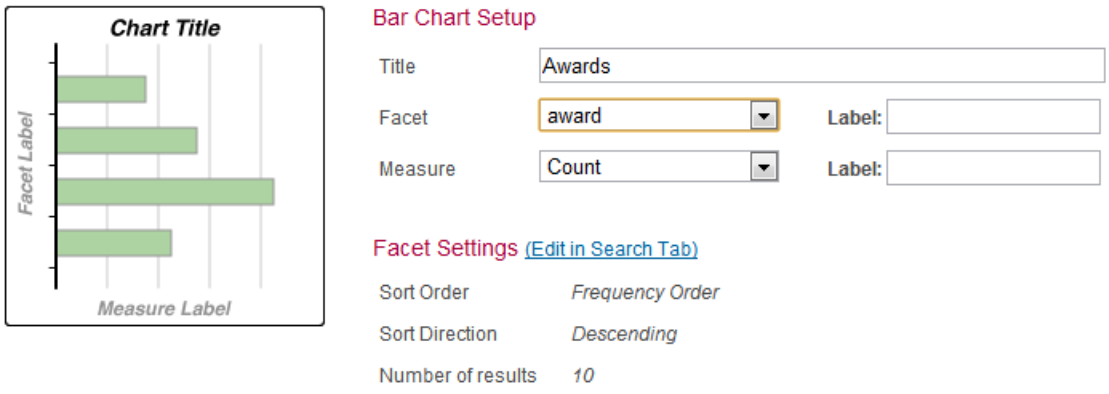
3. Click on the each widget box in your application layout:



4. Select the widget on the right that you want to add to the layout:



5. Each widget must be associated with a facet, which is set at the bottom of the Assemble page. For example, to have the column chart represent the number of awards on the data set, set the facet to ‘award’ and the measure to ‘Count’:



Note: Currently, Count is the only available measure for widgets.

6. Deploy your application, as described in Deploy Page in the *Application Builder Developer’s Guide*.

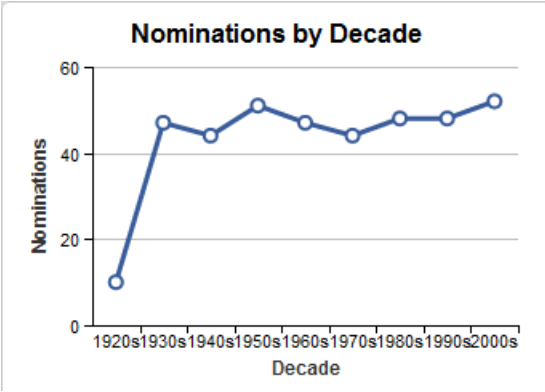
26.2.2 General Operation of Widgets

In general, clicking on a widget item (for example, a bar on a bar chart) changes the search query for the page and, as a result, the sidebar, results, and any other widgets on the page are updated to reflect the new search term. The only exceptions to this is the line chart widget and clicking on a marker in a map widget.

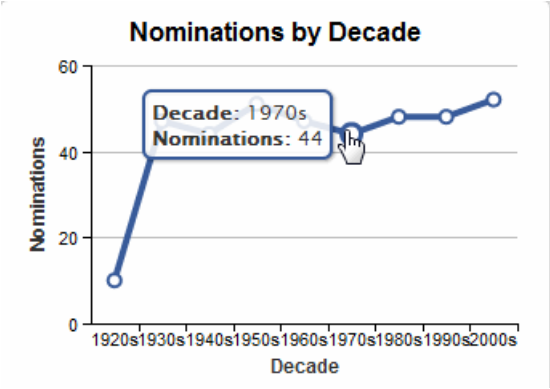
Searches initiated through the widgets are stemmed searches, which means the results returned for a term like “name” also include ‘names,’ ‘named’, and so on.

26.2.3 Working with the Line Chart Widget

The following Line Chart shows the search results on the "decade" facet, with each dot indicating the number of nominations during that "decade." The chart's y-axis is the number of nominations, and its x-axis is the year, going from the first Oscars in 1920 to the present.



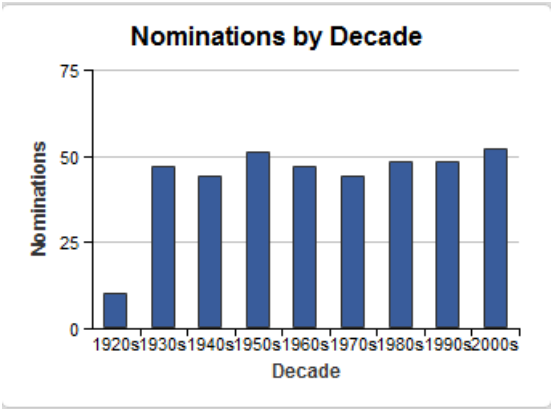
If you mouse over the Line Chart, a tool tip window appears, giving the date and number of nominations associated with the nearest data point, for example "Decade: 1970s" and "Nominations: 44"

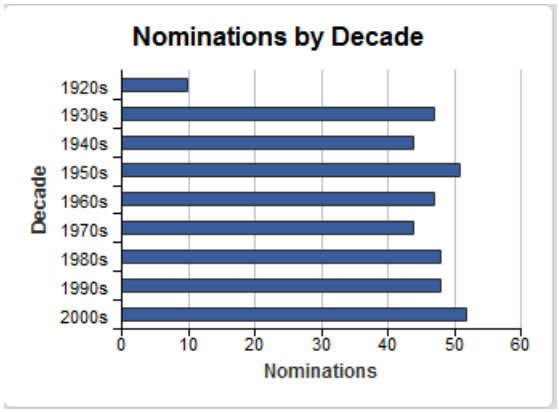


Note: Line charts are currently read-only and do not update the sidebar, results, or other widgets on the page.

26.2.4 Working with the Bar and Column Chart Widgets

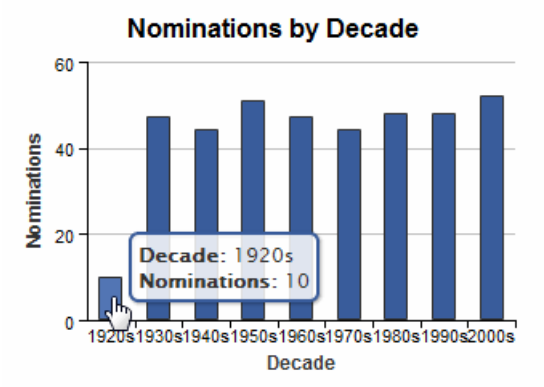
The following Bar and Column Charts show the search results on the “decade” facet.





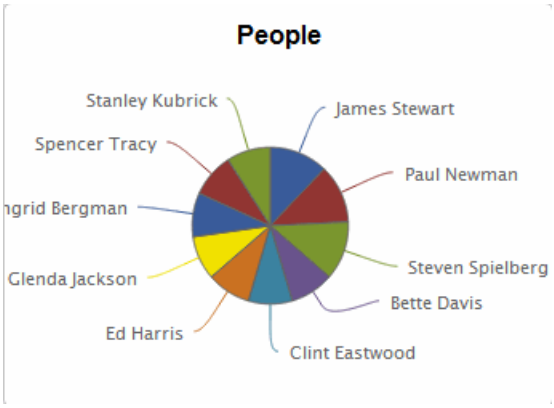
If you mouse over a Bar or Column Chart, a tool tip window appears, giving the number of nominations (either by decade or film, depending on which chart) and film name or decade (again, depending on which chart) associated with the nearest bar. Clicking and dragging the mouse over the chart does not do anything. Clicking a single bar changes the Results area of the page to only show the search results associated with the film or decade, depending on the chart. Clicks on charts also change other widgets on the page, applying the value of the clicked bar as an additional search term to those widgets.

Note in the chart shown below, which has had the 1920s bar clicked, the number of Results is 10, the same as the number of results which occurred in the 1920s.

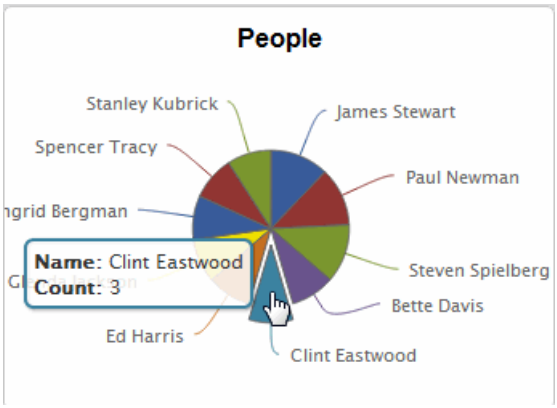


26.2.5 Working with the Pie Chart Widget

The following Pie Chart shows the search results on the “name” facet.

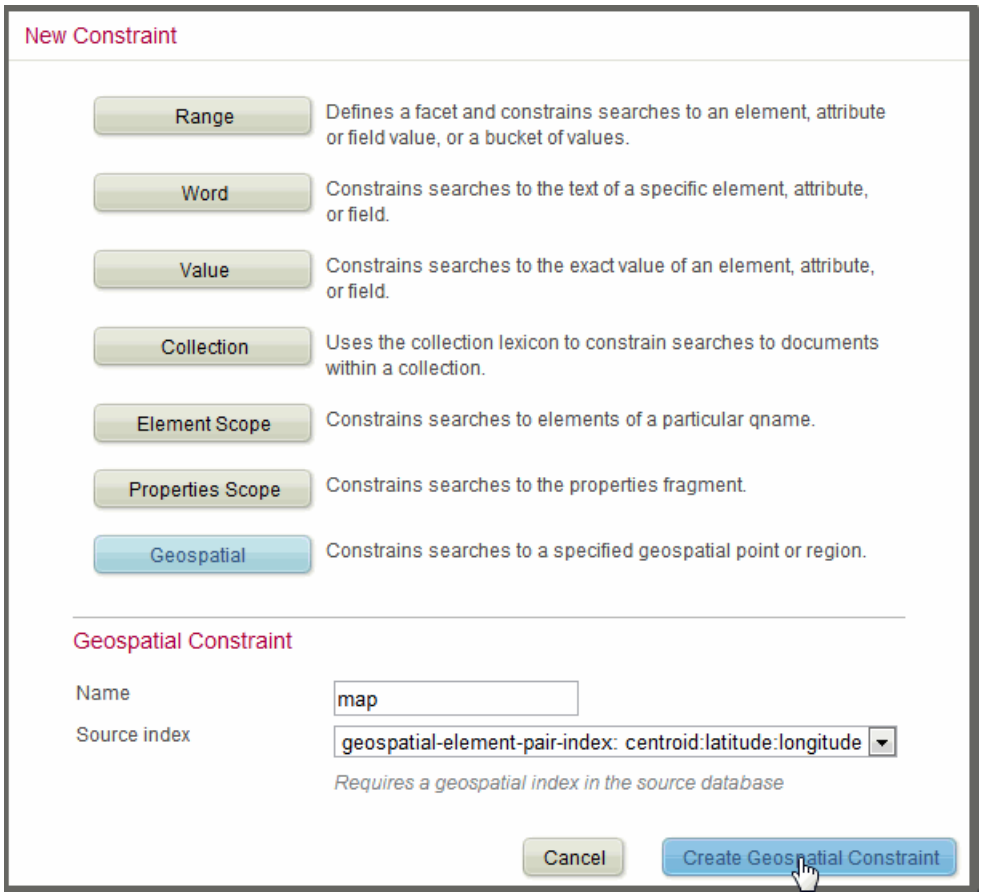


If you mouse over a Pie Chart, a tool tip window appears, giving the name value and number of search matching nominations associated with the nearest section, for example “Clint Eastwood: 3”. Clicking on a single Pie Chart section updates the Sidebar and Results sections, so that they only show the results for the Name value associated with that Pie Chart section. For example, if you click on the section for Clint Eastwood, only search results associated with Clint Eastwood appear in the page’s Sidebar and Results sections.

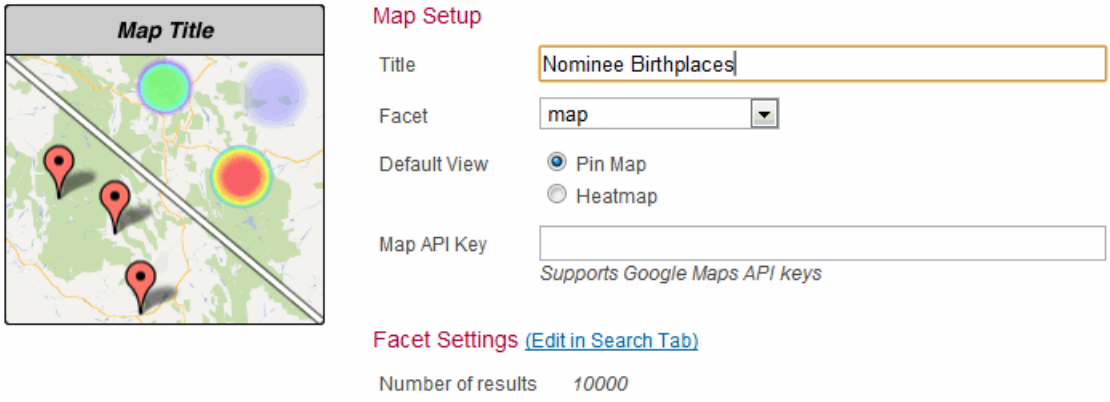


26.2.6 Working with the Map Widget

Before you can select a map widget, you must create a geospatial element pair index. Navigate to the Search tab in Application Builder, click Add New at the bottom of the page, select Geospatial, and name it 'map'. Click Create Geospatial Constraint.

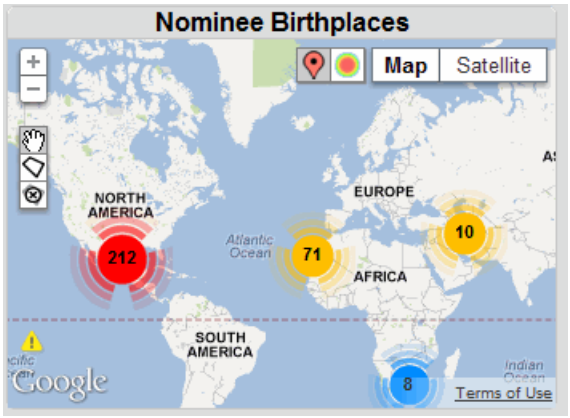


Navigate to the Assemble page, select the map widget, and assign it the map facet. Deploy the application.



Note: If you are writing an application in which you expect over 25,000 map loads per day, you will be required to purchase an a Map API Key from Google. Enter the key in the Map API Key field in the Map Setup section.

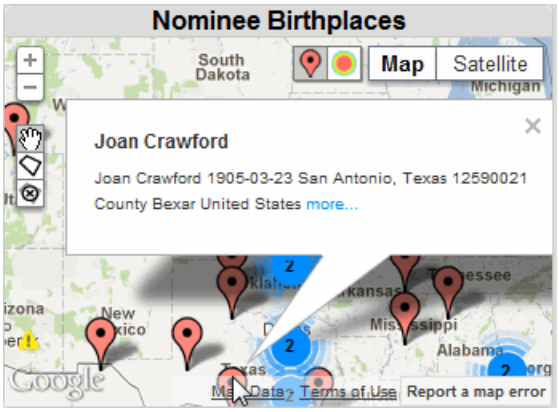
The map widget in the deployed application shows the search results on the entire Oscars data set. Each circle represents a cluster of markers. The color of and the number inside the marker cluster represents the number of results. If you select facets to narrow the data set, the map will change to represent that subset of results.



Clicking and dragging the mouse over the map widget causes the map display to pan or move in the drag direction. If you click on marker cluster, you can drill down to more marker clusters and finally to individual markers, which represent each nominee's birthplace.



Unlike the other widgets, clicking on a marker does *not* update the Sidebar or Results. Rather, it open an info window containing the actor's name, birth date, and birth place, such as "Joan Crawford / Born 1905-03-23 / San Antonio, Texas".



The Sidebar and Results will be updated if you use the Draw Shape tool to select one or more markers on the map. Click on the polygon icon to set the map to Draw Shape mode:



For example, to update the search results based on two markers, left-click points and drag lines around the markers to draw a polygon as shown below:



26.3 Overview of the Widget Architecture

The front end architecture of the widgets is based on modules that can both send messages and receive them, but no module has any knowledge of other modules or the page at large. Each module has a listener that listens for messages that meet specific parameters. Similarly, a module sends out messages into the common medium without specific knowledge of the recipient of that message, but other modules with the proper listeners will receive those messages. In this fashion, module can be added and removed in realtime without disrupting the flow of the application, and there is no need for a controller to keep tabs on all of the application's modules. This maintains a strong separation of interests and allows for easier unit testing.

Modules are not aware of other modules. They can only trigger events and listen for events. The only information exchange between modules is through listening to messages in the common medium. There is no tracking of the particular source of information or particular destination. Messages are released into the common medium and read by those modules that have the proper listeners for picking up those particular messages.

The widget architectural model makes heavy use of event bubbling in the Document Object Model (DOM). There are two basic types of modules: Widget modules and controller modules. In most instances, there is a single controller module and one or more widget modules.

This section contains the following topics:

- Widgets
- Controller
- Identification
- Interaction within a Widget
- Shadow Queries

26.3.1 Widgets

Using event handlers, widgets listen for newData events in order to listen for new pieces of data to render. Similarly, widgets can generate newQuery events in order to communicate a change in the query based on internal changes to the widget.

Two simple degenerate cases are a search bar widget and a results widget:

- The search bar is a widget which does not react to any newData messages, but it does generate newQuery messages in response to a user typing a text query and clicking Search.
- The results widget does not typically send out newQuery messages, but rather, merely renders the results of a query. The exception to this is, if results have been implemented with pagination, then the results widget will send newQuery messages in order to paginate.

Widgets usually use a hybrid of the two types, both displaying information and updating the query in order to update the view into that query. An individual widget doesn't necessarily know the whole query, but tends to have control over a portion of it. Most visualizations take ownership of a particular facet and apply or remove a filter on the search based on that facet.

26.3.2 Controller

The controller is a module just like a widget. It's main job is listening to newQuery events and posting newData events. The controller makes use of a private comm (communicator) object that is responsible for making Ajax calls to the server for new data. For example, when panning a map widget, the map needs to update the marker clusters, which requires a query. The map widget sends out an event that the controller hears and passes on to the comm object, which requests the data and sends it back to the map widget.

The controller and comm object offer a lot of extensibility room for features like caching and data manipulations. The controller accepts an option configuration object that specifies things like the endpoint to access on the server, the application wrapper, the widget class, etc. By specifying a limited application wrapper, multiple controllers, for example, can be created on the page each running a separate application. They only respond to messages within their application container. Similarly, two applications can share a container on the page and use different widget classes to control their individual widgets.

The controller receives newQuery events and translates the query object into a structured query, which is appended to the Ajax call generated in the comm object.

26.3.3 Identification

Widgets are identified and linked to a controller using an HTML class. Applications generated by Application Builder name this class “widget.”

For example, to add the results widget to the page, specify the following:

```
<div id="results" class="widget"></div>
```

26.3.4 Interaction within a Widget

Interactions within widgets consist of selecting data or pagination. Each widget has an internal `updateQuery` method that can be called with a query object to send out a newQuery event. In the case of visual widgets, it makes "selections" on facets to update the query. A column chart or pie chart takes a single selection and a map takes an area (a polygon). For example, clicking on a bar in a column chart generates a selection event which bubbles up from the DOM and is read by the widget, which translates the selection into a newQuery event. The translation logic lives in the widget.

26.3.5 Shadow Queries

The basic idea of a shadow query is that, with many widgets on a page, multiple selections can be made at once and, if widgets are updating with each search, clicking on a widget essentially nullifies its further usefulness in searching because it will only display the one result that is the current selection.

The idea of shadow queries is to provide a “context” or “shadow” for these selected widgets. When a widget is selected, the selection is highlighted, but the other categories remain visible while the other widgets on the page update. When the next widget is selected, two queries are fired, one is the general search that's updating unselected widgets and the results pane; the other is the “shadow query” for the selected widget. This shadow query is a copy of the main search query, except without the selected widget's contribution to that search. This renders a view "around" the selection in the widget and greatly expands the power of widgets on the page by offering a very deep view into the data.

Shadow Queries work by taking advantage of the datastream. The datastream is set in two places, once in each widget upon instantiation (`datastream` is a parameter in the `createWidget` method described in “ML.createWidget (Null Widget) Method” on page 345) and also upon creation of shadow queries in the controller. The datastream is used to match a shadow query to the widget that it is shadowing.

Here is how datastreams work:

- If a query has no datastream, it will update all unselected widgets (or widgets with no datastream of their own set).
- If a query has a datastream and it matches the widget's datastream, then the widget will update; whether the widget is selected or not.
- If a query has a datastream and it does not match the widget's datastream, then the widget will do nothing.

  Note: When a widget sets off a shadow query by making a selection, it will not render the next update it receives, as that update is informing the widget of its already current state. This is called shadowBack internally.

26.4 Adding Visualization Widgets to an HTML Page

Some code is common to both chart widgets (Bar, Column, Line, and Pie Charts) and map widgets (Point and Heat).

26.4.1 Common JavaScript And CSS In The <head> Element

In your `<head>` element, first, add your external library dependencies for line, bar, column, and pie charts:

```
<!-- external library dependencies -->
<script src="lib/external/jquery-1.7.1.min.js"
        type="text/javascript"></script>
<script src="lib/external/highscharts.src.js"
        type="text/javascript"></script>
```

If you are using map widgets, you must also include the following external libraries:

```
<!-- external web map files... -->
<script
src="http://maps.googleapis.com/maps/api/js?key=&amp;sensor=false&amp;
region=US&amp;libraries=drawing"
type="text/javascript">
</script>
<script src="lib/external/mxn-2.0.18/mxn.js?(googlev3)"
type="text/javascript"></script>
<script src="lib/external/heatmap.js" type="text/javascript"></script>
<script src="lib/external/heatmap-gmaps.js"
type="text/javascript"></script>
<script src="lib/external/markerclusterer_min.js"
type="text/javascript"></script>
```

Then add calls to the internal widget framework library. Of course, if you are not using histograph widgets, you do not need to include the `chart.js` library, and if you are not using map widgets you do not need to include the `map.js` library:

```
<!-- internal widget framework library -->
<script src="lib/controller.js" type="text/javascript"></script>
<script src="lib/widget.js" type="text/javascript"></script>
<script src="/lib/viz/chart/chart.js" type-"text/javascript"></script>
<script src="/lib/viz/map/map.js" type-"text/javascript"></script>
```

Finally, you may want to define a CSS style for your widget elements: This is optional, as there is a set of default CSS styles for the various widget types in the library (see `/lib/viz/chart.css` and `/lib/viz/map.css`). If you do define a custom style, do something similar to the following, defining your widgets' display parameters to meet your application's requirements. Of course, you do not have to call your element `widget`, especially if you want to have different display parameters for different visualization widgets and thus need to define several different elements.

```
<style type="text/css">
       .widget {
                width: 500px;
                display: block;
                float: left;
               }
</style>
```

26.4.2 Common Code In The <body> Element

You must define each widget as a separate `<div>` element with either the default class `"widget"` or the class name you specified when initializing widgets with `ML.controller.init` and its `widgetClass` option (see "ML.controller Methods" on page 343). In the example later on, which has three visualization widgets (two charts and one map), you have:

```
<div id="decadeContainer" class="widget"></div>
<div id="actorContainer" class="widget"></div>
<div id="locationContainer" class="widget"></div>
```

Next, define a `<script>` that:

1. Define the PHP search proxy that authenticates with the server and forwards queries to the REST endpoint on MarkLogic Server: :

   ```
   ML.controller.init({proxy: "proxy.php"});
   ```

   For details on how to define the PHP proxy, see "Set Up A Proxy" on page 342

2. Still in the `<script>`, define configuration variables for each of your visualization widgets. The configuration parameters differ for chart and map widgets (and also for Point Map and Heat Map map widgets), so see below for detailed examples and later for a complete list of possible configuration parameters for all visualization widget types. But placeholders for the three widgets in our full example would look like:

```
var yearConfig = {
...chart configuration parameters...
}

var actorConfig = {
...chart configuration parameters...
}

var mapConfig = {
...point and heat map configuration parameters...
}
```

3. Finally, activate the widget controller and the widgets themselves and close the `<script>`. `ML.controller.init` takes as its argument the variable you defined back in 1) that contains the information about the proxy server and this application's search endpoint, in this case `config`.

   `ML.chartWidget` and `ML.mapWidget` both have the same signature of widget element name, widget type, and configuration parameters variable name, and only differ in whether a chart or map widget is created.

```
ML.controller.init(config); //connect to the var with our proxy value

//chartWidget can take 'line' or 'bar' or 'column' or 'pie'
//mapWidget just takes 'map'
ML.chartWidget('decadeContainer', 'line', yearConfig);
ML.chartWidget('actorContainer', 'bar', actorConfig);
ML.mapWidget('locationContainer', 'map', mapConfig);
```

26.4.3 Initialize the Display

To initialize the widgets' display, use the use the `ML.controller.loadDat`a method of the controller:

```
ML.controller.loadData();
```

End the script with the usual `</script>`.

26.5 Example: Adding Widgets to Applications

This section describes how to add visualization widgets to an existing search application. In order to add visualization widgets, you must do the following:

- Obtain a set of files for creating the visualization widgets.
- Create an App Server that implements an instance of the MarkLogic REST API.
- Set up a proxy to connect to the REST server. For this document, we make use of a PHP server to host the proxy.
- Define a search options node to expose facet data to the widgets.
- Create an HTML page that configures and displays the widgets in your application.

It is assumed that you have a MarkLogic database set up when you begin the following process.

1. Use either the Admin interface or Information Studio to create a database, named `restaurants`, in MarkLogic Server.

2. Create a REST server for the database, as described in Creating a REST API Instance in the *Information Studio Developer's Guide*. This will automatically create a modules database that starts with the name of the REST server, followed by `-modules`. For example, if you name the REST server `restaurants`, a database, named `restaurants-modules`, is automatically created. In this example, the REST server is located at: `myhost:5437`.

3. Using the Admin interface, create the following element range indexes in the `restaurants` database. These correspond to the facets to visualize using the widget:

scalar type	localname
`string`	`Type`
`int`	`SqFtEst`

4. Use Query Console execute the following query to load some documents into your `restaurants` database:

```
xquery version "1.0-ml";

xdmp:document-insert(
  "/1.xml",
  <restaurant>
    <id>1</id>
    <DBA> 123 Deli - Lee's </DBA>
    <Type> Restaurant, &lt; 500 sq' </Type>
    <SqFtEst> 500 </SqFtEst>
    <StreetAddress> 123 1st St </StreetAddress>
    <City> San Francisco </City>
    <State> CA </State>
    <FullAddress> 123 1st St, San Francisco CA </FullAddress>
    <Latitude>37.78946</Latitude>
    <Longitude>-122.39717</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/2.xml",
  <restaurant>
    <id>2</id>
    <DBA> Dave's Dive </DBA>
    <Type> Restaurant, &lt; 700 sq' </Type>
    <SqFtEst> 700 </SqFtEst>
    <StreetAddress>2 1 Mission St </StreetAddress>
    <City> Bismarck </City>
    <State> ND </State>
    <FullAddress> 21 Mission St, Bismarck ND </FullAddress>
    <Latitude>46.48</Latitude>
    <Longitude>-100.47</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/3.xml",
  <restaurant>
    <id>3</id>
    <DBA> Wayne's Steakhouse </DBA>
    <Type> Restaurant, &lt; 1000 sq' </Type>
    <SqFtEst> 1000 </SqFtEst>
    <StreetAddress> 444 Feick St </StreetAddress>
    <City> Albany </City>
    <State> NY </State>
    <FullAddress>444 Feick St, Albany NY</FullAddress>
    <Latitude>42.40</Latitude>
    <Longitude>-73.45</Longitude>
  </restaurant>),
```

```
xdmp:document-insert(
  "/4.xml",
  <restaurant>
    <id>4</id>
    <DBA> Megha </DBA>
    <Type> Restaurant, &lt; 1000 sq' </Type>
    <SqFtEst> 1000 </SqFtEst>
    <StreetAddress> 4531 Front St </StreetAddress>
    <City> Carlsbad </City>
    <State> NM </State>
    <FullAddress> 4531 Front St, Carlsbad NM </FullAddress>
    <Latitude>32.26</Latitude>
    <Longitude>-104.15</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/5.xml",
  <restaurant>
    <id>5</id>
    <DBA> Gordon's Grubhaus </DBA>
    <Type> Restaurant, &lt; 300 sq' </Type>
    <SqFtEst> 300 </SqFtEst>
    <StreetAddress> 3421 Capital Wy </StreetAddress>
    <City> El Paso </City>
    <State> TX </State>
    <FullAddress> 421 Capital Wy, El Paso TX </FullAddress>
    <Latitude>31.46</Latitude>
    <Longitude>-106.28</Longitude>
  </restaurant>)
```

5. Use Query Console to load the following search options node into the `restaurants-modules` database. This exposes the facets required for the visualizations.

   Note: You must be logged into MarkLogic Server as a user with the `rest-admin` privilege to run this query.

```
xdmp:http-put("http://myhost:5437/v1/config/query/all",
<options xmlns="xdmp:http">
  <authentication method="digest">
    <username>admin</username>
    <password>admin</password>
  </authentication>
  <headers>
    <content-type>application/xml</content-type>
    <accept>application/xml</accept>
  </headers>
</options>,
text { xdmp:quote(
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="Type">
    <range type="xs:string" facet="true">
      <element ns="" name="Type"/>
      <facet-option>limit=10</facet-option>
      <facet-option>frequency-order</facet-option>
      <facet-option>descending</facet-option>
    </range>
    </constraint>
    <constraint name="SqFtEst">
      <range type="xs:int" facet="true">
        <element ns="" name="SqFtEst"/>
        <facet-option>limit=5</facet-option>
        <facet-option>frequency-order</facet-option>
        <facet-option>descending</facet-option>
      </range>
    </constraint>
</options>
) }
)
```

6. Enter the following URL, where `<server>:<rest-port>` identify your REST server. You should see an options node file with all your constraints set up.

```
http://<server>:<rest-port>/v1/config/query/all
```

7. Navigate to the public directory for your PHP-enabled Apache server. In this example, this is `/var/www/html`.

8. Download the files for the visualization widgets from:

```
http://localhost:8000/appbuilder/customappfiles.xqy
```

9. Unzip the `customappfiles.xqy` file and copy the five subfolders (constraint, css, images, lib, and skins) to the public directory for your PHP-enabled Apache server.

   Note: In this example, there is no 'application' directory, as there is in applications created by Application Builder.

10. Create a PHP proxy file, named `proxy.php`, in the public directory for your PHP-enabled Apache server with the following contents. Make sure the `CURLOPT_URL` option points to your `restaurants` server (`myhost:5437`, in this example) and that the `CURLOPT_USERPWD` option contains your login credentials for MarkLogic Server.

```
<?PHP

$queryString = $_SERVER['QUERY_STRING'];
parse_str($queryString, $vars);
unset($vars['proxyPath']);
$queryString = http_build_query($vars);

$options = array(
CURLOPT_URL => 'http://myhost:5437' . $_GET["proxyPath"] . '?' .
$queryString, // location of the rest server. Be sure to preserve the
query string unaltered

CURLOPT_HTTPAUTH => CURLAUTH_DIGEST, // it is very important to set the
authentication type to digest

CURLOPT_USERPWD => 'admin:admin', // this is your username and password
for the server. be sure your role has read permissions on the database

CURLOPT_HTTPHEADER => array('Content-type: application/json'), // the
server won't respond properly unless the content-type is
application/json
);

// Only set POST options for POSTs, not GETs
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
$options[CURLOPT_POST] =  1; // queries are sent in the post body, with
options like pagination in the query string

$options[CURLOPT_POSTFIELDS] = file_get_contents('php://input'); // we
are not sending a traditional form map through the post, so avoid using
the standard $_POST
}

$ch = curl_init(); // create curl
curl_setopt_array($ch, $options); // set curl options
curl_exec($ch); // execute curl

?>
```

11. Create an HTML file with the following contents in the public directory for your PHP-enabled Apache server.

```
<html>
<head>
  <title>Custom App</title>

  <!-- external library dependencies -->
  <script src="lib/external/jquery-1.7.1.min.js"
    type="text/javascript"></script>
  <script src="lib/external/highcharts.src.js"
    type="text/javascript"></script>

  <!-- internal widget framework library -->
  <script src="lib/controller.js" type="text/javascript"></script>
  <script src="lib/widget.js" type="text/javascript"></script>

  <!-- visualization framework libraries -->
  <script src="lib/viz/chart/chart.js" type="text/javascript">
  </script>
  <link media="screen, print" href="lib/viz/chart/chart.css"
    rel="stylesheet" type="text/css">

  <style type="text/css">
    .widget {
       width: 500px; display: block; float: left;
    }
  </style>
</head>

<body>
  <h2>MarkLogic Custom Application</h2>
  <div id="typeContainer" class="widget"></div>
  <div id="sqftContainer" class="widget"></div>
  <script type="text/javascript">

<!-- If you have a proxy, identify it here. -->
  var config = { proxy: "proxy.php" };

  var typeConfig = { constraint: 'Type',
                     constraintType: 'range-unbucketed',
                     dataType: 'xs:string',
                     title: 'Type',
                     dataLabel: 'Type' };
```

```
  var sqftConfig = { constraint: 'SqFtEst',
                     constraintType: 'range-unbucketed',
                     dataType: 'xs:int',
                     title: 'Square Feet',
                     dataLabel: 'Square Feet' };

  ML.controller.init(config);
  ML.chartWidget('typeContainer', 'column', typeConfig);
  ML.chartWidget('sqftContainer', 'bar', sqftConfig);

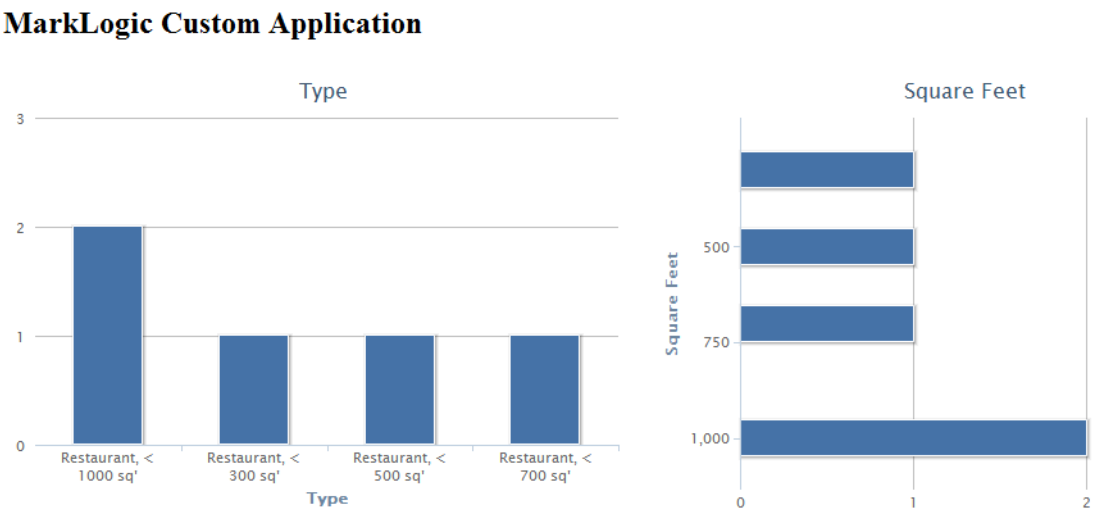
<!-- This will trigger the intial empty search -->
  ML.controller.loadData();
</script>
</body>
</html>
```

12. Make sure all of the files in the public directory for your PHP-enabled Apache server are publicly accessible.

13. In a browser, enter the URL for your PHP-enabled Apache server host. For example:

```
http://myhost/index.html
```

14. You should see a page like the following:



26.6 Visualization Widget Limitations

There are several limitations common to most or all widget types. These are:

- Non-map widgets size to their given container and do not scroll. Map widgets can be scrolled within the map itself, just as in most common map programs. You can zoom in or out, pan left, right, up, and down, etc. This does not affect the map's container, only the view of the map itself. The map can be set so as to disable zooming or panning. Panning is restricted North to South, so you are not allowed to pan past the North or South poles.
- Non-map widgets do not support multiple-dimension visualizations (i.e. one facet per widget).
- Map results are taken from the geo facet, rather than the search results themselves. … it maybe useful to highlight that and give a screen shot of /v1/search?options=all&view=facets
- In maps, panning and zoom generate a new query that is consumed by the map to 'paint' markers and marker clusters. The reason for this approach is to have an efficient mechanism to display maps that potentially have large data sets so that data queries are incremental, rather than returning all data points.
- Tooltip behavior is defined by the chart widget and cannot be customized through Application Builder

26.7 Set Up A Proxy

In order to run a custom app, you generally need to run your queries through a proxy in order to authenticate on your MarkLogic REST server instance.

There is a proxy variable in the controller configuration that should be set to the base path of your proxy. The controller will append a 'proxyPath' URL parameter that your proxy should read. The proxyPath will point to the absolute path on the MarkLogic REST sever instance that the query needs to be proxied to. If the proxy is called with a POST, it should pass along the POST data to the MarkLogic REST server instance unchanged. It should also pass all url parameters aside from the proxyPath parameter. In the case of a GET request, just the url parameters are needed. The responses should be passed straight through unaltered.

An example PHP implementation is described in "Example: Adding Widgets to Applications" on page 335. Note that several options, such as the location of the REST server and the username and password values, may not be the same as your values. Edit the code so that it uses your values. Put this code in a publicly accessible file on your PHP server, for example, `proxy.php`, which will serve as your application's search endpoint.

After confirming you have the proxy working correctly, add a `proxy` option to the `ML.controller.init` method in your `index.html` page to identify the PHP proxy. For example:

```
ML.controller.init({proxy: "proxy.php"});
```

26.8 Widget API Reference

This section serves as a reference to the configuration options for each visualization widget and related methods. Bar, Column, Line, and Pie Charts all have the same configuration options. While it may seem odd to include Pie Charts with the others, if you consider a "slice" of a Pie Chart as serving the same basic purpose as a bar/column/datapoint on the other charts, it should make sense.

The topics in this section are:

- ML.controller Methods
- ML.createWidget (Null Widget) Method
- General Widget Methods
- ML.chartWidget Method
- ML.mapWidget Method
- Widget Events
- Widget Query Structure
- Widget Query Update Structure
- Search Results Format
- Facet Structure
- Internal Widget Events

26.8.1 ML.controller Methods

Method	Description
`init`	Takes an optional config object made up of name/value pair configuration options in the above table. All values are optional and default to those used in the application built by Application Builder.
`getData(query)`	This is the direct method to call with a query for such actions as doing an initial load of data, or wanting to reset a search for some reason. The query object is required, but can be empty. For example: `ML.controller.getData({});`
`loadData`	Performs the initial data loading on the page. Will read the url for a bookmarked query if bookmarking is enabled. Call loadData() after initializing all the widgets.

The `ML.controller.init` method has the signature:

```
ML.controller.init ({option1: value1, option2: value2...});
```

Valid options are:

Option	Description
`appWrapper`	Lets you wrap your application in a section of the page so it doesn't interfere with the rest of it. This is the uppermost element of the widget application. It defaults to the body tag, encompassing the whole page, but can be passed any jQuery selector to allow for multiple separate widget applications to be on a single page (for example, if you want to display data from different databases) or as a safety measure for preventing interference with an exiting application. Events and interactions are completely contained within the appWrapper. Default: `body`
`version`	Version of the MarkLogic Server REST API. Default: `v1`
`optionsNode`	The options node that contains the constraints being queried against. Defaults to 'all', which is the name of the options node generated by Application Builder. Default: `all`
`startPage`	Sets the default starting page for results pagination. This does not affect visualization widgets, just the end search results. Default: `1`
`useShadows`	Turns shadows queries on or off globally in the application. Default: `true`
`widgetClass`	This is the jQuery selector that identifies widgets within the app. Any element with the class `widget` will receive newData events automatically in the default setup. Default: `.widget`

Option	Description
`proxy`	If using a proxy, provide the path to the proxy here. See proxies section for instructions on setting up a proxy. Default: `false`
`enableBookmarking`	Turns on bookmarking support. Default: `true`
`bookmarkingDelimiters`	Delimiters to be used encoding the current query in the URL. Default: `['*_*', '*__*']`
`pageSize`	Number of results returned for a query (does not affect constraints displayed in widgets). Default: `10`

26.8.2 ML.createWidget (Null Widget) Method

The 'null widget' as it is called, is the core class of a widget in the application. Visualizations such as charts and maps as well as controls like the search bar or displays like the results pane are all extensions of the basic null widget, which handles general communication behavior and handles incoming data.

```
ML.createWidget(container, renderCB, datastream, constraintType)
```

Parameter	Description
`container`	Required id of the container element for the widget. Recommended to be a div.
`renderCB`	Optional A callback function to execute when the widget receives a 'newData' event. The function is passed the server's response for the purpose of rendering those results.
`datastream`	Optional* An identifier that allows a widget to respond to shadow queries. Generally should be set to the constraint name for the widget

Parameter	Description
`constraintType`	Optional* the constraintType must be set if the widget is using faceted data. It must match the type of constraint on the server's index for that constraint. Must be "range", "constraint", or "geo". Determines how the structured query is generated. Map data is always of type geo. Other data can be range or constraint, depending on whether they are range indexes or collections on the server.

*required for shadow queries to work

26.8.3 General Widget Methods

Method	Description
`updateQuery(queryUpdate)`	Takes a queryUpdate object as an argument (see controller objects above) and causes the widget to trigger an updateQuery method. Abstract method for widgets without selection that need to update the query, like a searchbox.
`getSelection()`	Returns the currently selected value of the widget if it exists, null otherwise.

26.8.4 ML.chartWidget Method

The method used to create line, bar, column, and pie widgets is:

```
ML.chartWidget(containerID, type, config);
```

Parameter	Description
`containerID`	Id of the container element for the widget. Recommended to be a `div`. (required)
`type`	Type of chart. This can be: • `line` - line chart • `bar` - bar graph • `column` - column chart • `pie` - pie chart
`config`	The configuration object consisting of the options listed in the table below.

The `ML.chartWidget` configuration options are:

Option	Description
`constraint`	Name of the constraint facet whose data the chart renders. If invalid, the chart will not render. Defaults to `""`.
`constraintType`	Type of the constraint facet whose data the chart renders. Used to construct the query. Allowed values: `range`, `constraint`, `geo`. Value must specify the type of constraint the query is being run on in order to run the search. If invalid, either the query will not execute or cause an error on the server. Defaults to `""` (no actual default value).
`title`	String of the widget's displayed title. Defaults to `""`.
`subtitle`	String of the widget's displayed subtitle. Defaults to `""`.
`dataLabel`	String of the displayed label of type of values in the graph. Defaults to `""`.
`dataType`	String of the type of data displayed in the graph. Use the datatype of the constraint in the database. To disable the setting of the range, set `dataType` to `"string"`. Takes `"string"`, `"datetime"`, `"int"`, etc. Defaults to `"string"`.

Option	Description
`includeOthers`	When `true`, an "other" category is in the chart for when results are limited. For example, when showing the top 10 actors in terms of number of Oscar nominations as well as an others category for the total number of other actors with an Oscar nomination. Takes `true` or `false` Defaults to `false`.
`othersLabel`	String of the displayed label for "other" results as described in `includeOthers`. Defaults to `"Other"`.
`hideXAxisValues`	When `true`, hides the x-axis labels otherwise seen below the graph. Useful when dense columns result in unreadable labels. *Note*: Pie Charts have no x-axis, and Column Charts' x-axis is the vertical, not horizontal, axis. Takes `true` or `false`. Defaults to `false`

26.8.5 ML.mapWidget Method

The method used to create map widgets is:

```
ML.mapWidget('locationContainer', 'map', mapConfig);
```

The following are configuration options for map widgets. They are used to define a configuration variable similar to:

```
var mapConfig = {
       constraintType: 'geo',
       dataStream: 'results',
       statusContainerId: 'debug',
       width: 698,
       height: 512,
       ...
};
```

The point map configuration options are:

Option	Description
`geoConstraint`	Define a geographic constraint on the search results. Has the value of the geo constraint from the database associated with the map. Defaults to `""`.
`mapWidth`	Width of map (and its container) in pixels. Defaults to `485`.
`mapHeight`	Height of map (and its container) in pixels. Defaults to `300`.
`mapProvider`	Source of maps. At present, only the default value for Google Maps is supported. Defaults to `"googlev3"`.

Option	Description
`zoomControlType`	Show zoom control. Takes either `"small"` or `"large"` Defaults to `"small"`.
`showMapTypes`	Show the available types of maps (satellite, road, etc.). Takes either `"true"` or `"false"`. Defaults to `"false"`.
`showSelectControls`	Show the Map widget search selection controls for selecting an area as an overlay. Takes either `"true"` or `"false"`. "Defaults to `"false"`.
`showMapControls`	Show the Map widget controls (switch from Point to Heat Map and vice versa) overlay. Takes either `"true"` or `"false"`. Defaults to `"false"`.
`minZoomLevel`	Defines lower limit of map's zoom values. Defaults to `2`.
`maxZoomLevel`	Defines upper limit of map's zoom values. Defaults to `14`.
`constrainPan`	Determines if map's pan function is constrained (i.e. if true, the visual area of the map is fixed). Takes `"true"` or `"false"`. Defaults to `"true"`.
`constrainZoom`	Determines if map's zoom function is constrained (i.e. if true, you cannot zoom in or out). Takes `"true"` or `"false"`. Defaults to `"true"`.
`autoCenterZoom`	When true, auto centers on any object on the screen (markers, polygons) Takes `"true"` or `"false"`. Defaults to `"true"`.
`showMarkersOnLoad`	When true, shows the map pointers indicating search matches when the map is loaded. Takes `"true"` or `"false"`. Defaults to `"true"`.
`showHeatmap`	Determines if the map is a Heat Map (`"true"`) or a Point Map (`"false"`). Takes `"true"` or `"false"`. Defaults to `"false"`.
`defaultMarkerIcon`	Image used to display a search result hit on a map. Takes an image file. Defaults to `"/images/map_red_shadow.png"`.

In addition to the point map options above, heat maps also have the following options:

Option	Description
`heatMapRadius`	Larger values mean each point contributes to a larger "hot" area on the map. Takes an integer. Defaults to `15`.
`heatMapOpacity`	Takes an integer between `0` and `100` (least and most opaque) that determines entity opacity. Defaults to `90`.
`searchColor`	Defines a color for lines (polygon, circle, square, etc.) in search results. Takes an HTML hexadecimal color value. Defaults to `"#FF0000"` (red).

Option	Description
`searchFillColor`	Defines a color for fill in the search results, i.e. the search polygon area. Takes an HTML hexadecimal color value. Defaults to "".
`searchOpacity`	Defines the level of opacity in the color of search result circles. Takes a value between 0 and 1 inclusive, where 0 is least opaque and 1 is most opaque. Defaults to `1`.
`searchFillOpacity`	Defines the level of search fill opacity (i.e. in the search polygon area). Takes a value between 0 and 1 inclusive, where 0 is least opaque and 1 is most opaque. Defaults to `0`.

26.8.6 Widget Events

Event	Description
`updateQuery`	This event is posted by widgets making selections. The controller listens for this event bubbling up the DOM tree on the appWrapper element. See below for the data structure of the attached JSON object.

`newData`	This event is posted to all of the application's widgets with an attached Data object of the following form:
	
	`{`
	`  data: <server results>,`
	`  query: <query object for search>`
	`}`

`newQuery`	Resets the current query to the provided query instead of updating the current search query
`getBounds`	Gets the upper and lower bounds for the provided constraint from the values endpoint on the REST server. The constraint is attached to the event

`newBounds`	Publishes a bounds object to the widgets which will set the xAxis scale of the visualization widgets using constraint in the bounds object. It has the following form:
	
	`{`
	`  min: <min_value>`
	`  max: <max_value>`
	`  constraint: <constraint>`
	`}`

26.8.7 Widget Query Structure

Query Structure (can be empty)

Field	Description
`text`	Text query that is passed directly to server unaltered. Supports all the query syntax of previous build app versions. (optional)
`facets`	An array of facet objects representing selections from widgets on the page. See Facet Structure Below (optional)
`page`	Page number for use in paginating the search results (optional)
`datastream`	Used to determine which widgets will render the query (optional)

26.8.8 Widget Query Update Structure

Field	Description
`facet`	Name of the facet/constraint to update
`value`	Name of the selected value within the facet to search on
`text`	Text query that is passed directly to server unaltered. Supports all the query syntax of previous build app versions.
`constraintType`	Must be "range", "constraint", or "geo". Determines how the structured query is generated. Map data is always of type geo. Other data can be range or constraint, depending on whether they are range indexes or collections on the server.

26.8.9 Search Results Format

The search results format (REST API)

Field	Description
`total`	Total results returned by the search
`start`	Result number of the first returned result
`page-length`	Number of results returned in this query
`results`	Array of result objects

Field	Description

`facets`	List of all facets and facet-results for the search.
	
	`<facet-name>: {`
	`type: "xs:<data-type>",`
	`facetValues: [`
	`{name: <record name>, count: <int>},`
	`…`
	`]`
	`}`

`report`	String report with information about the query.
`metrics`	Metrics on the search from the server

26.8.10 Facet Structure

\

Field	Description
value	'name' of facet value that is selected and will be used to update the search. For example, in the actors facet of the Oscars app, this might be "Humphrey Bogart"
constraintType	Must be "range", "constraint", or "geo". Determines how the structured query is generated. Map data is always of type geo. Other data can be range or constraint, depending on whether they are range indexes or collections on the server.
geo	The geo property contains one of the following: • `geo.poly` - an array of lat/lng points • `geo.box` - 4 properties: north, south, east, west; whose values are lat/lng points • `geo.circle` - radius and a point object with latitude and longitude.

26.8.11 Internal Widget Events

Event	Description

`selection`	Selection events are caught by the widget container and used to generate queryUpdate events (see controller events). A selection event has an attached selection object of the following form:
	
	`{ facet:`
	`  <facet name>,`
	`  value: <selected value>`
	`}`
	
	To unselect, simply pass a facet with no value property like so:
	
	`{facet: "decade"}`

`page`	Page events are used for adjusting the pagination of the search results. The data passed along with the event has a page property that specifies the page number to display.

27.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement. For evaluation licenses, MarkLogic may provide support on an “as possible” basis.

For customers with a support contract, we invite you to visit our support website at http://support.marklogic.com to access information on known and fixed issues.

For complete product documentation, the latest product release downloads, and other useful information for developers, visit our developer site at http://developer.marklogic.com.

If you have questions or comments, you may contact MarkLogic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.

Combined Product Notices

MarkLogic Server 6.0 and supporting products.
Last updated: August 19, 2013

COPYRIGHT

Copyright © 2013 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

1. Contains SAP BusinessObjects Text Analysis XI from SAP AG. Copyright © 1996-2011. All rights reserved.

2. Highslide Software from Highslide Software Torstein Honsi. All Highslide Software is protected by local and international copyright laws. All rights reserved.

3. Icons developed by Yusuke Kamiyamane. Copyright © 2011 Yusuke Kamiyamane. All rights reserved. Icons are licensed subject to http://creativecommons.org/licenses/by/3.0/legalcode.

4. Antenna House OfficeHTML Copyright © 2000-2008 Antenna House, Inc. All rights reserved.

5. Argus Copyright ©1999-2008 Iceni Technology Ltd. All rights reserved.

6. Rosette Linguistics Platform 6.5.2/6.5.3 from Basis Technology Corporation, Copyright © 2004-2008 Basis Technology Corporation. All rights reserved.

7. ISYS Search. Copyright © 2011 ISYS™ Search Software, Inc. All rights reserved.

8. Software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/) Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved. Copyright © 1998-2011 The OpenSSL Project. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)" 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. 5. Products derived from this

software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. 6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)" THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Additional terms may apply to the foregoing software as further set forth at http://www.openssl.org/source/license.html

9. ICU v. 4.2.1 available from http://site.icu-project.org// Copyright © 1995-2011 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

10. Tidy available at http://www.w3.org/People/Raggett/tidy/. Copyright © 1998-2008 World Wide Web Consortium (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

11. 'zlib' general purpose compression library v. 1.2.3. Copyright © 1995-2010 Jean-loup Gailly and Mark Adler. http://www.zlib.net/zlib_license.html

12. RSA Data Security, Inc. MD5 Message-Digest Algorithm, copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

13. TRE software, available at http://laurikari.net/tre/. Copyright © 2001-2009 Ville Laurikari <vl@iki.fi>. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

14. Eigen Library, version 2.0.10, available at http://eigen.tuxfamily.org/index.php?title=Main_Page. Eigen Library is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. Eigen Library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with Eigen Library. If not, see: http://www.gnu.org/copyleft/lesser.html

15. FunctX XQuery Library v.1.0 available at http://www.xqueryfunctions.com/xq/, Copyright © Datypic (Priscilla Walmsley). You can redistribute such library and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the

hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License as available at http://www.gnu.org/licenses/lgpl-2.1.html#SEC4 for more details.

16. DeployJava.js v. 1.8, Copyright © 2006, 2011, Oracle and/or its affiliates. All rights reserved. ORACLE PROPRIETARY/CONFIDENTIAL. Use of the foregoing library is subject to the following license terms. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Oracle nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

17. YUI libraries v.2.8 available at http://developer.yahoo.com/yui/. Copyright © 2011 Yahoo! Inc. All rights reserved. Use of the foregoing libraries is subject to the following license terms. Redistribution and use of this software in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Yahoo! Inc. nor the names of YUI's contributors may be used to endorse or promote products derived from this software without specific prior written permission of Yahoo! Inc. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING

NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Additional notices related to components included in the foregoing library are also available at the following URL: http://yuilibrary.com/license/

18. XSLTForms libraries available at http://sourceforge.net/projects/xsltforms/ . You can redistribute such libraries and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. These libraries are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License as available at http://www.gnu.org/licenses/lgpl-2.1.html#SEC4 for more details.

19. jQuery v.1.7 libraries available at http://jquery.com/ and jQuery UI v.1.8.2 libraries available at http://jqueryui.com/. Copyright © 2011 John Resig. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OFMERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE ANDNONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BELIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTIONOF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTIONWITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

20. JSON v. 2 libraries available at http://www.json.org/json2.js. Copyright © 2002 JSON.org. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The Software shall be used for Good, not Evil. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

21. CodeMirror libraries available at http://codemirror.net/. Copyright © 2011 by Marijn Haverbeke marijnh@gmail.com. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

22. History.js available https://github.com/balupton/History.js/ Copyright © 2011, Benjamin Arthur Lupton. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Benjamin Arthur Lupton nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

23. Sample content provided for use with the software is copyrighted by the respective authors as contributed by each of them to Wikipedia pursuant to the Creative Commons Attribution-ShareAlike License found at http://creativecommons.org/licenses/by-sa/3.0/.

24. Visual Studio Tools for Office 3.0 (Microsoft VSTO 3.0) licensed under Microsoft Public License Microsoft Public License (Ms-PL) found at http://msdn.microsoft.com/en-us/library/ff647676.aspx.

25. The MarkLogic Connector for Hadoop contains Apache Jakarta Commons Modeler. Copyright 2001-2007 The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

26. jemalloc http://www.canonware.com/jemalloc/index.html, released under the terms of the following BSD-derived licenses: Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice(s), this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice(s), this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER(S) ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER(S) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Portions of the software are copyright (C) 2002-2013 Jason Evans, and portions copyright (C) 2009-2013 Facebook, Inc. and copyright (C) 2007-2012 Mozilla Foundation. Neither the name of Facebook, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. All rights reserved.

27. Gziphelper; version: 1.0, Feb, 2003, authored by Gao Dasheng, is distributed subject to the following terms: This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software. Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions: (1.) the origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required; (2) altered source versions must be plainly marked as such, and must not be misrepresented as being the original software; (3) this notice may not be removed or altered from any source distribution.

28. Some features of MarkLogic allow you to develop and use Google Maps. Review Google Map's privacy policy at http://www.google.com/policies/privacy/, legal notices at http://maps.google.com/help/legalnotices_maps.html, and terms of use at http://maps.google.com/help/terms_maps.html.

29. commons-codec, HttpComponents, commons-logging, and Google Maps Utility are used under the Licensed under the Apache License, Version 2.0 (the "License"). You may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

30. Mapstraction. Copyright (c) 2012, Mapstraction. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

31. logback-classic and logback-core are provided in object code form only under the Eclipse Public License - v 1.0 available at http://www.eclipse.org/legal/epl-v10.html. Copyright (C) 1999-2012, QOS.ch. All rights reserved. Source code for the Program is available http://logback.qos.ch/.

32. Simple Logging Facade for Java (SLF4J). Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

33. cURL v. 7.22.0 available at http://curl.haxx.se/. Copyright © 1996 - 2011, Daniel Stenberg, <daniel@haxx.se>. All rights reserved. Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

34. heatmap.js distributed subject to the "THE BEER-WARE LICENSE" (Revision 42): <phk@FreeBSD.ORG> wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy me a beer in return Poul-Henning Kamp.

35. Except for components that are otherwise subject to other open source licenses, the MarkLogic ODBC Driver is released under the GNU Lesser General Public License as published by the Free Software Foundation, version 2.1 of the License.

36. Solr-commons-csv-3.5.0 Copyright 2005-2006 The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

37. Avro v. 1.7.1. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

38. Guava v. 11.0.2. Copyright © Google. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

39. Hadoop-auth v. 2.0.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

40. hadoop-hdfs v. 2.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

41. commons-lang v. 2.4, 2.5. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

42. commons-configuration v. 2.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

43. jackson-core-asl v. 1.8.8. Copyright © FasterXML. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

44. jackson-mapper-asl v. 1.8.8. Copyright © FasterXML. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

45. log4j v. 1.2.16. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

46. protobuf-java v. 2.4.0a. Copyright © Google. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Google nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

47. xpp3 v. 1.1.3.3. Copyright © 2002 Extreme! Lab, Indiana University. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab(http://www.extreme.indiana.edu/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. The names "Indiana Univeristy" and "Indiana Univeristy Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/. Products derived from this software may not use "Indiana Univeristy" name nor may "Indiana Univeristy" appear in their name, without prior written permission of the Indiana University. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IFADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

48. xstream v. 1.4.2. Copyright © 2003-2006, Joe Walnes. Copyright © 2006-2009, 2011 XStream Committers. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of XStream nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR

BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

TRADEMARK NOTICE

The MarkLogic name and logo are registered trademarks of MarkLogic Corporation in the United States and/or other countries. Excel and PowerPoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Any other names or logos included in the Software, this notice or the documentation are property of the respective trademark owners.